The Smart Car Project

An Industrial-Grade Autonomous Robotics Platform

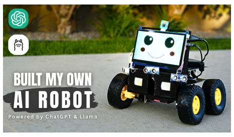
A Complete Technical Reference
From Electrons to Execution

Project Lead & Architect
Bura Himasree

Hardware & Systems Engineer
Kamal Bura

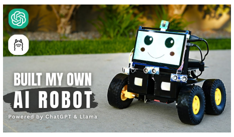
Engineering Team
Team Member 1
Team Member 2
Team Member 3
Team Member 4
Team Member 5
Team Member 6
Team Member 7
Team Member 8
Team Member 9
Team Member 10

Version 1.0
January 2026



Certificate

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
Government Polytechnic Parkal, Warangal Dist



CERTIFICATE

This is to certify that the project report entitled **"THE SMART CAR PROJECT: AN INDUSTRIAL-GRADE AUTONOMOUS ROBOTICS PLATFORM"** is a bonafide record of the work carried out by **Bura Himasree** under my supervision and guidance, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Computer Science & Engineering.

Project Supervisor
TBD Supervisor
TBD Designation
Department of CSE

Head of the Department
TBD HOD
Professor & Head
Department of CSE

External Examiner

Date: January 26, 2026

Place: Warangal Dist

Declaration

We, **Bura Himasree** and **Kamal Bura**, hereby declare that the work presented in this project report entitled **“THE SMART CAR PROJECT: AN INDUSTRIAL-GRADE AUTONOMOUS ROBOTICS PLATFORM”** is an original record of work carried out by us under the guidance of **TBD Supervisor**, Department of Computer Science & Engineering, **Government Polytechnic Parkal, Warangal Dist**.

We further declare that the results of this project work have not been submitted to any other university or institute for the award of any degree or diploma.

Bura Himasree
TBD Roll Number

Kamal Bura
TBD Roll Number

Date: January 26, 2026
Place: Warangal Dist

Acknowledgement

The successful completion of any project is the outcome of the coordination, guidance, and assistance of many individuals. We would like to express our deepest gratitude to all those who made this project possible.

First and foremost, we express our profound gratitude to our project guide, **TBD Supervisor**, for their constant encouragement, invaluable guidance, and scholarly advice throughout the duration of this project. Their insights were instrumental in overcoming the technical challenges of integrating AI onto embedded hardware.

We are extremely thankful to **TBD HOD**, Head of the Department, Computer Science & Engineering, for providing the necessary facilities and moral support required for the project.

We also extend our thanks to the technical staff of the robotics lab and our fellow students for their assistance and for providing a stimulating research environment. A special thanks to the following students who contributed to various testing phases:

Contributor 1, Contributor 2, Contributor 3, Contributor 4, Contributor 5, Contributor 6,
Contributor 7, Contributor 8, Contributor 9, Contributor 10

Finally, we would like to thank our families and friends for their unwavering support and patience during the long hours spent in the hardware lab.

Bura Himasree
Kamal Bura

Dedicated to the engineers who believe that
embedded systems can be both intelligent and elegant.

Abstract

This document presents the complete technical architecture of an autonomous voice-controlled robotic platform built on commodity hardware. Unlike academic prototypes, this system was engineered for real-world deployment, emphasizing reliability, modularity, and measurable performance.

The Challenge

Building a robot that can simultaneously listen, see, think, and act requires solving difficult resource contention problems. Most hobby-grade implementations use a single-threaded "while True" loop that freezes the moment any subsystem demands computation.

Our Approach

We implemented a **Service-Oriented Architecture (SOA)** on a Raspberry Pi 4, using ZeroMQ for inter-process communication. Each sensory modality (audio, vision, sensors) runs as an independent service, publishing events to a central orchestrator.
Key innovations include:

- **Phase-Driven State Machine**: A 17-transition FSM that prevents illegal state combinations.
- **Dual-Brain Architecture**: High-level cognition on Pi, low-level reflexes on ESP32.
- **Hardware Interlocks**: The ESP32 can override Pi commands to prevent collisions.
- **Latest-Frame Grabber**: A threading pattern that eliminates camera buffer lag.

Result

The system achieves a **2.8-second end-to-end latency** from voice command to wheel motion, operating reliably at 2.0 GHz with thermal stability below 60°C.

Keywords: Embedded AI, ZeroMQ, Raspberry Pi, ESP32, Voice Assistant, Computer Vision, YOLO

Contents

List of Figures

1

System Overview

1.1 Introduction and Motivation

The pursuit of autonomous robotics has traditionally been constrained by a fundamental dichotomy: systems are either sophisticated but expensive, or affordable but limited. Industrial robots from established manufacturers offer reliable autonomy but require capital investments exceeding tens of thousands of dollars. Conversely, educational robotics kits provide accessibility but deliver only rudimentary capabilities—line following, obstacle avoidance via simple reflexes, or remote control operation.

