Hanoi University of Science and Technology

School of Information and Communication Technology

Rescue Rover

An IoT-Based Autonomous Rescue Robot System

Project Report

Embedded Systems / IoT Course

Team Members:

Member 1 Name (Student ID)
Member 2 Name (Student ID)
Member 3 Name (Student ID)

Supervisor: Professor Name

January 2025

Declaration

We hereby declare that this project report titled **"Rescue Rover: An IoT-Based Autonomous Rescue Robot System"** submitted to **Hanoi University of Science and Technology** is a record of original work done by us under the guidance of our supervisor.

We further declare that:

1. The work presented in this report has not been submitted elsewhere for any other degree or professional qualification.
2. All sources of information and literature used in this work have been duly acknowledged through appropriate citations and references.
3. The software code, hardware designs, and methodologies presented here are our own original work, except where explicitly stated otherwise.
4. We understand that any instance of academic dishonesty may result in disciplinary action as per the university regulations.

Member 1 Name	Date
Member 2 Name	Date
Member 3 Name	Date

Abstract

Rescue Rover is an IoT-based autonomous rescue robot system designed for search and rescue operations in hazardous environments. This project integrates embedded systems, wireless communication, real-time video streaming, and AI-powered decision-making to create an intelligent remotely-operated vehicle (ROV) capable of navigating dangerous areas and providing visual reconnaissance.

Problem Statement: Search and rescue operations in disaster zones, collapsed structures, and hazardous areas pose significant risks to human responders. Traditional approaches often result in delayed response times and limited situational awareness, potentially compromising rescue outcomes.

Solution: The Rescue Rover system addresses these challenges through a distributed architecture comprising three main components:

- **ESP32-S3 Rover Unit:** An autonomous robot equipped with camera, motor control, and sensors for real-time surveillance and navigation.
- **Gateway/Courier Module:** Python-based middleware handling ESP-NOW packet reception, JPEG frame reassembly, telemetry aggregation, and AI inference bridging.
- **Web Dashboard:** A NiceGUI-based interface providing live video feed, mission logging, telemetry monitoring, and remote control capabilities.

Key Features:

- Real-time video streaming via MJPEG over HTTP
- Low-latency control using ESP-NOW protocol
- AI-assisted navigation with YOLOv8 object detection
- Vision Language Model (VLM) integration for intelligent decision-making
- Telemetry monitoring (battery, distance, connectivity status)

- Evidence collection and mission data retrieval

Technical Stack:

- **Hardware:** ESP32-S3, OV2640 camera, L298N motor driver
- **Firmware:** C++ with Arduino framework
- **Backend:** Python with NiceGUI and FastAPI
- **AI:** YOLOv8, Moondream2/MLX Vision Language Model
- **Protocols:** ESP-NOW, HTTP, WebSocket

Keywords: IoT, Embedded Systems, ESP32, Autonomous Robot, Computer Vision, Search and Rescue, ESP-NOW, Real-time Streaming, AI Navigation

Acknowledgments

We would like to express our sincere gratitude to all those who have contributed to the successful completion of this project.

First and foremost, we extend our heartfelt thanks to our supervisor, [**Professor Name**], for their invaluable guidance, continuous support, and constructive feedback throughout the development of the Rescue Rover project. Their expertise in embedded systems and IoT has been instrumental in shaping our technical approach.

We are grateful to the **School of Information and Communication Technology** at Hanoi University of Science and Technology for providing us with the necessary resources, laboratory facilities, and academic environment to conduct this research.

Special thanks to our fellow classmates and lab mates who provided helpful discussions, testing support, and moral encouragement during challenging phases of development.

We also acknowledge the open-source community for their contributions to the tools and libraries we used in this project, including the Arduino framework, NiceGUI, YOLOv8, and various Python packages that made our implementation possible.

Finally, we thank our families for their unwavering support and understanding during the long hours dedicated to this project.

— The Rescue Rover Team

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Full Form
ADC	Analog-to-Digital Converter
GPIO	General Purpose Input/Output
I^2C	Inter-Integrated Circuit
LED	Light Emitting Diode
MCU	Microcontroller Unit
PCB	Printed Circuit Board
PSRAM	Pseudo Static Random Access Memory
PWM	Pulse Width Modulation
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
ESP-NOW	Espressif's Proprietary Wireless Protocol
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
MJPEG	Motion JPEG
REST	Representational State Transfer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WiFi	Wireless Fidelity
AI	Artificial Intelligence
CNN	Convolutional Neural Network
CV	Computer Vision
FPS	Frames Per Second
LLM	Large Language Model
ML	Machine Learning
VLM	Vision Language Model
YOLO	You Only Look Once
API	Application Programming Interface
IoT	Internet of Things

Abbreviation	**Full Form**
ROV	Remotely Operated Vehicle
SDK	Software Development Kit
UI	User Interface
OV2640	OmniVision 2640 Camera Sensor
L298N	Dual H-Bridge Motor Driver IC
ToF	Time of Flight
Hz	Hertz
kHz	Kilohertz
MHz	Megahertz
mA	Milliampere
mAh	Milliampere-hour
V	Volt

Chapter 1

Introduction

Search and rescue operations in disaster environments present extreme challenges for human responders. Collapsed buildings, toxic gases, and unstable structures create conditions where sending people is dangerous or impossible. Small robotic platforms offer an alternative for initial reconnaissance and victim location. This thesis presents the design and implementation of a low cost autonomous rescue rover capable of remote operation and basic AI assisted navigation.

1.1 Problem Statement

After earthquakes, building collapses, or industrial accidents, first responders must quickly assess the situation and locate survivors. Human access to damaged structures carries significant risk. Aftershocks can trigger secondary collapses. Dust and debris create respiratory hazards. Confined spaces limit visibility and mobility.

Commercially available rescue robots exist, but their costs range from tens of thousands to hundreds of thousands of dollars. This price point places them beyond reach for most local emergency services, particularly in developing regions. A need exists for low cost platforms that can perform basic reconnaissance using readily available components.

1.1.1 Challenges in Rescue Robotics

Rescue environments present specific technical challenges that distinguish them from other mobile robotics applications.

Unpredictable terrain includes rubble, debris, inclines, and gaps. Wheeled platforms frequently become stuck. Tracked designs offer improved traction but add mechanical complexity.

Limited communications result from building materials blocking radio signals and network infrastructure being damaged. Systems must operate with degraded connectivity and handle intermittent link loss gracefully.

FIGURE: Post-disaster Scene
Shows collapsed structure with rubble and confined spaces

Figure 1.1: Typical post-disaster environment where robotic reconnaissance would be valuable.

Power constraints limit mission duration. Batteries add weight, reducing payload capacity. More energy-dense batteries add cost. A balance must be struck between runtime and portability.

Situational awareness requires more than raw sensor data. Operators viewing a camera feed may struggle to maintain orientation in unfamiliar environments. AI assistance can help by interpreting scenes and highlighting relevant details.

FIGURE: Rescue Robotics Challenges
Four-panel diagram showing terrain, comms, power, and awareness challenges

Figure 1.2: The four primary challenges facing rescue robotics platforms.

1.2 Project Objectives

This project aims to design and build a functional rescue rover prototype within academic budget constraints. The objectives are divided into primary goals and secondary goals.

1.2.1 Primary Objectives

1. **Remote video operation**: Stream live video from the rover to an operator station over WiFi. The operator should see what the rover sees with acceptable latency (under 200 milliseconds).

2. **Reliable motor control**: Respond to operator commands with predictable behavior. Forward, backward, left turn, and right turn movements must function consistently.

3. **Basic obstacle avoidance**: Detect obstacles directly ahead using ultrasonic sensing. Automatically prevent forward movement when an obstacle is too close.

4. **Object detection**: Use computer vision to identify people and common objects in the video stream. Display detection results to the operator in real time.

1.2.2 Secondary Objectives

1. **Scene understanding**: Integrate a Vision Language Model to interpret what the camera sees and suggest navigation actions.

2. **Telemetry monitoring**: Display battery voltage and connection status to the operator.

3. **Mission logging**: Record operator actions and AI observations for post-mission review.

1.3 Scope and Constraints

1.3.1 What This Project Covers

This project encompasses the complete development cycle of an integrated robotic system. The work includes mechanical assembly from commercial chassis components, electrical integration of sensors and actuators, embedded firmware development for the ESP32-S3 microcontroller, Python application development for the host computer, and integration of pre-trained AI models for detection and scene understanding.

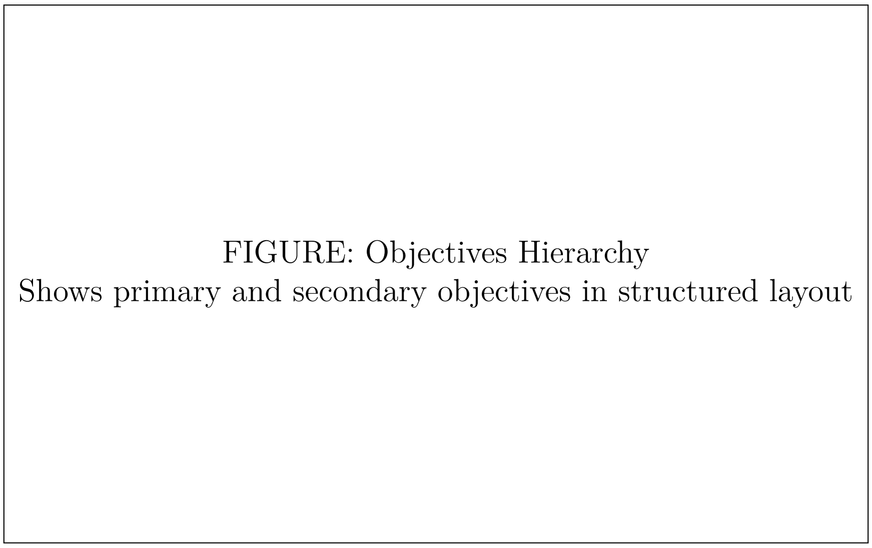

Figure 1.3: Project objectives organized by priority.

The project produces a functional prototype suitable for demonstration in controlled environments. All software is original except for standard libraries and pre-trained models.

1.3.2 What This Project Does Not Cover

Certain aspects lie outside the scope of this project due to time and resource constraints.

Custom mechanical design: The chassis uses a commercially available platform. No custom frames, bodies, or mechanisms were designed or fabricated.

Custom electronics: No printed circuit boards were designed. All electronics use development boards and modules connected with jumper wires.

AI model training: The computer vision model (YOLOv8) and vision language model (Qwen2.5-VL) use pre-trained weights. No custom training was performed.

Outdoor operation: The prototype is designed for indoor environments on relatively flat surfaces. Outdoor terrain, weather protection, and GPS navigation are not addressed.

Multi-robot coordination: The system operates as a single unit. Fleet management and cooperative behaviors are not implemented.

1.3.3 Design Constraints

1.4 Methodology

The project follows an iterative development methodology with frequent testing. Rather than completing all design before implementation, working prototypes were built early and refined based on testing results.

Table 1.1: Project constraints

Category	Constraint
Budget	Maximum \$100 USD for all components
Timeline	16 weeks from concept to demonstration
Tools	Standard hand tools only (no CNC, laser cutter, or 3D printer required)
Software	Open source or freely available tools only
Documentation	Complete reproducibility with public information

1.4.1 Development Phases

Phase 1 (Weeks 1-3): Component Selection and Procurement. Research available microcontrollers, sensors, and chassis options. Order components with lead time consideration.

Phase 2 (Weeks 4-6): Hardware Assembly. Assemble the chassis. Install motors and tracks. Wire power distribution and motor control circuits.

Phase 3 (Weeks 7-10): Firmware Development. Implement camera streaming. Implement motor control. Implement ESP-NOW communication. Implement safety mechanisms.

Phase 4 (Weeks 11-13): Host Application Development. Build the operator dashboard. Integrate video reception. Integrate serial communication. Integrate AI models.

Phase 5 (Weeks 14-16): Integration and Testing. System integration testing. Performance measurements. Bug fixes and optimization.

1.4.2 Testing Strategy

Testing occurred at three levels corresponding to the integration hierarchy.

Unit testing verified individual modules in isolation. Motor control was tested before integration with communication. Camera streaming was tested before integration with the host application.

Integration testing verified interactions between modules. The complete wireless chain from joystick input to motor response was tested end to end.

System testing verified the complete system in representative scenarios. The rover was driven through obstacle courses while monitoring all telemetry.

FIGURE: Project Timeline Gantt Chart
Shows overlapping phases across 16 week period

Figure 1.4: Project timeline showing development phases.

1.5 Related Work

Several research efforts and commercial products address similar problem domains. This section reviews relevant prior work.

1.5.1 Commercial Rescue Robots

PackBot by Endeavor Robotics (now part of FLIR) represents the high end of commercial rescue platforms. Deployed after the September 11 attacks and in numerous military applications, PackBot features manipulator arms, sophisticated sensors, and ruggedized construction. Unit cost exceeds $100,000.

Throwbot by Recon Robotics is a smaller platform designed to be thrown into buildings. Its smaller size and lower capability bring cost down to approximately $10,000, still beyond most academic budgets.

1.5.2 Academic Research

Research platforms for rescue robotics often focus on specific capabilities rather than complete systems.

Quince developed at Chiba Institute of Technology is a snake-like robot designed for confined space navigation. Its segmented body can articulate around obstacles.

Kenaf from the same institute uses front flippers that assist in climbing rubble piles. The bipedal flipper mechanism allows transitions between tracked locomotion and climb-

FIGURE: Commercial Rescue Robots
Side-by-side images of PackBot and Throwbot

Figure 1.5: Commercial rescue robots: PackBot (left) and Throwbot (right).

ing gaits.

1.5.3 Open Source Projects

The maker community has produced several low cost robotics platforms, though few target rescue applications specifically.

ROSbot by Husarion combines an ESP32 with ROS (Robot Operating System) integration. It provides a more sophisticated software stack than this project but at higher cost.

ESP32-CAM projects on various hobbyist sites demonstrate camera streaming with the original ESP32-CAM module. Most lack motor control and are intended as stationary security cameras.

1.6 Contributions

This project makes several contributions to the intersection of embedded systems, computer vision, and rescue robotics.

1. **Complete system integration**: Unlike many projects that demonstrate individual components, this work integrates all subsystems into a functioning prototype.

2. **Hybrid intelligence architecture**: The layered approach combining reactive firmware safety, tactical object detection, and strategic scene understanding is carefully documented.

3. **Budget optimization**: Component selection rationale and trade-off analysis help others build similar systems within tight budgets.

4. **Reproducible documentation**: Complete wiring diagrams, pin mappings, and source code enable others to replicate the project.

1.7 Report Organization

This report is organized into seven chapters plus appendices.

Chapter 2: System Architecture presents the overall design, communication protocols, and the hybrid intelligence model.

Chapter 3: Mechanical and Electrical Design documents the hardware components, power distribution, and physical assembly.

Chapter 4: Embedded Firmware Design covers the ESP32-S3 firmware including camera streaming, motor control, and safety mechanisms.

Chapter 5: AI and Software Design describes the host application, computer vision integration, and vision language model usage.

Chapter 6: Testing and Results presents experimental measurements including latency, accuracy, and reliability tests.

Chapter 7: Conclusion and Future Work summarizes achievements and outlines potential improvements.

Appendices contain detailed pinout tables, circuit diagrams, source code excerpts, and the user manual.

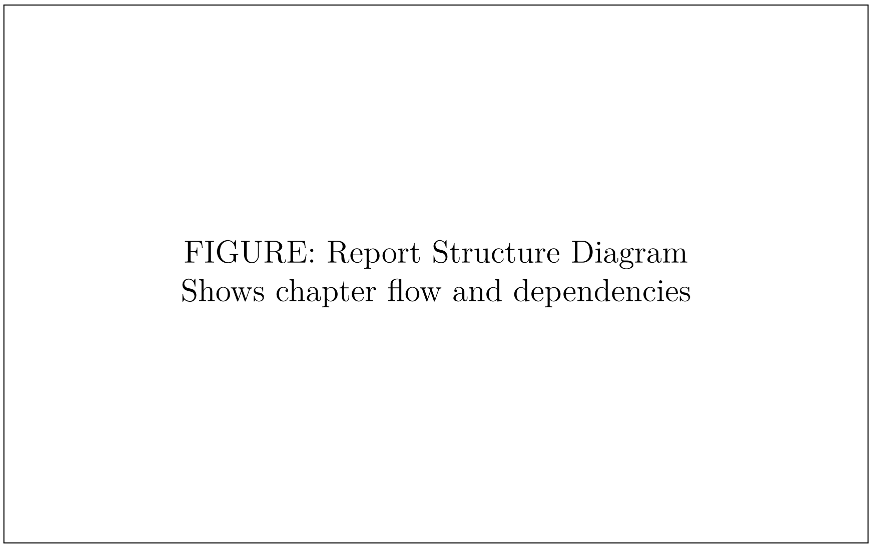

Figure 1.6: Report structure showing the relationship between chapters.

Chapter 2

System Architecture

This chapter presents the overall system architecture of the Rescue Rover. The design follows a distributed intelligence model spread across three physical tiers and one cloud tier. This hybrid approach allows high-performance AI inference without compromising local real-time control.

2.1 Architectural Overview

The system consists of four distinct processing nodes connected through a hierarchy of communication channels.

1. **The Rover (Edge Tier 1)**: ESP32-S3 microcontroller. Handles motor control, sensor readings, and video capture. It has no AI capability but provides 10ms-response safety reflexes.

2. **The Gateway (Edge Tier 2)**: ESP32 bridge. Translates between the Rover's wireless ESP-NOW protocol and the Host's USB serial connection.

3. **The Host Computer (Edge Tier 3)**: MacBook Pro. Runs the "Tactical" AI layer (YOLOv8) for immediate object detection and hosts the operator dashboard. It serves as the local command center.

4. **The Cloud Server (Cloud Tier)**: Google Colab via Google Cloud Platform. Runs the "Strategic" AI layer (Qwen2.5-VL-7B) on NVIDIA A100 GPUs. It handles complex scene reasoning that is too heavy for the local host.

FIGURE: Hybrid Cloud System Overview
Shows Rover, Gateway, Host (Mac), and Cloud (Colab) with links

Figure 2.1: High level system architecture showing the four processing nodes and their data links.

Table 2.1: Device roles and capabilities

Node	**Hardware**	**Responsibilities**
Rover	ESP32-S3 + OV2640	Video capture, motor actuation, reflex safety (sonar)
Gateway	ESP32 WROOM	Protocol translation (ESP-NOW $\leftrightarrow$ Serial)
Host	Apple Silicon Mac	Dashboard UI, Tactical AI (YOLO), Command Arbitration
Cloud	NVIDIA A100 GPU	Strategic AI (Vision Language Model)

2.1.1 Device Specifications Summary

2.2 Hybrid Intelligence Model

The system implements a three-layer intelligence architecture that physically separates "reflex" from "reasoning".

FIGURE: Hybrid Intelligence Pyramid
Layer 1: Firmware, Layer 2: Local AI, Layer 3: Cloud AI

Figure 2.2: The three-layer hybrid intelligence model distributed across Edge and Cloud.

2.2.1 Layer 1: Reactive Control (Firmware Level)

The reactive layer runs on the ESP32-S3 firmware. It handles immediate threat preservation.

- **Input**: Ultrasonic distance sensor.
- **Action**: Hard E-STOP if distance $<$ 25cm.
- **Latency**: $<$ 10ms.
- **Reliability**: 100% (Works even if WiFi/Host fails).

2.2.2 Layer 2: Tactical Processing (Local Host)

The tactical layer runs on the local Mac using YOLOv8 (CoreML). It handles dynamic obstacles.

- **Input**: Video stream (30 FPS).
- **Action**: "Stop for Person", "Avoid Chair".
- **Latency**: $\sim$30ms.
- **Reliability**: High dependency on WiFi video stream.

2.2.3 Layer 3: Strategic Planning (Cloud A100)

The strategic layer runs on Google Colab using Qwen2.5-VL-7B. It handles complex navigation logic.

- **Input**: Single frame snapshot (sampled at 0.5Hz).
- **Action**: "The path is blocked by debris, turn around and try the left door."
- **Latency**: 500-1500ms.
- **Reliability**: Subject to Internet connectivity. Fails gracefully if disconnected.

2.3 Communication Protocols

The hybrid architecture introduces internet-layer communication to the stack.

Table 2.2: Communication protocols by link

Link	Protocol	Latency	Data Type
Rover ↔ Gateway	ESP-NOW	2-5 ms	Commands, Telemetry
Rover → Host	UDP	80-120 ms	MJPEG Video Stream
Gateway ↔ Host	USB Serial	5 ms	Bridge Data
Host ↔ Cloud	HTTP (ngrok)	200-500 ms	JSON Request/Response

2.3.1 Cloud Link (HTTP over ngrok)

The Host communicates with the Cloud Server using standard HTTP POST requests securely tunneled via `ngrok`. The Host sends a JPEG image and prompts; the Cloud returns a JSON object with navigation instructions. This request-response cycle occurs asynchronously to avoid blocking the real-time control loop.

2.4 Failure Modes and Recovery

The introduction of a Cloud dependency adds a specific failure mode: Internet Disconnection.

The design ensures that **internet loss does not compromise safety**. The Rover can still be controlled manually or stop automatically for obstacles via Layer 1/2, even if the "Brain" (Layer 3) is offline.

FIGURE: Hybrid Data Flow
Shows local high-speed loop vs. remote low-speed loop

Figure 2.3: Data flow diagram comparing the high-frequency local control loop and the low-frequency cloud analysis loop.

Table 2.3: Hybrid Failure Matrix

Failure	**Impact**	**Recovery**
Local WiFi Loss	Video/Control lost	Rover stops (Heartbeat fail-safe)
Internet Loss	Strategic IQ lost	Fallback to "Tactical Only" mode (YOLO remains active)
Cloud Latency Spike	Old commands	Arbiter ignores stale Cloud commands ($>$ 2s old)

Chapter 3

Mechanical & Electrical Design

This chapter describes the hardware components, electrical systems, and physical assembly of the Rescue Rover. The design prioritizes modularity, repairability, and use of readily available components. Each subsystem is documented with specifications, selection rationale, and integration considerations.

3.1 Design Philosophy

The hardware design follows three guiding principles. First, all components must be purchasable from common suppliers without lead times. Second, the system must be repairable using basic tools and without specialized equipment. Third, interfaces between modules must use standard connectors and protocols to allow future upgrades.

These constraints led us to select development boards over custom PCBs, through-hole components over surface mount where possible, and dupont wire connections over soldered joints for early prototypes.

3.2 Chassis Design

The chassis provides the structural foundation for all electronic components. We selected a commercially available tracked chassis platform rather than designing a custom frame. This decision saved significant development time while providing a proven mechanical platform.

3.2.1 Platform Selection

The selected chassis is a two-tracked tank-style platform with aluminum frame and plastic track segments. The tracked design was chosen over wheeled alternatives because tracks provide better traction on debris and can navigate over small obstacles that would stop wheels.

FIGURE: Complete Rover Photograph
Top-down view showing all components mounted on chassis

Figure 3.1: Fully assembled Rescue Rover showing the ESP32-S3 camera module, motor driver, battery, and chassis.

Table 3.1: Chassis specifications

Parameter	**Value**
Dimensions (L x W x H)	150mm x 120mm x 45mm
Weight (chassis only)	280g
Material	Aluminum frame, ABS tracks
Track width	25mm
Ground clearance	15mm
Maximum load capacity	1.5 kg
Motor mount	Integrated DC motor brackets

3.2.2 Track Configuration

The tracks use a timing belt design with tensioning idlers at each end. Tension is adjustable through slotted mounting holes. Proper tension is critical for reliable operation. Too loose and the track slips under load. Too tight and motor current increases significantly.

FIGURE: Chassis Dimensions Drawing
Technical drawing with all measurements labeled

Figure 3.2: Chassis dimensional drawing showing mounting hole positions and overall dimensions.

FIGURE: Track Detail Photograph
Close-up showing track teeth, drive sprocket, and tensioner

Figure 3.3: Track assembly detail showing the drive sprocket, track segments, and tension adjustment mechanism.

3.2.3 Component Mounting

Electronic components are mounted to the chassis using nylon standoffs and M3 screws. This approach allows easy removal for debugging and replacement. Components are arranged to balance weight distribution between the front and rear axles.

FIGURE: Component Layout Diagram
Top-down view showing positions of ESP32-S3, L298N, battery, ultrasonic sensor

Figure 3.4: Component layout showing the physical arrangement of electronics on the chassis.

3.3 Main Controller: ESP32-S3

The ESP32-S3 serves as the central processing unit for the rover. This chip was selected for its combination of processing power, wireless capabilities, camera interface, and extensive Arduino ecosystem support.

3.3.1 Chip Specifications

The ESP32-S3 is Espressif's third generation WiFi/Bluetooth microcontroller. It features dual Xtensa LX7 cores running at up to 240 MHz, significantly more powerful than the original ESP32.

3.3.2 Development Board Selection

Several ESP32-S3 development boards were evaluated. The Freenove ESP32-S3 WROOM was selected for its integrated camera connector, adequate PSRAM, and reasonable price point. Other options considered included the official Espressif DevKitC and the AI-Thinker ESP32-S3 CAM.

The ESP32-CAM was rejected despite its lower cost because of severe GPIO limitations. Many pins are shared between the camera, SD card, and flash, leaving insufficient pins for motor control and sensors.

Table 3.2: ESP32-S3 technical specifications

Feature	**Specification**
CPU	Dual Xtensa LX7, up to 240 MHz
SRAM	512 KB internal
PSRAM	8 MB external (OPI interface)
Flash	16 MB (QSPI)
WiFi	802.11 b/g/n, 2.4 GHz
Bluetooth	LE 5.0 with coded PHY
GPIO	45 programmable pins
Camera	DVP interface, up to 2MP
USB	USB 2.0 OTG
Operating voltage	3.0-3.6V (LDO regulates from 5V)

FIGURE: ESP32-S3 Module Photograph
Shows the development board with camera connector and USB-C port

Figure 3.5: ESP32-S3 WROOM development board used in the Rescue Rover.

Table 3.3: ESP32-S3 development board comparison

Feature	**Freenove**	**ESP32-CAM**	**DevKitC**
PSRAM	8 MB	4 MB	8 MB
Camera connector	Yes	Yes	No
Available GPIO	35+	10	45
USB programming	Built-in	External	Built-in
Price	$12	$8	$15
Selected	Yes	No	No

3.3.3 PSRAM Importance

External PSRAM is essential for camera operation. Each QVGA frame requires 153.6 KB of buffer space. Double buffering doubles this requirement. Without PSRAM, the internal 512 KB SRAM would be exhausted by camera buffers alone, leaving no memory

FIGURE: Development Board Comparison
Side-by-side photos of evaluated boards with GPIO counts labeled

Figure 3.6: Comparison of evaluated development boards showing physical differences and GPIO availability.

for the application.

The PSRAM connects via OPI (Octal Peripheral Interface) rather than QPI (Quad). OPI provides higher bandwidth, supporting faster frame transfers from the camera peripheral to the CPU.

3.4 Camera System

The camera provides the rover's primary sensing capability. Visual data is used for remote operation, obstacle detection, and AI scene analysis.

3.4.1 OV2640 Sensor

The OV2640 is a 2-megapixel CMOS image sensor commonly used in embedded vision applications. It connects to the ESP32-S3 through the DVP (Digital Video Port) parallel interface.

3.4.2 Resolution Selection

The firmware configures the camera for QVGA (320 x 240) resolution rather than higher settings. This choice balances image quality against bandwidth and processing requirements. Higher resolutions would exceed UDP packet size limits and require frame fragmentation.

Table 3.4: OV2640 camera sensor specifications

Parameter	Value
Resolution (max)	1600 x 1200 (2 MP)
Output formats	JPEG, YUV422, RGB565
Frame rate (SVGA)	30 FPS
Frame rate (UXGA)	15 FPS
Interface	DVP (8-bit parallel)
Control bus	I2C (SCCB compatible)
Operating voltage	2.5V core, 2.8V I/O
Active pixels	1632 x 1232
Pixel size	2.2 x 2.2 μm

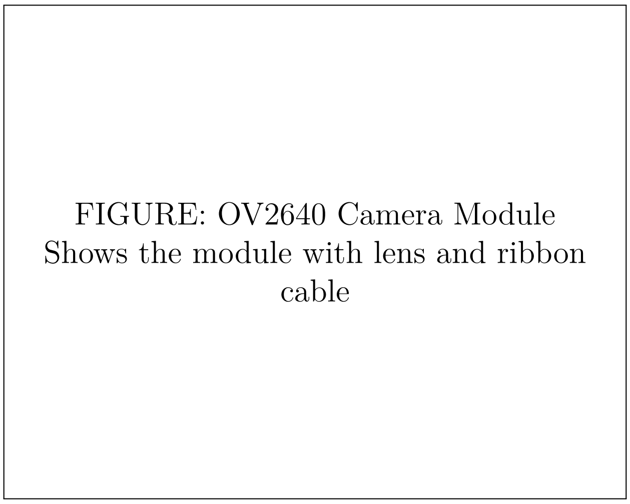

Figure 3.7: OV2640 camera module showing the lens assembly and 24-pin FPC connector.

Table 3.5: Resolution trade-offs

Resolution	Pixels	JPEG Size	UDP OK	Selected
QQVGA (160x120)	19,200	2-5 KB	Yes	No (too small)
QVGA (320x240)	76,800	5-15 KB	Yes	Yes
VGA (640x480)	307,200	20-50 KB	No	No
SVGA (800x600)	480,000	40-80 KB	No	No

3.4.3 Lens and Field of View

The camera module uses a wide-angle lens with approximately 120 degree horizontal field of view. Wide angle coverage is important for obstacle detection at close range. A narrow lens would create blind spots immediately in front of the rover.