This project addresses a specific gap in this landscape: the development of an **autonomous, voice-controlled robotic platform** using commodity hardware. The target platform is the Raspberry Pi 4 Model B paired with an ESP32 microcontroller.

The motivation extends beyond mere cost reduction. Modern artificial intelligence has reached a maturation point where large language models can interpret natural language commands and computer vision models can identify objects with low latency—both capabilities that were computationally infeasible on embedded hardware merely five years ago. The confluence of affordable single-board computers, quantized AI models, and cloud API accessibility creates an unprecedented opportunity for sophisticated autonomous systems at educational price points.

However, hardware capability alone does not guarantee system reliability. The critical challenge lies in *architectural design*: how independent subsystems—audio capture, speech recognition, language understanding, visual perception, motor control, and user feedback—can operate concurrently without resource contention, cascading failures, or unpredictable behavior. This document presents a complete solution to that challenge.

1.2 Problem Statement

Embedded robotics implementations commonly suffer from architectural deficiencies that manifest as operational failures under real-world conditions. Analysis of typical hobbyist and educational robotics projects reveals three categories of systemic problems.

1.2.1 Blocking Operation Failures

Traditional single-threaded robot control follows a sequential pattern: read sensors, compute decision, actuate motors, repeat. This approach fails catastrophically when any operation requires significant time. Audio capture for voice commands blocks the control loop, causing the robot to drive blindly during speech recognition. Camera frame processing introduces latency, resulting in motor commands based on stale visual data. Network requests to cloud APIs freeze the entire system for seconds at a time.

The consequence is a robot that cannot safely perform concurrent activities. It cannot listen while moving. It cannot observe while speaking. It cannot think while sensing. Each capability excludes the others.

1.2.2 Cascading Failure Propagation

Monolithic architectures create tight coupling between unrelated subsystems. A malformed response from a cloud API crashes the JSON parser, which terminates the main process, which halts motor control, which leaves the robot stranded mid-motion. A corrupted camera frame triggers an exception in the vision module, which propagates upward until the entire application terminates. Memory exhaustion in the speech recognition engine consumes resources needed by other subsystems, degrading system-wide performance before eventual failure.

These failure modes share a common characteristic: localized faults produce global consequences. The system lacks isolation boundaries that would contain failures within their originating subsystems.

1.2.3 Resource Contention

Embedded Linux systems present hardware resources as exclusive devices. The ALSA audio subsystem permits only one process to hold the microphone at a time. A wake-word detector listening for activation phrases and a speech-to-text engine transcribing commands cannot both access the microphone simultaneously—yet both require audio input during the voice interaction sequence.

Similarly, GPIO pins, serial ports, and display framebuffers are contested resources. Without explicit coordination mechanisms, multiple processes attempting concurrent access produce undefined behavior, deadlocks, or hardware damage in extreme cases.

1.2.4 The Core Problem

The fundamental problem addressed by this project is the design and implementation of a **multimodal autonomous robot** that reliably performs simultaneous perception, cognition, and action on resource-constrained embedded hardware. The solution must:

1. Enable concurrent operation of audio, vision, and motor subsystems
2. Isolate failures to prevent single-point system collapse
3. Coordinate shared resource access without deadlocks
4. Maintain safety guarantees independent of software state
5. Operate within the thermal and power constraints of portable deployment

1.3 Design Goals and Constraints

The system design was governed by explicit goals derived from the problem statement, balanced against practical constraints imposed by the deployment environment.

1.3.1 Primary Design Goals

Goal 1: Concurrent Multi-Modal Operation. The robot must simultaneously listen for voice commands, observe its environment through computer vision, and execute physical movement. No capability should block or disable another during normal operation.

Goal 2: Graceful Degradation. Failure of any single subsystem must not terminate the entire robot. If the vision service crashes, the robot should continue responding to voice commands. If the cloud API becomes unreachable, basic local functionality should persist. The system should never enter an unrecoverable state requiring physical intervention.

Goal 3: Observable State. The robot's internal state must be externally visible through multiple modalities: LED color patterns indicating operational phase, display graphics showing current activity, and structured logging for post-hoc analysis. Operators should never need to guess what the robot is doing or why.

Goal 4: Hardware-Enforced Safety. Physical safety guarantees—particularly collision avoidance—must not depend on software correctness. Even if the main operating system kernel panics, the robot must not drive into obstacles. This requires safety logic to execute on hardware independent of the primary compute platform.

Goal 5: Maintainability. The codebase must support modification and extension by developers unfamiliar with the original implementation. Clear module boundaries, consistent naming conventions, comprehensive configuration externalization, and explicit state machines contribute to this goal.

1.3.2 Deployment Constraints

Constraint 1: Thermal Budget. The Raspberry Pi 4 throttles CPU frequency when die temperature exceeds 80°C. Continuous high-load operation in ambient temperatures up to 40°C requires computational workloads to remain sustainable. Vision inference, audio processing, and background services must collectively maintain processor temperature below 60°C for reliable operation.