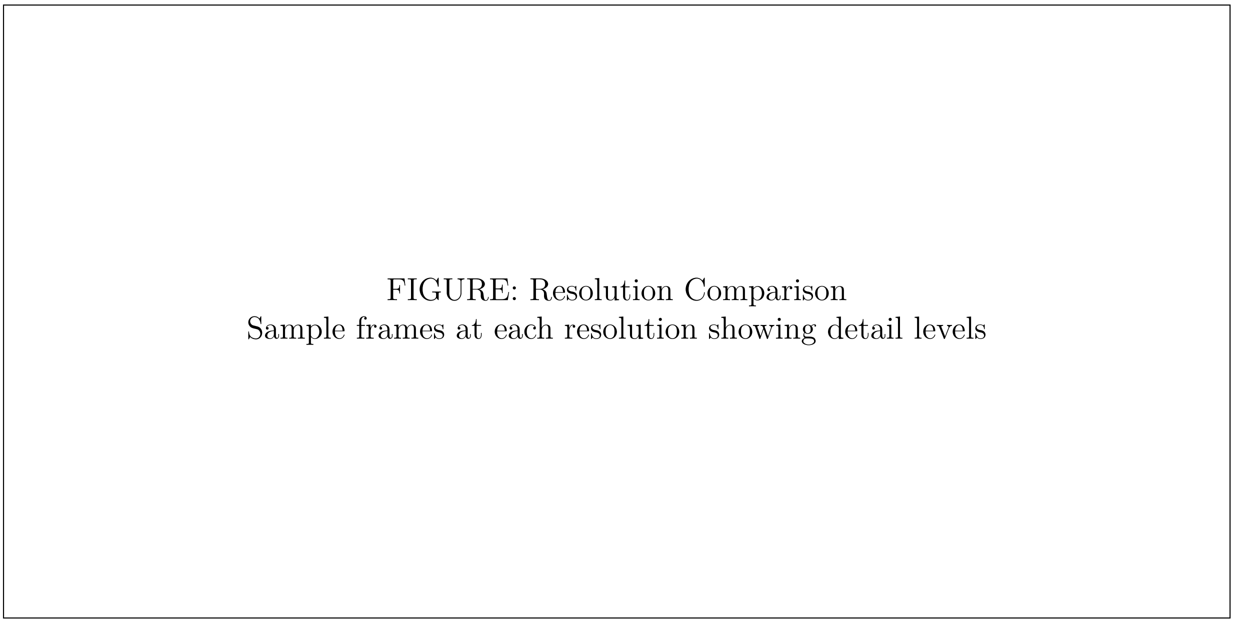

Figure 3.8: Visual comparison of different camera resolutions showing the quality versus size trade-off.

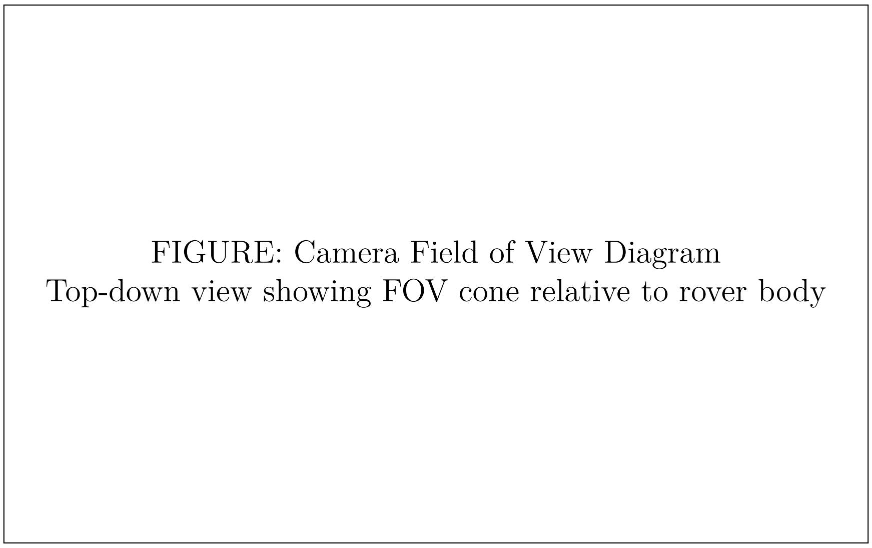

Figure 3.9: Camera field of view coverage showing the visibility area in front of the rover.

3.4.4 Camera Mounting

The camera is mounted at the front of the chassis, tilted downward by approximately 15 degrees. This angle provides visibility of both the immediate ground surface and obstacles at medium distance. The mount uses a 3D printed bracket that allows angle adjustment.

3.5 Motor Control System

The motor system provides locomotive power through two DC gear motors, one for each track. The L298N dual H-bridge driver controls motor direction and speed based on

FIGURE: Camera Mount Detail
Shows 3D printed bracket and tilt angle

Figure 3.10: Camera mounting bracket showing the tilt angle and attachment method.

commands from the ESP32-S3.

3.5.1 DC Gear Motors

The motors are brushed DC type with integrated planetary gearboxes. The gearbox reduces output speed while increasing torque, essential for moving the chassis against friction and over obstacles.

Table 3.6: Motor specifications

Parameter	**Value**
Nominal voltage	6-12V DC
No-load speed	150 RPM at 12V
Stall torque	3 kg-cm
Stall current	1.5 A
Operating current	200-500 mA
Gear ratio	1:48
Shaft diameter	6 mm (D-cut)

3.5.2 L298N Motor Driver

The L298N is a dual full-bridge driver IC. Each bridge can deliver up to 2A continuous current, sufficient for our motors which draw 500mA under load. The module includes an onboard 5V regulator that powers the ESP32-S3.

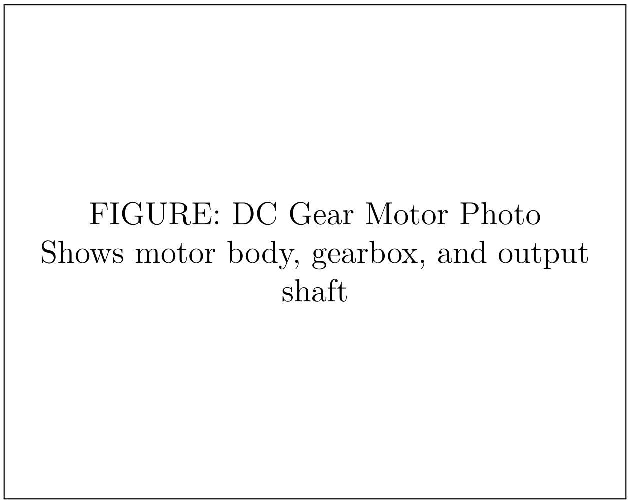

Figure 3.11: DC gear motor showing the motor body, planetary gearbox, and output shaft.

Table 3.7: L298N specifications

Parameter	**Value**
Motor supply voltage	5-35V
Logic supply	5V (from onboard regulator or external)
Per channel current	2A continuous, 3A peak
Total power	25W max
Control inputs	4 (IN1-IN4)
Enable inputs	2 (ENA, ENB)
Voltage drop	1.8V typical (at rated current)

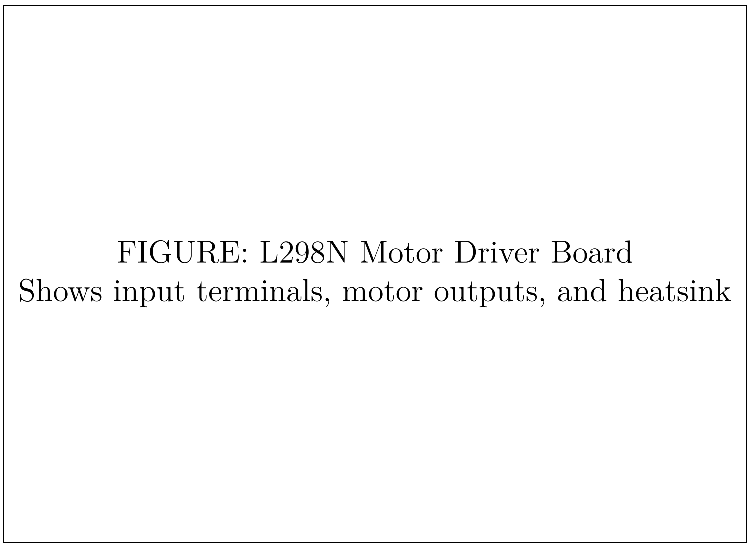

Figure 3.12: L298N motor driver module with the characteristic large heatsink and terminal blocks.

3.5.3 L298N vs TB6612FNG

During component selection, we compared the L298N against the more modern TB6612FNG. The TB6612 offers higher efficiency (90% vs 70%) due to MOSFET switching rather than bipolar transistors. However, the L298N was selected because of its higher current capacity and integrated voltage regulator.

Table 3.8: Motor driver comparison

Feature	**L298N**	**TB6612FNG**
Max current	2A	1.2A
Efficiency	70%	90%
Voltage drop	1.8V	0.3V
Heat generation	High	Low
5V regulator	Built-in	None
Cost	$3	$5
Selected	Yes	No

The lower efficiency of the L298N means more power is dissipated as heat. For extended operation, the heatsink temperature can exceed 60 degrees Celsius. Future revisions may switch to the TB6612 if thermal throttling becomes problematic.

FIGURE: Driver Efficiency Graph
Current vs efficiency curves for both drivers

Figure 3.13: Efficiency comparison between L298N and TB6612FNG at various load currents.

3.6 Power Distribution

The power system delivers appropriate voltages to all components from a single lithium polymer battery. Power management includes voltage regulation, protection circuits, and monitoring.

3.6.1 Battery Selection

A 3-cell (3S) lithium polymer battery provides the primary power. The 11.1V nominal voltage is compatible with both the motors (rated for 6-12V) and the L298N 5V regulator input range.

Table 3.9: Battery specifications

Parameter	**Value**
Chemistry	Lithium Polymer (LiPo)
Configuration	3S (3 cells in series)
Nominal voltage	11.1V
Fully charged	12.6V
Low cutoff	9.9V (3.3V per cell)
Capacity	2200 mAh
Discharge rate	25C continuous
Weight	180g

FIGURE: LiPo Battery Photo
Shows 3S pack with balance connector and XT60 plug

Figure 3.14: 3S LiPo battery used to power the Rescue Rover.

3.6.2 Power Budget

The power budget analysis ensures the battery can supply all loads simultaneously. Total maximum draw is approximately 2.5A, well within the battery's 55A (25C x 2.2Ah) capability.

At typical consumption of 851 mA, the 2200 mAh battery provides approximately 2.5 hours of operation. Under maximum load (both motors stalled), runtime drops to approximately 40 minutes. Actual runtime during normal operation falls between these extremes.

Table 3.10: System power budget

Component	**Voltage**	**Current (typ)**	**Current (max)**
ESP32-S3 + Camera	3.3V	300 mA	500 mA
L298N quiescent	5V	36 mA	50 mA
Left motor	12V	250 mA	1.5 A
Right motor	12V	250 mA	1.5 A
Ultrasonic sensor	5V	15 mA	20 mA
Total	–	851 mA	3.57 A

FIGURE: Power Distribution Schematic
Shows battery, regulator, and all loads with current paths

Figure 3.15: Power distribution schematic showing voltage rails and current flow to all components.

3.6.3 Voltage Regulation

The L298N module includes a 78M05 linear regulator that provides 5V output from the 12V motor supply. This 5V rail powers the ESP32-S3 and ultrasonic sensor. The ESP32's internal LDO then produces 3.3V for the microcontroller core and camera.

3.7 Ultrasonic Distance Sensor

The ultrasonic sensor provides proximity detection for obstacle avoidance. It measures the time of flight for a sound pulse to reach an obstacle and return.

FIGURE: Voltage Rail Diagram
Shows 12V, 5V, and 3.3V rails with regulator symbols

Figure 3.16: Voltage rail hierarchy showing the cascade of regulators from battery to components.

Table 3.11: Ultrasonic sensor specifications

Parameter	**Value**
Operating voltage	5V DC
Operating current	15 mA
Frequency	40 kHz
Range	2 cm to 400 cm
Resolution	0.3 cm
Measuring angle	15 degrees cone
Trigger pulse	10 μs minimum

FIGURE: HC-SR04 Sensor Photo
Shows the sensor with two transducer cylinders

Figure 3.17: HC-SR04 ultrasonic distance sensor showing the transmitter and receiver transducers.

3.7.1 HC-SR04 Specifications

3.7.2 Mounting Position

The sensor is mounted at the front of the chassis, below the camera. This position provides distance measurements along the rover's direction of travel. The narrow 15-degree beam angle means only obstacles directly ahead are detected. Side obstacles require camera based detection.

FIGURE: Ultrasonic Beam Pattern
Top-down view showing 15 degree detection cone

Figure 3.18: Ultrasonic beam pattern showing the narrow detection cone.

3.7.3 Level Shifting

The HC-SR04 operates at 5V logic levels while the ESP32-S3 GPIO pins are 3.3V. The echo pin outputs 5V, which could damage the ESP32 input. A simple resistor voltage divider scales the echo signal down to 3.3V safe levels.

```
R1 = 1kOhm, R2 = 2kOhm
V_out = V_in * R2 / (R1 + R2)
V_out = 5V * 2k / 3k = 3.33V
```

Listing 3.1: Voltage divider calculation

FIGURE: Level Shifter Circuit
Resistor divider schematic for ECHO pin

Figure 3.19: Resistor voltage divider for level shifting the 5V echo signal to 3.3V.

3.8 Wiring and Interconnections

This section documents all electrical connections between components. Dupont jumper wires are used throughout for easy modification during development.

3.8.1 Complete Wiring Diagram

FIGURE: Complete Wiring Diagram
Fritzing-style diagram showing all connections between components

Figure 3.20: Complete wiring diagram showing all connections between the ESP32-S3, L298N, motors, sensors, and power supply.

3.8.2 Wire Color Convention

3.9 Assembly Process

The complete assembly follows a specific order to ensure proper fit and cable routing.

Table 3.12: Wire color coding standard

Color	**Purpose**
Red	Positive power (12V, 5V, 3.3V)
Black	Ground
Yellow	Signal (GPIO outputs)
Orange	Signal (GPIO inputs)
Green	I2C SDA
Blue	I2C SCL

FIGURE: Actual Wiring Photograph
Photo of real rover showing wire routing and organization

Figure 3.21: Actual wiring on the prototype showing careful routing and color coding.

3.9.1 Assembly Steps

1. Mount motors to chassis brackets using M3 screws
2. Attach drive sprockets to motor shafts
3. Install tracks with proper tension
4. Mount battery holder to rear platform
5. Attach L298N driver with standoffs
6. Mount ESP32-S3 board adjacent to L298N
7. Install ultrasonic sensor at front
8. Mount camera module on adjustable bracket
9. Complete all wiring connections

10. Verify connections with multimeter before power on

FIGURE: Assembly Sequence Photos
Grid of 4-6 photos showing key assembly steps

Figure 3.22: Assembly sequence showing the progressive addition of components to the chassis.

3.9.2 Testing Procedure

After assembly, each subsystem is tested individually before full integration testing.

Table 3.13: Post-assembly test checklist

Subsystem	**Test Procedure**
Power	Verify 5V and 3.3V rails with multimeter
Motors	Check rotation direction for each motor separately
Camera	Verify video feed appears in browser
Ultrasonic	Move hand toward sensor, verify distance changes
WiFi	Confirm connection to access point
ESP-NOW	Verify command reception from gateway

FIGURE: Testing Setup Photo
Shows multimeter, test equipment, and rover on bench

Figure 3.23: Testing setup for post-assembly verification of subsystems.

Chapter 4

Embedded Firmware Design

The firmware running on the ESP32-S3 forms the foundation of the Rescue Rover system. This chapter documents the design decisions, implementation details, and technical challenges encountered while developing the embedded software layer. The firmware handles camera streaming, motor control, wireless communication, and safety mechanisms, all running concurrently on the dual-core processor.