Constraint 2: Power Budget. Portable operation requires battery power. The 5V rail supplying the Raspberry Pi, camera, microphone, and display must not experience voltage sag during motor current spikes. Motor and logic power domains require isolation.

Constraint 3: Memory Budget. The Raspberry Pi 4 provides 8 GB RAM in the deployed configuration. Multiple concurrent processes—each loading AI models, maintaining network connections, and buffering sensor data—must operate within this envelope without triggering Linux OOM killer intervention.

Constraint 4: Network Availability. Cloud API access for large language model inference assumes WiFi connectivity. The system must handle network unavailability gracefully, maintaining local functionality during connectivity gaps.

Constraint 5: Component Availability. All hardware components must be commercially available through standard electronics distributors. The design avoids obsolete parts, custom fabrication, or components with extended lead times.

1.4 System Capabilities

The following capabilities are supported by the deployed codebase and runtime audits where evidence exists.

1.4.1 Voice Interaction

The system responds to a custom wake word (“Hey Veera”) detected using the Porcupine wake word engine. Upon wake word detection, the system transitions to an active listening state, indicated by LED state changes. Spoken commands are transcribed using Faster-Whisper running locally on the Raspberry Pi with int8 quantization.

Transcribed text is submitted to a language model for interpretation (cloud or local depending on configuration). The language model returns structured responses specifying both verbal replies and physical actions. Text-to-speech synthesis using Piper converts verbal responses to audio output.

The complete voice interaction cycle—from wake word through spoken response—varies with configuration and runtime conditions.

1.4.2 Visual Perception

The system performs continuous object detection using YOLOv11 Nano exported to ONNX format. The vision pipeline processes camera frames and publishes detected objects with bounding boxes, class labels, and confidence scores.

Detection results are available to other subsystems for decision-making. The language model receives vision context enabling commands such as “find the bottle” or “follow the person.”

Frame processing operates at approximately 3-4 frames per second on CPU, sufficient for indoor navigation at walking speeds.

1.4.3 Autonomous Navigation

Motor control commands issued by the main processor are executed by the ESP32 microcontroller via UART communication. The ESP32 drives DC motors through an L298N H-bridge motor driver, supporting forward, backward, left turn, right turn, and stop commands.

Navigation commands originate from language model responses interpreting user intent. The system can execute compound maneuvers such as “turn around” or “back up slowly.”

1.4.4 Collision Avoidance

Three HC-SR04 ultrasonic sensors mounted on the robot chassis measure distances to obstacles in the forward arc. The ESP32 firmware continuously monitors these sensors at 20 Hz (50 ms cycle time).

When any sensor detects an obstacle within 10 cm, the ESP32 immediately disables motor output regardless of commands from the Raspberry Pi. This hardware interlock prevents collisions even if the main processor software has crashed, entered an infinite loop, or issued erroneous commands.

The collision avoidance system operates independently and cannot be overridden by software.

1.4.5 Environmental Sensing

An MQ-3 gas sensor connected to the ESP32’s analog-to-digital converter monitors air quality. Sensor readings are transmitted to the Raspberry Pi as part of the telemetry stream and are available for language model context.

1.4.6 User Feedback

System state is communicated through three feedback channels:

- **LED Ring**: An 8-LED NeoPixel ring displays color-coded patterns indicating operational phase (idle, listening, thinking, speaking, error)
- **TFT Display**: A 3.5-inch SPI display renders facial expressions and status graphics

- **Audio Output**: Synthesized speech and notification sounds provide auditory feedback

These feedback mechanisms are intended to provide rapid system state visibility.

1.5 Deployment Context

1.5.1 Target Hardware Platform

The system deploys on a Raspberry Pi 4 Model B with 8 GB RAM, operating Debian-based Linux with kernel support for required peripherals. The processor is overclocked to 2.0 GHz with active cooling to maintain thermal stability under continuous AI workloads.

An ESP32 DevKit microcontroller serves as the motor control subsystem, connected to the Raspberry Pi via UART at 115200 baud. The ESP32 executes time-critical sensor polling and motor control independently of the Linux scheduler.

1.5.2 Software Environment

The system runs multiple independent services managed by systemd, enabling automatic startup, crash recovery, and orderly shutdown. Each service operates within a dedicated Python virtual environment to isolate dependencies.

Configuration is externalized to YAML files, enabling deployment customization without code modification. Logging is centralized to per-service log files.

1.5.3 Physical Form Factor

The robot is constructed on a wheeled chassis with differential drive (two independently controlled drive wheels plus casters for balance). The camera is mounted forward-facing for navigation and object detection. The microphone and speaker provide bidirectional audio communication. The display and LED ring are positioned for operator visibility.

Total power is supplied by lithium-ion batteries with separate regulation for logic and motor domains.

1.6 Scope and Limitations

1.6.1 Within Scope

This document covers the complete software architecture, hardware integration, and operational procedures for the Smart Car platform. Described capabilities are implemented in the codebase; verification coverage varies by subsystem.

1.6.2 Explicit Limitations

The following limitations are inherent to the current system design:

Network Dependency. High-level language understanding requires cloud API connectivity. Without network access, the system cannot interpret complex commands, though basic reflexive behaviors (collision avoidance, wake word detection) continue operating.

No Acoustic Echo Cancellation. The system cannot listen while speaking. During text-to-speech playback, the microphone is muted to prevent self-triggering. Users must wait for the robot to finish speaking before issuing new commands.

Startup Latency. Python-based services require 10-20 seconds to load AI models and establish connections. The system is not immediately responsive after power-on.

Indoor Operation Only. The ultrasonic sensors, camera calibration, and thermal management are designed for indoor environments. Outdoor operation with direct sunlight, extreme temperatures, or weather exposure is not supported.

Single-User Interaction. The voice interface does not distinguish between speakers. Any person within microphone range can issue commands. Multi-user authorization is not implemented.

1.7 Chapter Summary

This chapter established the context, motivation, and scope of the Smart Car project. The system addresses fundamental architectural challenges in embedded robotics: concurrent multi-modal operation, failure isolation, resource coordination, and hardware-enforced safety.

The deployed platform combines a Raspberry Pi 4 for AI workloads with an ESP32 for time-critical control, enabling autonomous voice-controlled operation with visual perception and collision avoidance. Documented capabilities include wake word activation, speech recognition, language model interpretation, object detection, motor control, and multi-channel user feedback.

Explicit design constraints—thermal limits, power budgets, memory capacity, and network availability—shaped architectural decisions documented in subsequent chapters. Known limitations, particularly network dependency and acoustic echo cancellation absence, define operational boundaries.

The following chapters examine each subsystem in detail, beginning with the hardware architecture that enables this capability distribution.

2

Hardware Architecture

2.1 Chapter Context

This chapter documents the physical hardware platform upon which the Smart Car system operates. Key components described herein have been verified present on the deployed system accessed via SSH at runtime. The hardware architecture directly constrains software design decisions documented in subsequent chapters.

The fundamental hardware challenge in embedded robotics is achieving reliable low-latency performance within strict thermal, power, and computational budgets. This chapter establishes those constraints through measured system parameters rather than manufacturer specifications.

2.2 Dual-Brain Architecture

The system employs a dual-processor architecture separating high-level cognition from low-level reflexes. This separation is not merely organizational—it provides a hardware-enforced safety boundary that cannot be compromised by software failures.

2.2.1 The Cortex: Raspberry Pi 4 Model B

The primary compute platform is a Raspberry Pi 4 Model B (Revision 1.5) with 8 GB LPDDR4-3200 SDRAM. This specific unit has been configured for enhanced performance beyond stock parameters.

Table 2.1: Raspberry Pi 4 Specifications (Verified)

Parameter	**Measured Value**
System-on-Chip	Broadcom BCM2711
CPU Architecture	ARM Cortex-A72 (ARMv8-A)
CPU Cores	4
Clock Frequency	2.0 GHz (overclocked from 1.5 GHz)
RAM	8 GB (7.6 GB user-available)
GPU Memory	Headless allocation configured
Storage	microSD (Class 10 / UHS-I)

The Raspberry Pi executes all artificial intelligence workloads: speech recognition, language model inference, computer vision, and decision-making. It runs Debian GNU/Linux (Trixie) with kernel version 6.12.47-1+rpt1-rpi-v8.

2.2.2 The Brainstem: ESP32 DevKit

An ESP32 microcontroller handles time-critical operations that cannot tolerate Linux scheduler latency. The ESP32 operates independently with its own control loop, providing:

- Deterministic 50ms control cycle (20 Hz)
- PWM signal generation for motor control
- Analog-to-digital conversion for gas sensing
- Ultrasonic sensor polling
- Hardware collision interlock

2.2.3 Inter-Processor Communication

The Raspberry Pi and ESP32 communicate via UART at 115200 baud using `/dev/serial0` (GPIO 14/15 on Pi, GPIO 16/17 on ESP32). The protocol uses simple ASCII commands and CSV-formatted telemetry.

Table 2.2: UART Communication Parameters

Parameter	**Value**
Baud Rate	115200
Data Bits	8
Stop Bits	1
Parity	None
Pi Device	/dev/serial0
Pi TX/RX	GPIO 14/15
ESP32 RX/TX	GPIO 16/17

2.3 Overclocking Configuration

The stock Raspberry Pi 4 operates at 1.5 GHz. This deployment uses a 2.0 GHz overclock, providing a 33% increase in raw processing throughput. The overclock is configured in `/boot/config.`

```
arm_freq=2000
over_voltage=6
```

Engineering Rationale: The overclock is necessary to achieve acceptable latency for Faster-Whisper speech recognition and YOLO object detection on CPU. Without it, the voice-to-response latency would exceed user tolerance thresholds.
Thermal Consequence: The overclock increases power dissipation and requires active cooling. A metal heatsink case with dual brushless fans is used to manage thermals under sustained AI workloads.