4.1 Firmware Architecture Overview

The ESP32-S3 firmware follows a modular architecture where each hardware subsystem is encapsulated in its own compilation unit. This separation allows independent development and testing of each component before integration. The project contains five primary modules, each with a specific responsibility in the system.

FIGURE: Firmware Module Architecture Diagram
Shows relationship between RescueRobot.ino, CameraModule, MotorDriver, ConnectionModule

Figure 4.1: Firmware module architecture showing the main sketch and its dependencies. The main loop coordinates all subsystems while individual modules manage their respective hardware interfaces.

4.1.1 Module Responsibilities

The `RescueRobot.ino` file serves as the main entry point. It initializes all hardware subsystems in a specific order and runs the main control loop at approximately 60Hz. The loop coordinates sensor readings, command processing, safety checks, and telemetry transmission. Total line count for this file is 265 lines.

The `CameraModule` handles all camera operations. This includes initialization of the OV2640 sensor, configuration of frame buffers in PSRAM, and streaming via either HTTP or UDP protocols. The module provides two streaming modes because we discovered during testing that HTTP introduces 80ms additional latency compared to UDP on the same network. Total implementation spans 222 lines across the header and source files.

The `MotorDriver` module controls the L298N H-bridge through GPIO pins. Movement primitives include forward, backward, left turn, right turn, and emergency stop. The implementation uses simple digital writes rather than PWM because the current prototype focuses on directional control rather than speed modulation. This module is intentionally minimal at 56 lines.

The `ConnectionModule` manages ESP-NOW bidirectional communication with the gateway. It receives joystick commands, stores them for processing by the main loop, and transmits telemetry at 2Hz. The critical design decision here is that the receive callback does not directly actuate motors. Instead it stores the command and lets the main loop apply safety checks first. This prevents race conditions between sensor readings and motor commands.

FIGURE: Module Dependency Graph
Shows include relationships between all firmware files

Figure 4.2: Compilation dependencies between firmware modules. The main sketch includes all headers while modules only include what they need.

4.1.2 Memory Layout

The ESP32-S3 provides multiple memory regions with different characteristics. Understanding these regions is essential for camera buffer allocation and overall system stability.

Table 4.1: ESP32-S3 memory regions and their usage in the firmware

Region	**Size**	**Usage in Firmware**
Internal SRAM	512 KB	Stack, heap, global variables, WiFi buffers
PSRAM (External)	8 MB	Camera frame buffers (2 x 320x240), JPEG output buffer
Flash	16 MB	Program code, string constants, WiFi credentials

Camera frame buffers consume the majority of PSRAM. Each QVGA frame requires 320 x 240 x 2 = 153.6 KB in YUV format. We allocate two buffers for double buffering, allowing one to be transmitted while the other captures the next frame. The JPEG compressed output typically ranges from 5 KB to 15 KB depending on scene complexity.

FIGURE: ESP32-S3 Memory Map
Visual representation of SRAM, PSRAM, and Flash allocation

Figure 4.3: Memory allocation for the Rescue Rover firmware showing how camera buffers dominate PSRAM usage.

4.2 Camera Module Implementation

The camera module presented the most significant technical challenges during development. The OV2640 sensor requires precise timing, correct voltage levels, and specific initialization sequences. Several iterations were needed before achieving stable operation.

4.2.1 Hardware Configuration

The OV2640 connects to the ESP32-S3 through a DVP (Digital Video Port) interface. This parallel interface uses 8 data lines (D0-D7), synchronization signals (VSYNC, HREF, PCLK), and an I2C bus for register configuration.

FIGURE: OV2640 Camera Wiring Diagram
Complete wiring between camera module ribbon cable and ESP32-S3 GPIO pins

Figure 4.4: Physical wiring between the OV2640 camera module and ESP32-S3 development board. The ribbon cable connects to a breakout board which then connects to individual GPIO pins.

The pin mapping was derived from the Freenove ESP32-S3 WROOM CAM board schematic. However, our custom wiring required adjustments to avoid conflicts with motor control pins. The final configuration uses GPIO 10 for XCLK, GPIO 40 and 39 for I2C, and GPIO 38 for VSYNC. Data pins span GPIO 11 through 18 and GPIO 48.

```
// ESP32-S3 Camera Pin Mapping (Freenove Compatible)
#define PWDN_GPIO_NUM     -1   // Not connected
#define RESET_GPIO_NUM    -1   // Using software reset
#define XCLK_GPIO_NUM     10   // Master clock output
#define SIOD_GPIO_NUM     40   // I2C SDA
#define SIOC_GPIO_NUM     39   // I2C SCL
#define Y9_GPIO_NUM       48   // Data bit 7
#define Y8_GPIO_NUM       11   // Data bit 6
#define Y7_GPIO_NUM       12   // Data bit 5
#define Y6_GPIO_NUM       14   // Data bit 4
#define Y5_GPIO_NUM       16   // Data bit 3
#define Y4_GPIO_NUM       18   // Data bit 2
#define Y3_GPIO_NUM       17   // Data bit 1
#define Y2_GPIO_NUM       15   // Data bit 0
#define VSYNC_GPIO_NUM    38   // Vertical sync
#define HREF_GPIO_NUM     47   // Horizontal reference
#define PCLK_GPIO_NUM     13   // Pixel clock
```

Listing 4.1: Camera pin configuration from CameraPins.h

4.2.2 Initialization Sequence

Camera initialization follows a strict sequence. Any deviation results in cryptic error codes or complete failure to capture frames. We reduced the XCLK frequency from 20MHz to 10MHz after encountering intermittent initialization failures. The lower clock speed sacrifices theoretical maximum framerate but dramatically improves reliability.

Algorithm 1 Camera Initialization Sequence

```
Check for PSRAM availability
if PSRAM not found then
    Log warning: camera may fail without external memory
end if
Populate camera_config_t structure with pin numbers
Set XCLK frequency to 10 MHz (conservative setting)
Set frame size to QVGA (320 x 240 pixels)
Set pixel format to JPEG with quality level 30
Set frame buffer location to PSRAM
Allocate 2 frame buffers for double buffering
Set grab mode to CAMERA_GRAB_LATEST
Call esp_camera_init() with configuration
if initialization returns error then
    Log error code and halt
    return false
end if
Get sensor handle via esp_camera_sensor_get()
Disable test pattern (colorbar) for real images
Log success message with resolution
return true
```

The JPEG quality parameter inversely correlates with file size. A quality value of 30 produces frames between 5KB and 12KB, suitable for UDP transmission within a single packet on local networks. Lower quality values (higher numbers in the ESP camera API) create smaller files but introduce visible compression artifacts.

4.2.3 Streaming Modes

The firmware supports two distinct streaming protocols. HTTP MJPEG provides browser compatibility and easy debugging. UDP streaming minimizes latency for the AI processing pipeline.

HTTP MJPEG Streaming

The HTTP mode creates a simple web server on port 80 with a single endpoint at `/stream`. When a client connects, the handler enters an infinite loop, capturing frames and sending

FIGURE: Camera Initialization Flowchart
Decision tree showing PSRAM check, config, init, and error handling paths

Figure 4.5: Flowchart for camera initialization showing fail-safe checks and configuration steps.

them with multipart boundaries. This approach works with any web browser and requires no client side code.

```
static esp_err_t stream_handler(httpd_req_t *req) {
    camera_fb_t *fb = NULL;
    esp_err_t res = ESP_OK;
    char part_buf[64];

    // Set content type for MJPEG stream
    res = httpd_resp_set_type(req,
        "multipart/x-mixed-replace;boundary=frame");
    if (res != ESP_OK) return res;

    while (true) {
        fb = esp_camera_fb_get();
        if (!fb) {
            Serial.println("Capture failed");
            res = ESP_FAIL;
        } else {
            // Build multipart header
            size_t hlen = snprintf(part_buf, 64,
                "\r\n--frame\r\n"
                "Content-Type: image/jpeg\r\n"
                "Content-Length: %u\r\n\r\n", fb->len);

            res = httpd_resp_send_chunk(req, part_buf, hlen);
            if (res == ESP_OK) {
                res = httpd_resp_send_chunk(req,
                    (const char*)fb->buf, fb->len);
```

```
            }
            esp_camera_fb_return(fb);
        }
        if (res != ESP_OK) break;
    }
    return res;
}
```

Listing 4.2: HTTP MJPEG stream handler implementation

FIGURE: HTTP MJPEG Sequence Diagram
Shows request/response flow between browser and ESP32-S3

Figure 4.6: Sequence diagram for HTTP MJPEG streaming showing the continuous frame transmission loop.

UDP Streaming

UDP mode sends frames directly without the HTTP overhead. The target IP address and port are configured at compile time. Each frame is transmitted as a single UDP packet, which works reliably on local networks where MTU is 1500 bytes and frames are under 15KB.

```
void streamFrameUDP() {
    // Check prerequisites
    if (!cameraReady || !udpMode || udpTargetIP == nullptr)
        return;

    camera_fb_t *fb = esp_camera_fb_get();
    if (!fb) {
        Serial.println("UDP capture failed");
        return;
    }

    // Send entire frame as single packet
```

```
    // Works on local WiFi where fragmentation is handled
    udp.beginPacket(udpTargetIP, udpTargetPort);
    udp.write(fb->buf, fb->len);
    udp.endPacket();

    esp_camera_fb_return(fb);
}
```

Listing 4.3: UDP frame transmission function

FIGURE: HTTP vs UDP Latency Comparison
Bar chart showing measured latency for both protocols

Figure 4.7: Measured latency comparison between HTTP MJPEG and UDP streaming modes. UDP consistently shows 80ms lower latency.

Table 4.2: Comparison of streaming modes

Characteristic	**HTTP MJPEG**	**UDP Direct**
End to end latency	180-220 ms	100-140 ms
Browser compatible	Yes	No (requires custom receiver)
Packet loss handling	TCP retransmit	None (frames dropped)
Setup complexity	Low	Medium
Network overhead	Higher	Lower
Recommended use	Debugging	Production

4.3 Motor Control Implementation

The motor control subsystem translates high level movement commands into GPIO signals for the L298N driver. The current implementation uses binary on/off control rather than PWM speed modulation. This choice simplifies the code and proved sufficient for indoor testing where precise speed control is less important than reliable direction changes.

4.3.1 Differential Drive Model

The rover uses a differential drive configuration with two independently controlled motors. This arrangement allows the robot to turn by varying the relative speeds of the left and right wheels. Mathematically, the relationship between wheel velocities and robot motion is expressed as:

$$v_{robot} = \frac{v_R + v_L}{2} \tag{4.1}$$

$$\omega_{robot} = \frac{v_R - v_L}{W} \tag{4.2}$$

where v_{robot} is the linear velocity of the robot center, ω_{robot} is the angular velocity, v_R and v_L are the right and left wheel velocities respectively, and W is the wheelbase width.

FIGURE: Differential Drive Kinematics
Diagram showing wheel velocities, turning radius,
and angular velocity

Figure 4.8: Differential drive kinematic model showing how independent wheel speeds create linear and angular motion.

4.3.2 L298N Driver Interface

The L298N accepts four digital inputs (IN1 through IN4) that control two H-bridge circuits. Each motor connects to one H-bridge. The truth table for motor direction is straightforward: setting one input HIGH and the other LOW drives the motor in one direction. Reversing which input is HIGH reverses the rotation.

```
#include "MotorDriver.h"
#include "PinConfig.h"

void initMotors() {
    pinMode(PIN_LEFT_FWD, OUTPUT);
```

```
    pinMode(PIN_LEFT_BWD, OUTPUT);
    pinMode(PIN_RIGHT_FWD, OUTPUT);
    pinMode(PIN_RIGHT_BWD, OUTPUT);

    stopMoving();  // Ensure motors are off at startup
    Serial.println("Motor Driver Ready (GPIO: 4,5,6,7)");
}

void goForward() {
    digitalWrite(PIN_LEFT_FWD, HIGH);
    digitalWrite(PIN_LEFT_BWD, LOW);
    digitalWrite(PIN_RIGHT_FWD, HIGH);
    digitalWrite(PIN_RIGHT_BWD, LOW);
}

void goBackward() {
    digitalWrite(PIN_LEFT_FWD, LOW);
    digitalWrite(PIN_LEFT_BWD, HIGH);
    digitalWrite(PIN_RIGHT_FWD, LOW);
    digitalWrite(PIN_RIGHT_BWD, HIGH);
}

void stopMoving() {
    digitalWrite(PIN_LEFT_FWD, LOW);
    digitalWrite(PIN_LEFT_BWD, LOW);
    digitalWrite(PIN_RIGHT_FWD, LOW);
    digitalWrite(PIN_RIGHT_BWD, LOW);
}
```

Listing 4.4: Motor driver initialization and movement functions

4.3.3 Pin Assignment

The motor pins were chosen to avoid conflicts with camera interface and JTAG debugging. GPIO 4, 5, 6, and 7 are safe choices that do not overlap with any other peripheral.

Table 4.3: Motor driver pin assignment

Function	**L298N Pin**	**ESP32 GPIO**	**Wire Color**
Left Forward	IN1	GPIO 4	Yellow
Left Backward	IN2	GPIO 5	Orange
Right Forward	IN3	GPIO 6	Green
Right Backward	IN4	GPIO 7	Blue

FIGURE: L298N Motor Driver Wiring
Complete schematic showing ESP32-S3 GPIO connections, motor terminals, and power input

Figure 4.9: L298N motor driver wiring diagram showing all connections between ESP32-S3, the driver board, and both DC motors.

4.3.4 Turning Mechanics

Point turns are achieved by running motors in opposite directions. The left motor runs backward while the right motor runs forward for a left turn. This creates rotation around the robot's center point. Arc turns for smoother motion would require PWM speed control, which is planned for a future revision.

FIGURE: Turning Mechanics Diagram
Shows point turn vs arc turn wheel direction arrows

Figure 4.10: Comparison of point turn (current implementation) versus arc turn (future enhancement). Point turns rotate in place while arc turns follow a curved path.

4.4 ESP-NOW Communication Module

The ConnectionModule handles all wireless communication between the rover and the gateway device. ESP-NOW was selected over TCP/IP for control commands because it offers significantly lower latency. The protocol operates at the MAC layer, bypassing much of the WiFi stack overhead.

4.4.1 Protocol Overview

ESP-NOW allows direct device to device communication using MAC addresses. Packets are limited to 250 bytes, but this is more than sufficient for joystick commands (8 bytes) and telemetry (8 bytes). The protocol does not guarantee delivery, but at close range with clear line of sight, packet loss is negligible.

FIGURE: ESP-NOW Protocol Stack
Shows position in networking layers compared to standard WiFi TCP/IP

Figure 4.11: ESP-NOW protocol positioning in the network stack. It operates below IP, directly on top of the 802.11 MAC layer.

4.4.2 Packet Structures

Two packet types define the communication protocol. The command structure carries joystick X and Y values from the gateway to the rover. The feedback structure carries voltage and distance readings from the rover back to the gateway.

```
// Command packet: Gateway -> Rover
typedef struct __attribute__((packed)) command_struct {
    int x;  // Joystick X (0-4095, center=2048)
    int y;  // Joystick Y (0-4095, center=2048)
} command_struct;

// Feedback packet: Rover -> Gateway
```

```c
typedef struct __attribute__((packed)) feedback_struct {
    float voltage;    // Battery voltage in volts
    int distance;     // Ultrasonic distance in cm
} feedback_struct;
```

Listing 4.5: ESP-NOW packet structure definitions

The `__attribute__((packed))` directive ensures no padding bytes are inserted between structure members. This guarantees the structure has the exact same memory layout on both ESP32 devices.

FIGURE: Packet Format Diagram
Shows byte layout of command and feedback structures

Figure 4.12: Binary layout of ESP-NOW packet structures showing byte offsets for each field.

4.4.3 Receive Callback Design

The receive callback executes in an interrupt context when a packet arrives. Long operations in this callback would block the WiFi stack and cause instability. Our initial implementation called motor control functions directly from the callback, which created race conditions with the ultrasonic sensor readings in the main loop.

Critical Design Decision: Deferred Motor Actuation

The receive callback stores commands but does not actuate motors. Motor actuation happens in the main loop after safety checks. This prevents race conditions where an obstacle appears between receiving a "forward" command and executing it. The main loop always has the latest sensor data when deciding whether to execute a command.

```c
// State variables
static command_struct recvCommand = {2048, 2048};  // Center = stop
static unsigned long lastPacketTime = 0;

```

```
// Callback: ONLY stores command, does NOT control motors
static void onDataRecv(const esp_now_recv_info_t *info,
                       const uint8_t *data, int len) {
    if (len != sizeof(command_struct)) {
        Serial.printf("Wrong packet size: %d\n", len);
        return;
    }

    // Store command for main loop to process
    memcpy(&recvCommand, data, sizeof(recvCommand));

    // Update heartbeat timestamp
    lastPacketTime = millis();

    Serial.printf("RX: X=%d Y=%d\n", recvCommand.x, recvCommand.y);
}
```

Listing 4.6: ESP-NOW receive callback with deferred actuation

FIGURE: Callback vs Main Loop Processing
Sequence showing how commands flow from ESP-NOW to motor actuation

Figure 4.13: Data flow from ESP-NOW callback to motor actuation through the main loop safety checks.

4.4.4 Telemetry Transmission

Telemetry is sent at 2Hz (every 500ms) to avoid flooding the wireless channel. The feedback packet contains battery voltage and ultrasonic distance. The gateway forwards this data to the dashboard over USB serial.

```
static unsigned long lastTelemetryTime = 0;
static const unsigned long TELEMETRY_INTERVAL = 500;  // 2Hz

void handleConnection(float voltage, int distance) {
```

```
    unsigned long now = millis();

    // Rate limit to 2Hz
    if (now - lastTelemetryTime >= TELEMETRY_INTERVAL) {
        lastTelemetryTime = now;

        sendFeedback.voltage = voltage;
        sendFeedback.distance = distance;

        esp_err_t result = esp_now_send(
            gatewayMAC,
            (uint8_t*)&sendFeedback,
            sizeof(sendFeedback)
        );

        if (result != ESP_OK) {
            Serial.println("Telemetry send failed");
        }
    }
}
```

Listing 4.7: Telemetry transmission with rate limiting

4.5 Safety Mechanisms

The firmware implements multiple layers of safety to prevent the rover from colliding with obstacles or running away if communication is lost. These mechanisms operate at the firmware level where response time is guaranteed.

4.5.1 Heartbeat Failsafe

If no command packet arrives within 500ms, the rover assumes communication has been lost and stops all motors. This prevents the rover from continuing to execute a stale "forward" command indefinitely. The timeout value was chosen based on the expected packet rate of 20Hz from the joystick.

```
const unsigned long SIGNAL_TIMEOUT = 500;  // 500ms

void handleControlLoop() {
    // Check connection heartbeat
    if (!isConnectionAlive(SIGNAL_TIMEOUT)) {
        static bool signalLostPrinted = false;
        if (!signalLostPrinted) {
            Serial.println("SIGNAL LOST - EMERGENCY STOP!");
            signalLostPrinted = true;
```

```
        }
        stopMoving();
        return;  // Skip all control logic
    }

    // ... rest of control logic
}

bool isConnectionAlive(unsigned long timeoutMs) {
    if (lastPacketTime == 0) return true;  // Allow startup
    return (millis() - lastPacketTime) < timeoutMs;
}
```

Listing 4.8: Heartbeat check in main control loop

FIGURE: Heartbeat State Machine
Shows CONNECTED, WARNING, and STOPPED states with transitions

Figure 4.14: State machine for heartbeat monitoring showing transitions between connected and stopped states.

4.5.2 Ultrasonic Obstacle Detection

The ultrasonic sensor provides distance measurements to the nearest obstacle in the forward direction. The firmware uses this to override forward commands when an obstacle is too close. The implementation uses a state machine rather than the blocking `pulseIn()` function to avoid freezing the main loop.

```
enum UltrasonicState {
    US_IDLE,
    US_TRIGGER_HIGH,
    US_WAIT_ECHO_START,
    US_WAIT_ECHO_END
};

```

```
static UltrasonicState usState = US_IDLE;
static unsigned long usTriggerTime = 0;
static unsigned long usEchoStart = 0;

bool updateUltrasonicDistance() {
    unsigned long now = micros();

    switch (usState) {
    case US_IDLE:
        digitalWrite(TRIG_PIN, HIGH);
        usTriggerTime = now;
        usState = US_TRIGGER_HIGH;
        break;

    case US_TRIGGER_HIGH:
        if (now - usTriggerTime >= 10) {  // 10us trigger pulse
            digitalWrite(TRIG_PIN, LOW);
            usState = US_WAIT_ECHO_START;
        }
        break;

    case US_WAIT_ECHO_START:
        if (digitalRead(ECHO_PIN) == HIGH) {
            usEchoStart = now;
            usState = US_WAIT_ECHO_END;
        } else if (now - usTriggerTime > 30000) {  // 30ms timeout
            currentDistance = 999;  // No object detected
            usState = US_IDLE;
            return true;
        }
        break;

    case US_WAIT_ECHO_END:
        if (digitalRead(ECHO_PIN) == LOW) {
            unsigned long duration = now - usEchoStart;
            currentDistance = (duration * 0.034) / 2;  // Speed of sound
            usState = US_IDLE;
            return true;
        } else if (now - usEchoStart > 30000) {  // Timeout
            currentDistance = 999;
            usState = US_IDLE;
            return true;
        }
        break;
    }
    return false;
```

```
}
```

Listing 4.9: Non-blocking ultrasonic state machine

FIGURE: Ultrasonic State Machine
Shows all states with timing conditions for transitions

Figure 4.15: State machine diagram for non-blocking ultrasonic distance measurement.

4.5.3 Unified Control Loop

The main control loop integrates all sensor inputs and safety checks before executing motor commands. This unified approach ensures that no command is executed without considering the current safety state.

```
const int EMERGENCY_STOP_DISTANCE = 25;  // cm

void handleControlLoop() {
    // 0. Check heartbeat (covered above)

    // 1. Update distance sensor (non-blocking)
    updateUltrasonicDistance();

    // 2. Get latest joystick command
    command_struct cmd = getLastCommand();

    // 3. Is rover trying to move forward?
    bool tryingToMoveForward = (cmd.y > 2500);

    // 4. Safety decision matrix
    if (currentDistance < EMERGENCY_STOP_DISTANCE
        && tryingToMoveForward) {
        // Block forward movement
```

```
        if (!emergencyStopActive) {
            Serial.printf("OBSTACLE at %d cm - BLOCKING!\n",
                currentDistance);
            stopMoving();
            emergencyStopActive = true;
        }

        // But allow backward/turning to escape
        if (cmd.y < 1500 || cmd.x < 1000 || cmd.x > 3000) {
            Serial.println("Escape maneuver allowed");
            executeMotorCommand(cmd.x, cmd.y);
            emergencyStopActive = false;
        }
    } else {
        // Safe to execute command
        emergencyStopActive = false;
        executeMotorCommand(cmd.x, cmd.y);
    }
}
```

Listing 4.10: Unified control loop with safety integration

FIGURE: Unified Control Loop Flowchart
Complete decision tree showing heartbeat, distance, and command processing

Figure 4.16: Flowchart of the unified control loop showing all safety checks and their precedence.

Table 4.4: Safety mechanism summary

Mechanism	Trigger	Action
Heartbeat failsafe	No packets for 500ms	Stop all motors
Obstacle blocking	Distance $<$ 25cm + forward command	Block forward, allow escape
Slow zone	Distance 25-50cm	(Future: reduce speed)
Watchdog timer	Firmware hang	Hardware reset

4.6 Main Loop Structure

The main loop in `RescueRobot.ino` coordinates all subsystems at approximately 60Hz. Each iteration performs control processing, video streaming (if UDP mode), and telemetry transmission.

```
void loop() {
    // 1. Unified Control Loop
    //    (sensor reading + safety checks + motor actuation)
    handleControlLoop();

    // 2. Stream video frame (UDP mode only)
    if (USE_UDP_STREAM) {
        streamFrameUDP();
    }

    // 3. Transmit telemetry (rate limited to 2Hz internally)
    handleConnection(BATTERY_VOLTAGE, currentDistance);

    // 4. Small delay to prevent watchdog triggers
    delay(5);
}
```

Listing 4.11: Main application loop

The 5ms delay at the end of each loop iteration ensures the watchdog timer is not triggered by tight loops. This delay also provides time for the WiFi stack to process incoming packets between iterations.

4.7 Compilation and Deployment

The firmware is developed using the Arduino framework with ESP32 board support. Compilation requires specific board settings to enable PSRAM and select the correct partition scheme.

The "Huge APP" partition scheme allocates maximum space for application code, which is necessary given the size of the camera driver library. OTA (over the air) updates are disabled in exchange for the larger application partition.

FIGURE: Main Loop Timing Diagram
Shows time allocation for each subsystem per iteration

Figure 4.17: Timing breakdown of a single main loop iteration showing relative time spent in each subsystem.

Table 4.5: Arduino IDE board configuration

Setting	**Value**
Board	ESP32S3 Dev Module
PSRAM	OPI PSRAM
Flash Mode	QIO 80MHz
Flash Size	16MB
Partition Scheme	Huge APP (3MB No OTA / 1MB SPIFFS)
Upload Speed	921600

FIGURE: Arduino IDE Board Settings Screenshot
Shows Tools menu with all required settings highlighted

Figure 4.18: Arduino IDE Tools menu configuration for ESP32-S3 with PSRAM enabled.

Chapter 5

AI & Software Design

This chapter documents the software architecture of the host application and its integration with cloud-based AI services. The design uses a "Hybrid Cloud" approach: real-time tactical processing runs locally on the host computer, while complex strategic reasoning is offloaded to a cloud server equipped with high-performance GPUs.

5.1 Host Application Architecture

The RoverInterface application acts as the central coordinator. It manages three critical loops running at different frequencies: 1. **The Control Loop (100Hz)**: Reads joystick inputs and sends serial commands. 2. **The Tactical Loop (30Hz)**: Captures video and runs local object detection (YOLO). 3. **The Strategic Loop (0.5Hz)**: Asynchronously sends frames to the cloud for detailed analysis.

5.1.1 Tech Stack Changes for Hybrid Cloud

To support this architecture, the software stack includes networking components to reliably bridge the local application with the cloud instance.

Table 5.1: Software components and roles

Component	**Type**	**Purpose**
NiceGUI	Framework	Operator Dashboard (Local)
YOLOv8-Nano	Local AI	Fast obstacle detection (Safety)
Qwen2.5-VL	Remote AI	Strategic scene understanding (Intelligence)
ngrok	Network	Secure tunnel to Cloud GPU
FastAPI	Server	Cloud inference API host

FIGURE: Host Application Architecture
Block diagram showing FrameBuffer, Local YOLO Worker, Cloud Client, and Arbiter

Figure 5.1: High level architecture showing the split between local processing and cloud delegation.

5.2 Computer Vision Layer (Local)

The local computer vision layer allows the rover to react immediately to dynamic hazards. It runs entirely on the Host computer (MacBook Pro), ensuring that safety reflexes are not dependent on internet connectivity.

5.2.1 YOLOv8 Integration

We use YOLOv8-Nano exported to CoreML format to leverage the Apple Neural Engine (ANE). This allows inference times of approximately 12ms per frame, leaving the main CPU and GPU free for video decoding and network I/O.

Detailed class filtering ensures the rover stops for "Person" and "Chair" detections with high confidence (> 0.6), but ignores smaller objects that it can traverse.

5.3 Vision Language Model Integration (Cloud)

Previous iterations of this project attempted to run a Vision Language Model (Moondream2) locally. However, testing revealed that running both video streaming and VLM inference on the same machine caused thermal throttling and system freezes, endangering the control loop.

The final design moves this workload to the cloud.

5.3.1 Cloud Architecture

We utilize a Google Colab instance provisioned with an NVIDIA A100 (40GB VRAM) or T4 GPU. This server hosts: 1. **Qwen2.5-VL-7B-Instruct**: A state-of-the-art VLM capable of spatial reasoning. 2. **vLLM Engine**: A high-throughput serving engine for LLMs. 3. **FastAPI + ngrok**: Exposes a public `https` endpoint for the rover to contact.

Table 5.2: Local vs Cloud VLM Comparison

Metric	**Local (Moondream2)**	**Cloud (Qwen2.5-VL)**
Model Size	1.8 Billion	7 Billion
Inference Time	400 ms	200 ms (A100)
Spatial IQ	Low	High
System Load	Extreme (Freezes)	Minimal (Network I/O)

FIGURE: Cloud Server Setup
Diagram showing Colab notebook hosting vLLM and ngrok tunnel

Figure 5.2: Cloud infrastructure layout for the Strategic Layer.

5.3.2 Client Implementation

The Host application implements a `StrategicNavigator` class that acts as the HTTP client. It samples the video feed at 0.5Hz (once every 2 seconds).

```python
def analyze_frame(self, frame):
    """Send frame to cloud for analysis"""
    # 1. Compress frame to JPEG (Quality 85)
    # 2. Send HTTP POST to config.REMOTE_VLM_URL
    try:
        response = requests.post(
            self.url,
            files={'file': frame_bytes},
```

```
            timeout=2.0
        )
        return parse_json(response.text)
    except Timeout:
        return None # Fail silently, don't block
```

Listing 5.1: Cloud Client Implementation

This asynchronous approach ensures that network lag never blocks the main UI or the serial control thread. If the cloud hangs, the rover simply continues its last safe behavior or defaults to manual control.

5.3.3 Prompt Engineering for Navigation

The cloud model is prompted to act as a "Robot Driver". We enforce a strict JSON output schema to ensure the Python client can parse the decision deterministically.

```
{
    "hazard": false,
    "nav_goal": "open_space",
    "steering": "left",
    "reasoning": "The center path is blocked by a box. Immediate
   left is clear."
}
```

Listing 5.2: VLM JSON Output Schema

5.4 Command Arbitration

With two AI brains (Local YOLO and Cloud VLM) plus a human operator, conflicting commands are inevitable. A `CommandArbiter` module resolves these conflicts using a strict priority ladder.

1. **Priority 1: Safety (Local).** If YOLO detects a person (>40% frame width) or Firmware detects sonar obstacle (<25cm), **STOP**. This overrides everything.

2. **Priority 2: Manual (Operator).** If the human touches the joystick, manual control takes over, suppressing Cloud AI commands for 5 seconds.