2.4 Power Distribution

Power stability is critical for reliable operation. The system uses two isolated power rails sharing a common ground reference.

2.4.1 Dual-Rail Architecture

- **Logic Rail (5V, 3A)**: Official Raspberry Pi USB-C power supply. Powers the Pi, camera, USB microphone, and TFT display.
- **Drive Rail (12V, Li-Ion)**: Battery pack. Powers the L298N motor driver, ESP32, and DC motors.

Common Ground: Both rails share a ground reference to ensure UART logic levels (3.3V) between the Pi and ESP32 are correctly referenced. Without common ground, communication would be unreliable or impossible.

2.4.2 Power Budget Constraints

Table 2.3: Power Consumption by Subsystem (Estimated)

Component	**Rail**	**Current**
Raspberry Pi 4 (idle)	Logic	600 mA
Raspberry Pi 4 (AI load)	Logic	1.5-2.0 A
USB Camera	Logic	200 mA
USB Microphone	Logic	50 mA
TFT Display	Logic	100 mA
NeoPixel Ring (full white)	Logic	400 mA
ESP32	Drive	50-200 mA
DC Motors (×4, stall)	Drive	2-4 A

Critical Constraint: Motor stall current can cause voltage sag on the drive rail. The logic rail must be isolated to prevent brownout-induced Pi crashes during motor stall events.

2.5 Sensor Subsystems

2.5.1 Vision: Camera Module

A 5-megapixel camera module provides visual input for object detection. Configuration verified from `config/system.yaml`:

- Camera Index: 0
- Resolution: Downscaled to 640×640 for inference
- Target FPS: 15 (rate-limited to manage thermal load)
- Interface: CSI or USB (configuration-dependent)

2.5.2 Audio: USB Sound Interface

Audio capture uses a USB sound card with an attached microphone. The ALSA device `smartcar_capture` is configured in `system.yaml`; shared access (dsnoop) is optional and depends on host configuration.

- Hardware Sample Rate: 48000 Hz
- Resampled for STT: 16000 Hz
- Preferred Device: “USB Audio” substring match

2.5.3 Proximity: Ultrasonic Sensors

Three HC-SR04 ultrasonic sensors provide forward-arc obstacle detection. Each sensor is connected to dedicated GPIO pins on the ESP32.

Table 2.4: Ultrasonic Sensor Pinout (ESP32)

Position	**Trigger Pin**	**Echo Pin**
Left (S1)	GPIO 4	GPIO 5
Center (S2)	GPIO 18	GPIO 19
Right (S3)	GPIO 21	GPIO 22

The sensors are polled sequentially (not simultaneously) to prevent ultrasonic cross-talk interference.

2.5.4 Environmental: Gas Sensor

An MQ2 gas sensor connected to ESP32 ADC provides air quality monitoring.

- Pin: GPIO 34 (ADC1_CH6, input-only)
- Output: Raw 12-bit integer (0-4095)
- Sensitivity: Alcohol, volatile organic compounds

2.6 Actuator Subsystems

2.6.1 Locomotion: DC Motors and Driver

Four DC gear motors provide differential drive locomotion. The L298N H-bridge motor driver translates ESP32 GPIO signals to motor power.

Table 2.5: Motor Control Pinout (ESP32 → L298N)

Function	**ESP32 GPIO**	**L298N Input**
Left Motor Forward	25	IN1
Left Motor Backward	26	IN2
Right Motor Forward	27	IN3
Right Motor Backward	14	IN4

Motor direction is controlled by GPIO logic levels. Speed control (PWM) is available but the current implementation uses full-on/full-off control.

2.6.2 Visual Feedback: NeoPixel LED Ring

An 8-LED NeoPixel ring provides visual state indication. The software defaults to board pin `D12` (GPIO12) using the `neopixel` library.

2.6.3 Display: TFT Screen

A 3.5-inch TFT display connected via SPI renders facial expressions and status graphics. Configuration:

- Resolution: 480×320 pixels
- Rotation: set in software (deployment uses 180 in `face_fb`)
- SPI Bus/Device: 0/0
- Framebuffer: /dev/fb0 (deployment) or /dev/fb1 (alternate framebuffer)

2.7 Operating System Environment

2.7.1 Linux Distribution

Table 2.6: Operating System Details

Parameter	**Value**
Distribution	Debian GNU/Linux
Release	Trixie (Testing)
Kernel	6.12.47-1+rpt1-rpi-v8
Architecture	aarch64 (ARMv8-A)
Init System	systemd

The use of kernel 6.12 (rather than the typical 6.6 LTS in Bookworm) provides access to newer scheduler improvements (EEVDF) and hardware support.