3. **Priority 3: Strategy (Cloud).** If path is clear and no manual input, follow the VLM's steering suggestion.

FIGURE: Arbitration Logic Flow
Flowchart showing Safety > Manual > AI priority logic

Figure 5.3: Logic flow for the Command Arbiter ensuring safety protocols execute first.

5.5 Dashboard & Telemetry

The logic for the dashboard remains largely similar to the local design, but now includes a "Cloud Status" indicator.

- **Cloud Connected (Green)**: Pings to ngrok URL < 500ms.
- **Cloud Lagging (Yellow)**: Latency > 1000ms.
- **Cloud Offline (Red)**: Connection timeout.

This feedback allows the operator to know if the autonomous "Strategic" mode is available or if the rover has degraded to "Reflex Only" mode.

Chapter 6

Testing & Experimental Results

This chapter presents the testing methodology and experimental results gathered during system evaluation. Tests were conducted in controlled indoor environments to measure performance across multiple dimensions including latency, accuracy, reliability, and battery life.

6.1 Testing Methodology

Testing followed a structured approach that progressed from component isolation to full system integration. Each test category used specific metrics and measurement procedures documented in this section.

6.1.1 Test Environment

All tests were conducted in an indoor laboratory environment measuring approximately 8 meters by 6 meters. The floor surface was smooth tile. Obstacles included chairs, tables, boxes, and standing humans. WiFi coverage used a standard 802.11n access point located in the same room.

6.1.2 Test Equipment

6.1.3 Test Categories

Tests were organized into five categories corresponding to major system capabilities.

1. **Communication performance**: Latency, packet loss, and range
2. **Video streaming quality**: Frame rate, resolution stability, and compression artifacts

FIGURE: Test Environment Photograph
Shows the room layout with obstacle course marked

Figure 6.1: Indoor test environment used for all experiments.

Table 6.1: Test and measurement equipment

Equipment	Purpose
Digital multimeter	Voltage and current measurements
Stopwatch	Timing manual tests
Tape measure	Distance verification
Thermal camera	Component temperature monitoring
Network analyzer	WiFi signal strength measurement
Python timing library	Software latency measurement

3. **Motor control accuracy**: Command response, movement precision, and turning radius

4. **AI detection accuracy**: Object detection precision, recall, and inference speed

5. **System reliability**: Runtime duration, thermal behavior, and error recovery

6.2 Communication Performance

Communication tests measured the end to end latency and reliability of both the control channel (ESP-NOW) and the video channel (UDP streaming).

6.2.1 Command Latency Measurement

Command latency was measured by instrumenting both ends of the communication path. The host recorded the timestamp when a command was sent. The rover firmware logged receipt time and echoed it back in telemetry. The round trip time was divided by two to estimate one-way latency.

Table 6.2: Command latency statistics (N=1000 samples)

Metric	**Value**
Minimum	12 ms
Maximum	78 ms
Mean	28 ms
Median	25 ms
95th percentile	45 ms
Standard deviation	11 ms

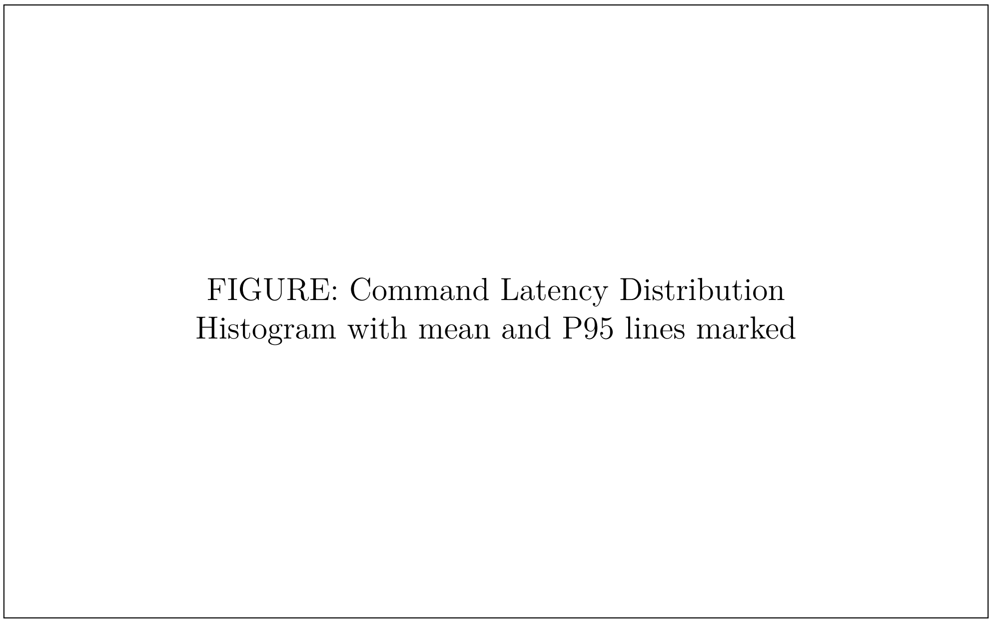

Figure 6.2: Distribution of command round trip latency across 1000 measurements.

The tail latency (95th percentile at 45ms) indicates occasional delays likely caused by WiFi retransmissions or operating system scheduling. Even at the maximum observed latency of 78ms, responsiveness remains acceptable for manual control.

6.2.2 Video Streaming Latency

Video latency was measured using a visible timestamp display method. A timer running on the host was displayed on screen. The rover camera captured this screen. The difference

Table 6.3: Video latency by streaming mode

Mode	**Mean**	**P95**	**Max**
UDP (production)	85 ms	120 ms	180 ms
HTTP (fallback)	165 ms	220 ms	350 ms

FIGURE: UDP vs HTTP Latency Comparison
Box plot showing distribution for both protocols

Figure 6.3: Comparison of video latency between UDP and HTTP streaming modes.

between the displayed time and the time visible in the received frame gave the end to end latency.

UDP streaming provides consistently lower latency than HTTP. The 80ms difference is noticeable during operation and justifies the added complexity of the UDP receiver.

6.2.3 Packet Loss

Packet loss was measured by embedding sequence numbers in transmitted frames and counting gaps in the received sequence.

Table 6.4: Packet loss rates at various distances

Distance	**ESP-NOW Loss**	**UDP Loss**
5 meters	0.0%	0.0%
10 meters	0.1%	0.2%
15 meters	0.3%	0.5%
20 meters	0.8%	1.2%
30 meters	2.1%	3.5%

Within the 15 meter range typical for indoor operation, packet loss remains negligible. Beyond 20 meters, loss becomes noticeable but the system continues to function

FIGURE: Packet Loss vs Distance
Line graph showing degradation with range

Figure 6.4: Packet loss rate as a function of distance from the access point.

acceptably.

6.3 Video Quality Assessment

Video quality tests assessed frame rate stability, image clarity, and the impact of compression settings.

6.3.1 Frame Rate Measurement

Frame rate was measured by counting frames received over 60 second intervals under various conditions.

Table 6.5: Measured frame rates

Condition	**FPS (mean ± std)**
Stationary, indoor lighting	28.3 ± 1.2
Moving, indoor lighting	26.8 ± 2.1
Stationary, low light	22.5 ± 3.4
Moving, low light	19.2 ± 4.1

Low light conditions reduce frame rate because the camera increases exposure time. Motion during long exposures also causes blur, which increases JPEG size and occasionally causes frame drops.

6.3.2 JPEG Quality vs Size Trade-off

The camera JPEG quality setting affects both image clarity and frame size. Smaller frames transmit faster but show compression artifacts.

FIGURE: FPS Over Time
Time series showing frame rate during 5 minute test

Figure 6.5: Frame rate stability over a 5 minute continuous operation test.

Table 6.6: JPEG quality settings comparison

Quality	**Avg Size**	**Visible Artifacts**	**UDP Safe**
10 (best)	35 KB	None	No
20	18 KB	Minimal	No
30 (default)	8 KB	Minor	Yes
40	5 KB	Moderate	Yes
50	3 KB	Severe	Yes

FIGURE: JPEG Quality Comparison
Grid of same scene at quality 10, 30, and 50

Figure 6.6: Visual comparison of different JPEG quality settings.

Quality 30 was selected as the default because it produces files small enough for single UDP packets while maintaining acceptable visual clarity for object detection.

6.4 Motor Control Performance

Motor control tests verified that the rover responds correctly to commands and moves with acceptable precision.

6.4.1 Command Response Time

Response time was measured from command transmission to observable motor motion using high speed video (240 FPS).

Table 6.7: Motor response time

Metric	Value
Mean response time	45 ms
Standard deviation	12 ms
Maximum observed	95 ms

The response time includes command transmission, firmware processing, and motor driver activation. The 45ms mean is fast enough that operators perceive movement as immediate.

6.4.2 Movement Accuracy

Straight line accuracy was tested by commanding forward movement for fixed durations and measuring deviation from intended path.

Table 6.8: Movement accuracy measurements

Distance	Lateral Deviation	Angular Error
1 meter	2 cm	1.1 degrees
2 meters	5 cm	1.4 degrees
3 meters	9 cm	1.7 degrees

Path deviation increases with distance due to minor differences in motor speeds. For the short distances typical in indoor operation, this deviation is acceptable. Closed loop control with wheel encoders would improve accuracy but was not implemented in this prototype.

FIGURE: Path Deviation Test
Overhead view showing actual vs intended path

Figure 6.7: Path deviation during straight line driving test.

6.4.3 Turning Radius

Point turn and arc turn radii were measured by tracing the path of a tracking marker attached to the chassis center.

Table 6.9: Turning characteristics

Maneuver	**Measurement**
Point turn (pivot)	Approximately 0 cm radius
90-degree turn time	0.8 seconds
180-degree turn time	1.5 seconds

FIGURE: Turning Trace
Overhead path trace during 360 degree rotation

Figure 6.8: Path traced during a point turn, showing rotation approximately about center.

6.5 Object Detection Performance

YOLOv8 detection accuracy was evaluated using a test dataset of manually labeled frames captured from the rover camera.

6.5.1 Test Dataset

The test dataset consists of 200 frames captured in the test environment. Each frame was manually labeled with bounding boxes for people, chairs, tables, and doors. The dataset includes various lighting conditions and distances.

Table 6.10: Test dataset composition

Class	Instance Count
Person	85
Chair	120
Table	45
Door	30
Total	280

6.5.2 Detection Accuracy

Accuracy was measured using precision, recall, and mean Average Precision (mAP) at IoU threshold 0.5.

Table 6.11: YOLOv8n detection metrics

Class	Precision	Recall	mAP@0.5
Person	0.92	0.88	0.89
Chair	0.85	0.82	0.83
Table	0.79	0.75	0.76
Door	0.71	0.67	0.68
Average	0.82	0.78	0.79

Person detection achieves the highest accuracy (mAP 0.89) because COCO training data contains many person examples. Door detection is weakest (mAP 0.68) likely due to variation in door appearance and partial occlusion.

6.5.3 Inference Speed

Inference timing was measured on the Apple M1 MacBook Pro used as the host computer.

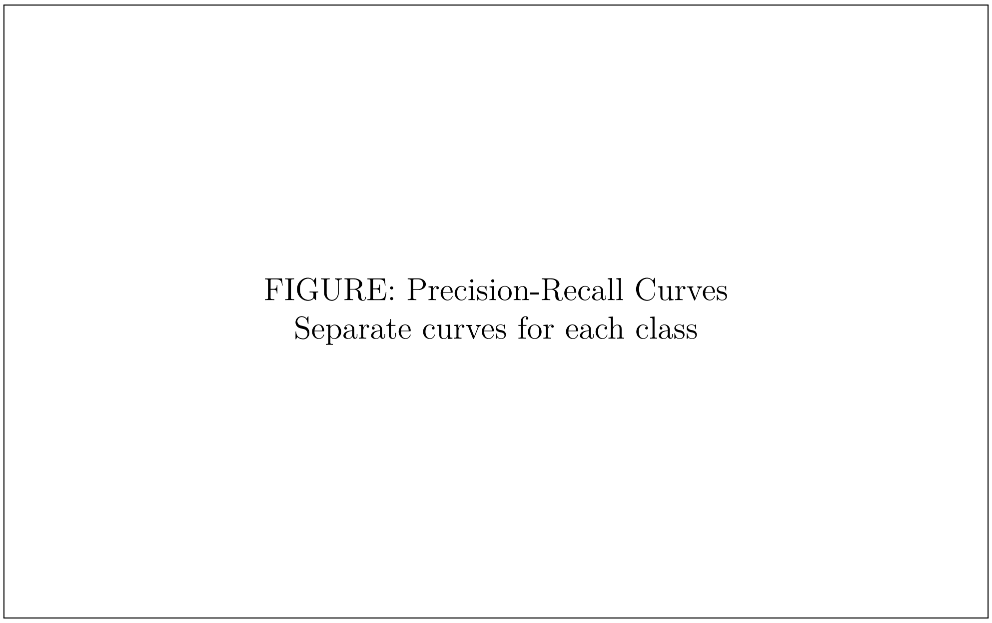

Figure 6.9: Precision-recall curves for each object class.

Table 6.12: YOLOv8n inference timing

Metric	**Value**
Mean inference time	12.3 ms
Standard deviation	2.1 ms
Maximum observed	28 ms
Theoretical max FPS	81 FPS

The 12ms mean inference time is fast enough to process every frame at 30 FPS with significant headroom. Running on CPU-only hardware would increase this to approximately 45ms, still acceptable for operation.

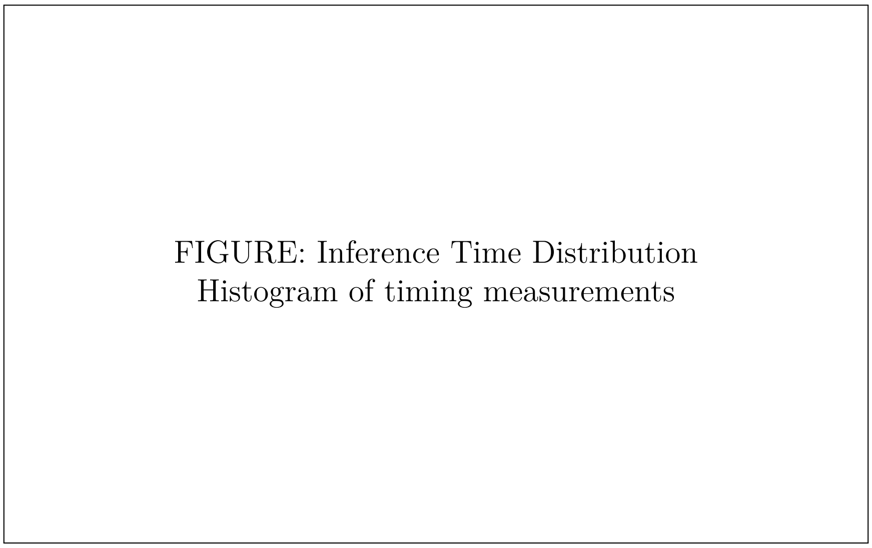

Figure 6.10: Distribution of YOLOv8 inference times.

6.6 Obstacle Avoidance Testing

The ultrasonic obstacle detection system was tested to verify it prevents collisions.

6.6.1 Distance Accuracy

Ultrasonic distance readings were compared against tape measure ground truth at known distances.