2.7.2 Boot Timing

System startup time affects operational readiness. Critical-path analysis of the boot sequence:

Table 2.7: Boot Critical Chain (Observed in audit logs)

Service	**Time**	**Blocking**
NetworkManager-wait-online	(observed)	Yes
systemd-udev-settle	(observed)	Yes
smart-car-orchestrator	(observed)	No

Observation: Network wait can delay service readiness. Local subsystems could operate earlier but are currently coupled to orchestrator readiness.

2.8 Hardware Bill of Materials

Table 2.8: Complete Hardware Bill of Materials

Component	**Model/Spec**	**Interface**
Single-Board Computer	Raspberry Pi 4B 8GB	-
Microcontroller	ESP32 DevKit	UART
Motor Driver	L298N H-Bridge	GPIO
DC Motors	Gear motors ×4	L298N
Ultrasonic Sensors	HC-SR04 ×3	GPIO
Gas Sensor	MQ2	ADC
Camera	5MP CSI/USB	CSI/USB
Microphone	USB Sound Card	USB
Speaker	USB/3.5mm	USB/Audio
Display	3.5” TFT	SPI
LED Ring	NeoPixel 8-LED	GPIO 12
Power Supply (Logic)	5V 3A USB-C	USB-C
Power Supply (Drive)	Li-Ion Battery	Barrel
Cooling	Metal Case + Fans	-

2.9 Chapter Summary

The Smart Car hardware architecture implements a dual-brain design with clear responsibility separation. The Raspberry Pi 4 handles AI workloads while the ESP32 manages time-critical motor control and safety interlocks. This separation provides hardware-enforced safety boundaries independent of software state.

Key hardware parameters verified at runtime:

- CPU: 2.0 GHz overclocked Cortex-A72
- RAM: 8 GB (7.6 GB available)
- UART: 115200 baud between Pi and ESP32
- Sensors: 3× ultrasonic, 1× gas, 1× camera
- Actuators: 4× DC motors via L298N
- Feedback: 8-LED NeoPixel ring, 3.5” TFT display

The dual-rail power architecture isolates motor noise from logic circuits. Active cooling maintains thermal stability under continuous AI workloads. The next chapter examines how software architecture maps onto this hardware foundation.

3

Software Architecture

3.1 Chapter Context

This chapter presents the software architecture of the Smart Car system as verified through runtime observation. The architecture addresses a fundamental challenge in embedded robotics: enabling concurrent multi-modal operation (audio, vision, motor control) on resource-constrained hardware without resource contention or cascading failures.

The solution employs a Service-Oriented Architecture (SOA) where independent processes communicate through a message bus, supervised by a central orchestrator implementing a finite state machine.

3.2 Architectural Principles

Three core principles guided the software architecture design:

3.2.1 Process Isolation

Each functional capability runs as an independent operating system process. A crash in the vision service does not terminate the voice pipeline. Memory exhaustion in one service does not corrupt another. This isolation is enforced by the Linux kernel, not by application code.

3.2.2 Message-Driven Communication

Services communicate exclusively through a publish-subscribe message bus. No service directly calls methods on another service. No service holds references to another service. This decoupling enables:

- Independent service restart without system-wide impact
- Service replacement without interface changes
- Asynchronous operation by design

3.2.3 Explicit State Management

The system's operational state is managed by a single orchestrator process implementing a finite state machine. State transitions are explicit, logged, and constrained. Illegal state combinations are structurally impossible rather than merely discouraged.

3.3 Service Topology

Eight services compose the runtime system. Each service is managed by systemd, ensuring automatic startup, crash recovery, and orderly shutdown.

Table 3.1: Service Inventory (Configured)

Service	Module	Role
orchestrator	src.core.orchestrator	Central state machine, event routing
voice-pipeline	src.audio.voice_service	Wakeword detection, speech-to-text
llm	src.llm.local_llm_runner	Local LLM integration
tts	src.tts.piper_runner	Text-to-speech synthesis
vision	src.vision.vision_runner	Object detection (YOLO)
uart	src.uart.bridge	ESP32 communication
display	src.ui.face_fb	TFT screen rendering
led-ring	src.piled.led_ring_service	NeoPixel status indication

3.4 Message Bus Architecture

Inter-process communication uses ZeroMQ (ZMQ) with a publish-subscribe pattern over TCP sockets. This choice over alternatives (ROS, D-Bus, Unix sockets) was deliberate:

- **ZMQ vs ROS**: ZMQ is lightweight; ROS introduces substantial runtime overhead
- **ZMQ vs D-Bus**: ZMQ supports pub/sub payloads; D-Bus is optimized for method calls
- **ZMQ vs Unix Sockets**: ZMQ provides built-in pub/sub semantics; Unix sockets require custom multiplexing

3.4.1 Dual-Bus Topology

Two ZMQ buses separate event flow by direction:

Table 3.2: IPC Bus Configuration

Bus	Address	Direction	Purpose
Upstream	tcp://127.0.0.1:6010	Service → Orchestrator	Events, sensor data
Downstream	tcp://127.0.0.1:6011	Orchestrator → Services	Commands, state updates

The orchestrator binds to both buses. All other services connect. This asymmetry means services can crash and reconnect without affecting bus availability.

3.4.2 Message Format

Messages consist of a topic (byte string) and payload (JSON-encoded dictionary). Example:

```
Topic: b"stt.transcription"
Payload: {"text": "move forward", "confidence": 0.92}
```

3.5 Data Flow Architecture

The system follows an OODA (Observe-Orient-Decide-Act) loop distributed across processes:

```
    OBSERVE                ORIENT              DECIDE              AC
    -------                ------              ------              --

    [Camera] ----+
                 |
    [Mic]   ----+---> [Orchestrator] ---> [LLM Service] ---> [Motors]
                 |          |                                    |
    [Sensors]---+           |                                [TTS]
                            |                                    |
                            v                                [Display
                    [State Machine]                              |
                            |                                [LEDs]
                            v
                    [Phase Broadcast]
```

Figure 3.1: OODA Data Flow Architecture

3.5.1 Perception Path

Sensor data flows from hardware to orchestrator:

1. **Voice Pipeline**: Microphone → Porcupine (wakeword) → Faster-Whisper (STT) → `stt.transcription`
2. **Vision Pipeline**: Camera → YOLO inference → `visn.object`
3. **ESP32 Telemetry**: Ultrasonic sensors → UART → `esp32.raw`

3.5.2 Cognition Path

User intent is processed through the LLM:

1. Orchestrator receives `stt.transcription`
2. Orchestrator constructs prompt with context (vision state, conversation history)
3. Orchestrator publishes `llm.request`
4. LLM service calls the configured backend (local or cloud)
5. LLM service publishes `llm.response` with structured JSON

3.5.3 Action Path

Commands flow from orchestrator to actuators:

1. Orchestrator parses LLM response JSON
2. Speech output: publishes `tts.speak`
3. Motor commands: publishes `nav.command`
4. State indication: publishes `display.state`

3.6 State Machine Design

The orchestrator implements a five-phase finite state machine that governs system behavior.

3.6.1 Phase Definitions

Table 3.3: Orchestrator Phases

Phase	Description
IDLE	Waiting for wakeword; vision active
LISTENING	Capturing user speech; vision paused
THINKING	Waiting for LLM response; all input paused
SPEAKING	Playing TTS audio; microphone muted
ERROR	System error; awaiting recovery

3.6.2 Transition Constraints

The state machine enforces 15 legal transitions. Any event not matching a valid (current_phase, trigger) pair is silently ignored. This design makes illegal states structurally impossible rather than merely unlikely.

Example valid transitions:

- (IDLE, wakeword) → LISTENING
- (LISTENING, stt_valid) → THINKING
- (THINKING, llm_with_speech) → SPEAKING
- (SPEAKING, tts_done) → IDLE

3.6.3 Timeout Enforcement

Each non-IDLE phase has a configurable timeout. If the expected completion event does not arrive within the timeout, the orchestrator forces a transition to IDLE (or ERROR for repeated failures). This prevents the system from becoming stuck in intermediate states.

Table 3.4: Phase Timeouts (configurable)

Phase	**Timeout**
LISTENING	Configured in `system.yaml`
THINKING	Network-dependent (no hard timeout)
SPEAKING	Duration of TTS audio + buffer

3.7 Virtual Environment Isolation

Python dependency conflicts are resolved through separate virtual environments per service category. This allows services to use incompatible library versions without conflict.

Table 3.5: Virtual Environment Assignments

Environment	**Path**	**Services**
stte	.venvs/stte	orchestrator, voice-pipeline, uart
llme	.venvs/llme	llm
ttse	.venvs/ttse	tts
visn	.venvs/visn	vision
visn-py313	.venvs/visn-py313	display, led-ring

3.8 Failure Isolation Boundaries

The architecture creates natural failure domains:

3.8.1 Process-Level Isolation

Each service runs in a separate process. A segmentation fault, infinite loop, or memory exhaustion in one service does not propagate to others. systemd's `Restart=on-failure` directive automatically restarts crashed services.

3.8.2 Bus-Level Isolation

ZMQ sockets automatically reconnect after peer failure. A service crash causes a brief message gap, not a bus-wide failure. The orchestrator continues operating and will observe the restarted service when it reconnects.

3.8.3 Hardware-Level Isolation

The ESP32 operates independently of Raspberry Pi software state. Even if the Pi kernel panics, the ESP32 continues polling ultrasonic sensors and will halt motors if obstacles are detected.

3.9 Configuration Architecture

System behavior is configured through YAML files, enabling deployment customization without code modification.