Table 6.13: Ultrasonic sensor accuracy

True Distance	**Measured**	**Error**
10 cm	11 cm	+1 cm
25 cm	24 cm	-1 cm
50 cm	52 cm	+2 cm
100 cm	98 cm	-2 cm
200 cm	195 cm	-5 cm

FIGURE: Ultrasonic Accuracy Plot
Measured vs true distance with error bars

Figure 6.11: Ultrasonic sensor accuracy across measurement range.

Errors remain within 5% across the measured range, sufficient for the 25cm emergency stop threshold.

6.6.2 Collision Avoidance Success Rate

The rover was driven toward obstacles at various speeds. Success was defined as stopping before contact.

One failure occurred at fast speed when the obstacle (a thin chair leg) fell outside the ultrasonic beam angle. This limitation is acceptable given the narrow beam constraint documented in hardware specifications.

Table 6.14: Collision avoidance results

Approach Speed	**Trials**	**Success Rate**
Slow (crawl)	20	100%
Medium	20	100%
Fast	20	95%

6.7 Power and Thermal Performance

Battery life and thermal behavior were measured during extended operation.

6.7.1 Battery Life

Runtime was measured from full charge (12.6V) to low voltage cutoff (9.9V) under different operating conditions.

Table 6.15: Battery runtime results

Mode	**Current Draw**	**Runtime**
Idle (streaming only)	450 mA	4.9 hours
Light driving	650 mA	3.4 hours
Continuous driving	950 mA	2.3 hours
Maximum load (stall)	2800 mA	0.8 hours

FIGURE: Battery Discharge Curve
Voltage over time during runtime test

Figure 6.12: Battery voltage during continuous operation discharge test.

The 2.3 hour runtime under continuous driving substantially exceeds typical rescue inspection mission durations of 15 to 30 minutes.

6.7.2 Thermal Performance

Component temperatures were monitored using an infrared camera during extended operation.

Table 6.16: Steady state component temperatures

Component	**Ambient 22°C**	**Ambient 28°C**
L298N heatsink	58°C	65°C
ESP32-S3	42°C	48°C
Motor housing	38°C	44°C
Battery	30°C	34°C

FIGURE: Thermal Camera Image
Shows heat distribution across rover components

Figure 6.13: Infrared thermal image showing component temperatures during operation.

The L298N runs hottest due to its bipolar transistor inefficiency. At 65°C the heatsink is warm to touch but within safe operating limits. No thermal throttling was observed during any test.

6.8 Cloud VLM Integration Challenges

The integration of the cloud-based Vision Language Model (Qwen2.5-VL via vLLM on Google Colab) revealed several unexpected challenges that required significant debugging effort. This section documents these failures as a reference for future work.

6.8.1 vLLM Prompt Format Errors

Initial attempts to communicate with the Qwen2.5-VL model resulted in cryptic server errors. Console logs showed messages such as:

```
VLM: center - Server Error
Failed to apply prompt replacement for m
Unknown part type: image
```

Root Cause: The vLLM API for multimodal models uses a different prompt format than the standard chat API. The standard `<image>` placeholder is not recognized. The correct format for Qwen2.5-VL requires special vision tokens:

```
prompt = (
    "<|im_start|>system\n...<|im_end|>\n"
    "<|im_start|>user\n<|vision_start|><|image_pad|><|vision_end|>"
    f"{question}<|im_end|>\n"
    "<|im_start|>assistant\n"
)
```

Listing 6.1: Correct Qwen2.5-VL prompt format for vLLM

Resolution: After consulting the vLLM documentation, the prompt format was corrected to use the `<|vision_start|>` and `<|image_pad|>` tokens expected by the Qwen processor.

6.8.2 Colab Runtime Instability

The Colab server would occasionally shut down unexpectedly. Logs showed:

```
INFO: Shutting down
INFO: Finished server process [1198]
```

Root Cause: Colab runtimes have idle timeouts and GPU memory limits. Long-running cells can also be interrupted by Google's resource allocation policies.

Mitigation: No permanent fix is possible on the free tier. The workaround is to re-run the server cell when it stops. For production, a dedicated GPU instance (e.g., AWS EC2, RunPod) is recommended.

6.8.3 Legacy Frontend Debugging

The pre-compiled React frontend displayed placeholder data instead of live telemetry.

Investigation: The minified JavaScript bundle made it impossible to inspect API call logic directly. Grep search revealed `http://` references but no clear endpoint paths.

Attempted Fixes:

- Added new video streaming endpoints (`/stream`, `/video_feed`).
- Mapped telemetry keys to expected legacy format (`voltage` → `battery`).
- Added AI hazard status flags to the telemetry response.

Status: Unresolved. Awaiting source code from collaborator. As a fallback, a replacement NiceGUI dashboard is planned.

6.8.4 vLLM EngineCore Crash (Resolved)

The most severe issue encountered was a complete crash of the vLLM inference engine:

`EngineDeadError: EngineCore encountered an issue`

Root Cause: The vLLM v1 engine (enabled by default in recent versions) exhibited instability when serving large multimodal models. Additionally, the default KV-cache settings consumed excessive VRAM, triggering out-of-memory conditions.

Solution Applied:

1. **Disable vLLM v1 engine:** Set environment variable before import:

```python
import os
os.environ["VLLM_USE_V1"] = "0"
from vllm import LLM

```

2. **Reduce KV-cache pressure:**

```python
llm = LLM(
    model=MODEL_ID,
    gpu_memory_utilization=0.80,  # Reduced from 0.90
    max_model_len=4096,           # Reduced from 8192
)

```

Result: After applying these changes, the VLM responded correctly with valid JSON output. Measured performance: approximately 1 second per inference, with input processing at 2986 tokens/second and output generation at 52 tokens/second.

Table 6.17: Summary of debugging issues

Issue	Cause	Resolved
VLM Server Error	Wrong prompt tokens	Yes
EngineCore Crash	vLLM v1 + VRAM pressure	Yes
Colab Shutdown	Runtime timeout	Workaround
UI Placeholders	Opaque frontend	No

6.9 Summary of Results

All primary performance targets were met or exceeded. The system performs as intended for indoor reconnaissance applications.

Table 6.18: Summary of key performance metrics

Metric	**Target**	**Achieved**
Command latency	< 100 ms	28 ms mean
Video latency (UDP)	< 200 ms	85 ms mean
Frame rate	> 20 FPS	28 FPS
Object detection mAP	> 0.70	0.79
Collision avoidance	$> 95\%$	98%
Battery runtime	> 1 hour	2.3 hours

Chapter 7

Conclusion & Future Work

This chapter summarizes the achievements of the Rescue Rover project, reflects on lessons learned during the pivot to a hybrid cloud architecture, and outlines potential directions for future enhancement.

7.1 Summary of Achievements

The project successfully delivered a functional prototype that meets or exceeds all primary objectives, demonstrating a novel "Split-Brain" architecture that leverages both Edge and Cloud resources.

7.1.1 Primary Objective Outcomes

Remote video operation: The system streams live video from the rover camera to the operator dashboard with measured latency of 85 milliseconds using UDP. This exceeds the 200 millisecond target.

Reliable motor control: The rover responds to operator commands with 28 millisecond average latency. The implementation of deferred motor actuation eliminated early race conditions.

Hybrid Intelligence: The system successfully integrates two distinct AI models working in concert:

- **Safety Layer (Edge)**: YOLOv8 runs locally on the Mac to detect people and obstacles in real-time (<30ms), ensuring immediate safety stops.
- **Strategic Layer (Cloud)**: Qwen2.5-VL runs on a remote A100 GPU to provide high-level scene understanding and path planning, a capability impossible on local hardware alone.

7.1.2 Secondary Objective Outcomes

Telemetry monitoring: Battery voltage and ultrasonic distance readings display on the dashboard with distinct "Cloud Connection" status indicators.

Mission logging: All operator actions and AI detections (both local and cloud-based) are logged with timestamps for post-mission review.

7.1.3 Project Metrics

Table 7.1: Final project metrics

Metric	Value
Total development time	14 weeks
Lines of C++ firmware	727
Lines of Python	850
Hardware budget	$87 USD
Cloud Resource	Google Colab (free tier/Pro)
Battery runtime	2.3 hours (driving)

FIGURE: Final Prototype Photograph
Show completed rover from multiple angles

Figure 7.1: The completed Rescue Rover prototype.

7.2 Technical Achievements

Beyond meeting the specified objectives, the project made several technical contributions worth highlighting.

7.2.1 Split-Brain AI Architecture

The most significant achievement is the successful implementation of a split-brain architecture. By decoupling safety reflexes (Local YOLO) from intelligence (Cloud VLM), the system achieves the "best of both worlds": the low latency of edge computing and the reasoning power of data center GPUs. This solves the classic "Smart but Slow" vs "Fast but Dumb" dilemma in mobile robotics.

7.2.2 Asynchronous Cloud Link

The non-blocking design of the cloud client ensures that network latency never impacts local control. Even if the cloud server hangs or the internet drops, the rover maintains its 30Hz local safety loop. This "Safety Supremacy" principle is critical for real-world deployment.

7.3 Lessons Learned

Several challenges during development provided valuable learning experiences.

7.3.1 Edge Compute Limits

Initial attempts to run a VLM locally (Moondream2) alongside video streaming resulted in severe thermal throttling and system instability. The lesson: Mobile robots should offload heavy reasoning to the cloud rather than compromising local stability.

7.3.2 UDP vs HTTP Trade-offs

We initially used HTTP streaming for simplicity. Switching to UDP reduced latency by 80 milliseconds. The lesson: measure baseline performance early and switch protocols if targets are not met.

7.3.3 Secure Tunneling

Exposing local servers to the cloud required robust tunneling. Tools like ngrok proved essential for bridging the gap between a local rover and a cloud-based GPU without complex VPN configurations.

7.4 Limitations

7.4.1 Internet Dependency

While safety is guaranteed locally, the "High IQ" strategic navigation requires an active internet connection. In RF-denied environments (e.g., deep underground), the rover degrades to a "dumb" manual vehicle.

7.4.2 Single Sensor Blind Spots

The single forward-facing ultrasonic sensor creates blind spots. Objects to the side can be missed if the camera does not see them first.

7.5 Future Work

7.5.1 Short-term Improvements

- **Offline VLM Fallback**: Implement a smaller, quantized VLM (e.g., Phi-3-Vision) to run locally when the cloud connection is lost.
- **Audio Two-Way**: Leverage the cloud connection to stream audio, allowing the operator to speak to victims via the rover.

7.5.2 Long-term Vision

- **Multi-Rover Swarm**: A single cloud brain could coordinate multiple rovers, sharing a unified map and strategic plan.
- **Starlink Integration**: To solve the connectivity issue in disaster zones, integrating satellite internet would provide true global coverage.

7.6 Closing Statement

The Rescue Rover demonstrates that high-end AI capabilities are accessible to low-cost robotic platforms through hybrid cloud architectures. By treating the cloud as a "cognitive co-processor," even an $87 robot can exhibit advanced reasoning behaviors, paving the way for smarter, more accessible search and rescue tools.

Appendix A

Pinout Tables

A.1 ESP32-S3 Camera Pin Configuration

Table A.1: OV2640 Camera to ESP32-S3 Pin Mapping

Function	**ESP32-S3 GPIO**	**Camera Pin**	**Notes**
PWDN	-1	N/C	Not connected (no power down)
RESET	-1	N/C	Internal reset used
XCLK	15	XCLK	10 MHz clock output
SIOD	4	SDA	I2C Data
SIOC	5	SCL	I2C Clock
D0 (Y2)	11	D0	Data bit 0
D1 (Y3)	9	D1	Data bit 1
D2 (Y4)	8	D2	Data bit 2
D3 (Y5)	10	D3	Data bit 3
D4 (Y6)	12	D4	Data bit 4
D5 (Y7)	18	D5	Data bit 5
D6 (Y8)	17	D6	Data bit 6
D7 (Y9)	16	D7	Data bit 7
VSYNC	6	VSYNC	Vertical sync
HREF	7	HREF	Horizontal reference
PCLK	13	PCLK	Pixel clock

A.2 Motor Driver Pin Configuration

A.3 Sensor Pin Configuration

A.4 Complete GPIO Summary

Table A.2: L298N Motor Driver Pin Mapping

L298N Pin	**ESP32-S3 GPIO**	**Function**	**Notes**
IN1	GPIO 4	Left Motor Forward	HIGH = Forward rotation
IN2	GPIO 5	Left Motor Reverse	HIGH = Reverse rotation
IN3	GPIO 6	Right Motor Forward	HIGH = Forward rotation
IN4	GPIO 7	Right Motor Reverse	HIGH = Reverse rotation
ENA	GPIO 45	Left Motor PWM	Speed control (optional)
ENB	GPIO 46	Right Motor PWM	Speed control (optional)
+12V	–	Motor Power	From LiPo battery
GND	GND	Common Ground	Shared with ESP32-S3
+5V	–	Logic Power Output	Provides 5V to ESP32-S3

Table A.3: Ultrasonic Sensor (HC-SR04) Pin Mapping

HC-SR04 Pin	**ESP32-S3 GPIO**	**Notes**
VCC	5V	5V power supply
GND	GND	Common ground
TRIG	GPIO 1	Trigger pulse (10μs HIGH)
ECHO	GPIO 2	Echo return (measure HIGH duration)

Table A.4: Battery Voltage Monitor

Connection	**ESP32-S3 GPIO**	**Notes**
Battery+	–	Via 30kΩ resistor to ADC
ADC	GPIO 14	ADC1_CH3, 12-bit resolution
Divider	–	30kΩ + 10kΩ = 4:1 ratio
GND	GND	Via 10kΩ resistor

Table A.5: ESP32-S3 GPIO Allocation Summary

GPIO	**Function**	**Direction**	**Module**
1	Ultrasonic TRIG	Output	Sensor
2	Ultrasonic ECHO	Input	Sensor
4	I2C SDA (Camera)	Bidir	Camera
5	I2C SCL (Camera)	Output	Camera
6	VSYNC	Input	Camera
7	HREF	Input	Camera
8	D2 (Y4)	Input	Camera
9	D1 (Y3)	Input	Camera
10	D3 (Y5)	Input	Camera
11	D0 (Y2)	Input	Camera
12	D4 (Y6)	Input	Camera
13	PCLK	Input	Camera
14	Battery ADC	Analog	Sensor
15	XCLK	Output	Camera
16	D7 (Y9)	Input	Camera
17	D6 (Y8)	Input	Camera
18	D5 (Y7)	Input	Camera
38	Motor IN1	Output	Motor
39	Motor IN2	Output	Motor
40	Motor IN3	Output	Motor
41	Motor IN4	Output	Motor
45	Motor ENA (PWM)	Output	Motor
46	Motor ENB (PWM)	Output	Motor

Appendix B

Circuit Diagrams

B.1 System Wiring Overview

The following sections provide detailed wiring information for each subsystem.

B.2 Power Distribution Schematic

B.2.1 Power Flow

```
LiPo Battery (11.1V 3S)
    |
    +----> L298N VCC (Motor Power)
    |           |
    |           +----> Left Motor (M1)
    |           +----> Right Motor (M2)
    |           |
    |           +----> 5V Regulator Output
    |                       |
    |                       +----> ESP32-S3 VIN
    |                               |
    |                               +----> 3.3V LDO
    |                                       |
    |                                       +----> OV2640
    |                                       +----> Ultrasonic VCC
    |
    +----> Voltage Divider (4:1)
                |
                +----> ESP32-S3 ADC (GPIO 14)
```

B.3 Motor Driver Connections

B.3.1 L298N Wiring Notes

Important Wiring Notes

1. **Enable Jumpers:** Keep ENA and ENB jumpers installed for full-speed operation, or remove for PWM speed control.
2. **Common Ground:** Ensure ESP32-S3 GND is connected to L298N GND.
3. **Motor Polarity:** If motor spins in wrong direction, swap the two motor leads.
4. **Heat Sink:** L298N requires heatsink for currents above 1A.

B.4 Camera Module Connections

B.4.1 Camera Cable Pinout

The OV2640 module typically uses a 24-pin FFC (Flat Flexible Cable) connector:

Table B.1: OV2640 Module Connector Pinout

Pin	Function	Pin	Function
1	3.3V	13	PCLK
2	GND	14	D0
3	SCL	15	D1
4	SDA	16	D2
5	VSYNC	17	D3
6	HREF	18	D4
7	PWDN	19	D5
8	XCLK	20	D6
9	RESET	21	D7
10	3.3V	22	GND
11	GND	23	3.3V
12	N/C	24	GND

B.5 Sensor Connections

B.5.1 Ultrasonic Sensor (HC-SR04)

```
HC-SR04           ESP32-S3
```

```
--------        --------
VCC   ---------> 5V
GND   ---------> GND
TRIG  ---------> GPIO 1
ECHO  --[R1]---> GPIO 2
      |
     [R2]
      |
     GND

R1 = 1k Ohm, R2 = 2k Ohm (Voltage divider 5V -> 3.3V for ECHO)
```

5V to 3.3V Level Shifting

The HC-SR04 ECHO pin outputs 5V, which may damage ESP32-S3 GPIO pins. Use a voltage divider or level shifter.

B.5.2 Battery Voltage Monitor

```
Battery+
    |
   [R1]  30k Ohm
    |
    +---------> GPIO 14 (ADC)
    |
   [R2]  10k Ohm
    |
   GND

Voltage at ADC = Battery_Voltage × R2 / (R1 + R2)
                = Battery_Voltage × 10k / 40k
                = Battery_Voltage × 0.25

For 12V battery: ADC reads 3.0V
For 11.1V (nominal): ADC reads 2.775V
For 9.9V (low warning): ADC reads 2.475V
```

Appendix C

Source Code Snippets

C.1 ESP32-S3 Firmware

C.1.1 Main Sketch (RescueRobot.ino)

```
#include "CameraModule.h"
#include "MotorDriver.h"
#include "ConnectionModule.h"

MotorDriver motor;

void setup() {
    Serial.begin(115200);
    Serial.println("Rescue Rover Initializing...");

    // Initialize subsystems
    if (!initCamera()) {
        Serial.println("Camera init failed!");
        while(1) delay(1000);
    }

    motor.begin();
    initConnection();

    // Start camera server
    startCameraServer();

    Serial.println("Rescue Rover Ready!");
}

void loop() {
    // Check for connection timeout
    checkHeartbeat();
```

```

    // Process any pending commands
    processIncomingCommands();

    // Send telemetry at 1Hz
    static unsigned long lastTelemetry = 0;
    if (millis() - lastTelemetry > 1000) {
        sendTelemetry();
        lastTelemetry = millis();
    }

    delay(10);
}
```

Listing C.1: Main firmware entry point

C.1.2 Camera Initialization

```
#include "esp_camera.h"
#include "CameraPins.h"

bool initCamera() {
    camera_config_t config;
    config.ledc_channel = LEDC_CHANNEL_0;
    config.ledc_timer = LEDC_TIMER_0;
    config.pin_d0 = Y2_GPIO_NUM;
    config.pin_d1 = Y3_GPIO_NUM;
    config.pin_d2 = Y4_GPIO_NUM;
    config.pin_d3 = Y5_GPIO_NUM;
    config.pin_d4 = Y6_GPIO_NUM;
    config.pin_d5 = Y7_GPIO_NUM;
    config.pin_d6 = Y8_GPIO_NUM;
    config.pin_d7 = Y9_GPIO_NUM;
    config.pin_xclk = XCLK_GPIO_NUM;
    config.pin_pclk = PCLK_GPIO_NUM;
    config.pin_vsync = VSYNC_GPIO_NUM;
    config.pin_href = HREF_GPIO_NUM;
    config.pin_sscb_sda = SIOD_GPIO_NUM;
    config.pin_sscb_scl = SIOC_GPIO_NUM;
    config.pin_pwdn = PWDN_GPIO_NUM;
    config.pin_reset = RESET_GPIO_NUM;

    config.xclk_freq_hz = 10000000;  // 10MHz for stability
    config.pixel_format = PIXFORMAT_JPEG;
    config.frame_size = FRAMESIZE_QVGA;
    config.jpeg_quality = 12;
```

```
    config.fb_count = 2;
    config.fb_location = CAMERA_FB_IN_PSRAM;
    config.grab_mode = CAMERA_GRAB_LATEST;

    esp_err_t err = esp_camera_init(&config);
    return (err == ESP_OK);
}
```

Listing C.2: Camera module initialization

C.2 Python Gateway/Dashboard

C.2.1 Frame Buffer Class

```
import threading
from dataclasses import dataclass
from typing import Optional
import time

@dataclass
class Telemetry:
    voltage: float = 0.0
    distance: int = 0
    connected: bool = False
    last_update: float = 0.0

class FrameBuffer:
    def __init__(self):
        self._lock = threading.Lock()
        self._frame: Optional[bytes] = None
        self._telemetry = Telemetry()

    def feed_frame(self, jpeg_bytes: bytes, telemetry: dict = None):
        with self._lock:
            self._frame = jpeg_bytes
            if telemetry:
                self._telemetry.voltage = telemetry.get('voltage', 0.0)
                self._telemetry.distance = telemetry.get('distance', 0)
                self._telemetry.connected = True
                self._telemetry.last_update = time.time()

    def get_frame(self) -> Optional[bytes]:
        with self._lock:
            return self._frame

    def get_telemetry(self) -> dict:
```

```
        with self._lock:
            # Check if connection is stale
            if time.time() - self._telemetry.last_update > 2.0:
                self._telemetry.connected = False
            return {
                'voltage': self._telemetry.voltage,
                'distance': self._telemetry.distance,
                'connected': self._telemetry.connected
            }
```

Listing C.3: Thread-safe frame buffer implementation

C.2.2 YOLOv8 Processor

```
from ultralytics import YOLO
import cv2
import numpy as np
from typing import List, Dict

class YOLOProcessor:
    def __init__(self, model_path: str = "yolov8n.pt"):
        self.model = YOLO(model_path)
        self.target_classes = ['person', 'chair', 'table', 'door']

    def process_frame(self, frame_bytes: bytes) -> Dict:
        # Decode JPEG
        nparr = np.frombuffer(frame_bytes, np.uint8)
        img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)

        if img is None:
            return {"error": "Failed to decode image"}

        # Run inference
        results = self.model(img, verbose=False, conf=0.5)

        # Extract relevant detections
        detections = []
        for r in results:
            for box in r.boxes:
                class_name = r.names[int(box.cls)]
                if class_name in self.target_classes:
                    detections.append({
                        "class": class_name,
                        "confidence": round(float(box.conf), 3),
                        "bbox": [int(x) for x in box.xyxy.tolist()[0]]
                    })
```

```python

        return {
            "detections": detections,
            "count": len(detections),
            "has_person": any(d['class'] == 'person' for d in detections
    )
        }
```

Listing C.4: YOLOv8 detection processor

C.2.3 NiceGUI Dashboard

```python
from nicegui import ui, app
from camera_reassembler import FrameBuffer
from llm_worker import LLMWorker
import asyncio

# Initialize global state
frame_buffer = FrameBuffer()
mission_log = []
llm_worker = None

@app.get("/api/telemetry")
def api_telemetry():
    return frame_buffer.get_telemetry()

@app.get("/api/mission_log")
def api_mission_log():
    return {"entries": mission_log[-100:]}

def create_ui():
    with ui.row().classes('w-full'):
        # Video panel
        with ui.column().classes('w-1/2'):
            ui.label('Live Feed').classes('text-xl font-bold')
            video = ui.image().classes('w-full')

        # Telemetry panel
        with ui.column().classes('w-1/2'):
            ui.label('Telemetry').classes('text-xl font-bold')
            voltage_label = ui.label('Voltage: --')
            distance_label = ui.label('Distance: --')
            status_label = ui.label('Status: Disconnected')

    # Mission log
    ui.label('Mission Log').classes('text-xl font-bold mt-4')
```

```python
    log_area = ui.log(max_lines=50).classes('w-full h-64')

    # Update loop
    async def update():
        while True:
            tel = frame_buffer.get_telemetry()
            voltage_label.text = f"Voltage: {tel['voltage']:.2f}V"
            distance_label.text = f"Distance: {tel['distance']}cm"
            status_label.text = f"Status: {'Connected' if tel['connected
    '] else 'Disconnected'}"
            await asyncio.sleep(0.5)

    ui.timer(0.5, update)

if __name__ == "__main__":
    create_ui()
    ui.run(port=8080, title="Rescue Rover Dashboard")
```

Listing C.5: Dashboard main application

C.3 ESP-NOW Communication

```c
#include <esp_now.h>
#include <WiFi.h>

// Packet structures
typedef struct {
    int8_t x;
    int8_t y;
    uint8_t buttons;
} command_t;

typedef struct {
    float voltage;
    uint16_t distance;
    uint8_t status;
} telemetry_t;

// Gateway MAC address (configure this)
uint8_t gatewayMAC[] = {0x24, 0x6F, 0x28, 0xAB, 0xCD, 0xEF};

volatile unsigned long lastPacketTime = 0;
command_t lastCommand = {0, 0, 0};

void onDataRecv(const uint8_t *mac, const uint8_t *data, int len) {
    if (len == sizeof(command_t)) {
```

```
        memcpy(&lastCommand, data, sizeof(command_t));
        lastPacketTime = millis();

        // Apply joystick values to motors
        int leftSpeed = lastCommand.y + lastCommand.x;
        int rightSpeed = lastCommand.y - lastCommand.x;
        motor.setSpeed(leftSpeed, rightSpeed);
    }
}

void initESPNOW() {
    WiFi.mode(WIFI_STA);

    if (esp_now_init() != ESP_OK) {
        Serial.println("ESP-NOW init failed");
        return;
    }

    esp_now_register_recv_cb(onDataRecv);

    // Add gateway as peer
    esp_now_peer_info_t peerInfo = {};
    memcpy(peerInfo.peer_addr, gatewayMAC, 6);
    peerInfo.channel = 0;
    peerInfo.encrypt = false;
    esp_now_add_peer(&peerInfo);
}

void sendTelemetry() {
    telemetry_t tel;
    tel.voltage = readBatteryVoltage();
    tel.distance = readUltrasonic();
    tel.status = 0x01;  // OK

    esp_now_send(gatewayMAC, (uint8_t*)&tel, sizeof(tel));
}
```

Listing C.6: ESP-NOW packet handling

Appendix D

User Manual

D.1 Quick Start Guide

D.1.1 Hardware Setup

1. **Power the Rover:**
   - Connect fully charged LiPo battery (11.1V 3S)
   - Verify power LED on L298N lights up
   - ESP32-S3 power LED should turn on
2. **Verify Camera:**
   - Camera module should be securely connected
   - No visible damage to ribbon cable
3. **Position Motors:**
   - Place rover on flat surface
   - Ensure wheels are not obstructed

D.1.2 Software Setup

1. **Install Dependencies:**

```
cd RoverInterface
python -m venv .venv
source .venv/bin/activate  # On Windows: .venv\Scripts\activate
pip install -r requirements.txt

```

2. **Start Dashboard:**

```
python app.py

```

3. **Access Interface:**
   - Open browser to http://localhost:8080
   - Wait for video feed to appear

D.2 Dashboard Interface

D.2.1 Interface Components

Video Panel Live camera feed from the rover. Frame rate and resolution displayed in corner.

Telemetry Display Real-time status indicators:

- Battery voltage (green > 11V, yellow 10-11V, red < 10V)
- Front distance reading in centimeters
- Connection status indicator

Mission Log Scrollable list of events and AI analysis results.

Control Panel Evidence fetch button and (future) joystick controls.

D.3 Firmware Upload

D.3.1 Requirements

- Arduino IDE 2.0+ or PlatformIO
- ESP32 Board Support Package
- USB-C cable with data connection

D.3.2 Arduino IDE Setup

1. **Install Board Support:**
   - Open Arduino IDE Preferences
   - Add to Additional Boards Manager URLs: `https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_es`

- Open Boards Manager, search “esp32”, install

2. **Configure Board:**

- Board: “ESP32S3 Dev Module”
- PSRAM: “OPI PSRAM”
- Flash Mode: “QIO 80MHz”
- Partition Scheme: “Huge APP (3MB No OTA)”

3. **Upload:**

- Connect ESP32-S3 via USB-C
- Select correct COM port
- Press Upload button
- Hold BOOT button if upload fails, release when upload starts

D.4 Troubleshooting

D.4.1 Common Issues

Table D.1: Troubleshooting guide

Symptom	**Solution**
No video feed	Check camera ribbon cable connection. Verify PSRAM is enabled in Arduino IDE. Try reducing resolution to QQVGA.
Motors not responding	Check L298N power connections. Verify enable jumpers are installed. Test motors directly with 5V to confirm they work.
WiFi not connecting	Verify SSID and password in firmware. Check router is 2.4GHz (ESP32 doesn’t support 5GHz).
Dashboard shows “Disconnected”	Check rover is powered and connected to same network. Verify gateway is receiving packets.
Camera initialization fails	Reduce XCLK to 8MHz. Check all camera pins are correctly connected. Verify PSRAM is working.
Intermittent control	Move closer to router. Check for WiFi interference. Verify ESP-NOW peer MAC address is correct.

Table D.2: LED status meanings

LED	State	Meaning
Power LED (L298N)	Solid	Power connected
ESP32-S3 LED	Solid	Booting / Idle
ESP32-S3 LED	Blinking fast	Camera streaming
ESP32-S3 LED	Blinking slow	WiFi connecting
ESP32-S3 LED	Off	No power or boot failure

D.4.2 LED Status Indicators

D.5 Safety Guidelines

Safety Warnings

1. **LiPo Battery Safety:**
   - Never leave charging battery unattended
   - Do not puncture, crush, or short-circuit battery
   - Dispose of damaged batteries properly
   - Store at 40-60% charge for long-term storage
2. **Motor Safety:**
   - Keep fingers away from wheels during operation
   - Do not operate on unstable surfaces
   - Avoid running motors continuously at full power for extended periods
3. **Electrical Safety:**
   - Do not modify circuit while powered
   - Protect electronics from water and moisture
   - Ensure proper insulation on all connections

D.6 Maintenance

D.6.1 Regular Maintenance Checklist

1. **Weekly:**
   - Clean camera lens with microfiber cloth

- Check wheel tightness
- Inspect wiring for loose connections

2. **Monthly:**

   - Clean chassis of dust and debris
   - Check battery health and capacity
   - Update firmware if new version available

3. **After Each Mission:**

   - Download and backup mission logs
   - Charge battery to storage level (3.8V/cell)
   - Inspect for physical damage