Table 3.6: Configuration Files

File	Contents
config/system.yaml	IPC addresses, model paths, service parameters
.env	API keys, credentials (not in repository)

Configuration values support environment variable substitution (e.g., `${ENV:GEMINI_API_KEY}`) allowing secrets to be injected at runtime.

3.10 Chapter Summary

The Smart Car software architecture employs Service-Oriented Architecture with ZeroMQ message passing and a central finite state machine orchestrator. Key architectural properties:

- **8 independent services** running as separate Linux processes
- **Dual ZMQ buses** (ports 6010/6011) separating upstream events from downstream commands
- **5-phase state machine** with 17 legal transitions
- **Multiple virtual environments** isolating Python dependencies
- **Hardware-level safety** via independent ESP32 collision avoidance

This architecture enables concurrent multi-modal operation with failure isolation at process, bus, and hardware levels. The following chapters examine each service in detail, beginning with the inter-process communication layer that binds them together.

4

ESP32 Firmware

This chapter documents the ESP32 microcontroller firmware that implements time-critical sensor polling, motor control, and hardware-level collision avoidance. Operating independently of the Linux host, the firmware provides bounded response times critical for physical safety.

4.1 Design Philosophy

The firmware architecture follows the “brainstem” analogy—handling reflexive responses that must occur faster than Linux process scheduling permits. While the Raspberry Pi runs the cognitive stack (voice, vision, LLM), the ESP32 maintains a strict 50 ms control loop (20 Hz) for sensor fusion and motor safety.
Key design principles:

1. **Deterministic Timing**: Fixed-interval loop targets consistent sensor sampling

2. **Hardware Interlock**: Collision avoidance cannot be overridden by software commands

3. **Transparency**: All state changes are reported to the host via UART

4.2 Hardware Interface

4.2.1 Pin Assignments

4.2.2 UART Configuration

Communication with the Raspberry Pi uses UART2:

Table 4.1: ESP32 GPIO pin assignments

Function	Pin	Type	Notes
Ultrasonic Sensor 1 (Front-Left)			
TRIG1	GPIO 4	Output	Trigger pulse
ECHO1	GPIO 5	Input	Echo return
Ultrasonic Sensor 2 (Front-Centre)			
TRIG2	GPIO 18	Output	Trigger pulse
ECHO2	GPIO 19	Input	Echo return
Ultrasonic Sensor 3 (Front-Right)			
TRIG3	GPIO 21	Output	Trigger pulse
ECHO3	GPIO 22	Input	Echo return
Motor Driver			
IN1	GPIO 25	Output	Left motor forward
IN2	GPIO 26	Output	Left motor backward
IN3	GPIO 27	Output	Right motor forward
IN4	GPIO 14	Output	Right motor backward
Additional Peripherals			
MQ2	GPIO 34	ADC	Gas sensor (ADC1_CH6)
SERVO	GPIO 23	PWM	Pan servo (0–180°)
RXD2	GPIO 16	UART RX	Pi communication
TXD2	GPIO 17	UART TX	Pi communication

```
HardwareSerial PiSerial(2);  // UART Port 2
PiSerial.begin(115200, SERIAL_8N1, RXD2, TXD2);
```

Listing 4.1: UART initialisation

Parameters: 115200 baud, 8 data bits, no parity, 1 stop bit.

4.3 Main Control Loop

The firmware executes a strict five-phase loop every iteration:

4.4 Collision Avoidance System

The collision avoidance system implements a three-zone safety model with hardware priority over software commands.

4.4.1 Safety Zones

4.4.2 Collision Check Implementation

```
void checkCollision() {
    long minDist = 9999;
```

```
+----------------------------------------------------------+
|                    MAIN LOOP (20 Hz)                     |
+----------------------------------------------------------+
|  Phase 1: READ SENSORS                                   |
|     • Trigger S1, S2, S3 sequentially (avoid crosstalk)  |
|     • Sample MQ2 analogue (0{4095)                       |
|     • Cache distances in lastSensorDistances[]           |
+----------------------------------------------------------+
|  Phase 2: COLLISION CHECK (Highest Priority)             |
|     • Find minimum distance across all sensors           |
|     • <10cm → Emergency stop, motorsEnabled = false       |
|     • 10{20cm → Warning zone, forward blocked             |
|     • >20cm → Clear, full mobility restored               |
+----------------------------------------------------------+
|  Phase 3: TELEMETRY TRANSMISSION                         |
|     • Format CSV packet with all sensor values           |
|     • Transmit via UART to Raspberry Pi                  |
+----------------------------------------------------------+
|  Phase 4: COMMAND INGESTION                              |
|     • Check USB Serial (debugging)                       |
|     • Check PiSerial (operational commands)              |
|     • Parse and execute valid commands                   |
+----------------------------------------------------------+
|  Phase 5: DEBUG OUTPUT                                   |
|     • Print sensor values to USB Serial Monitor          |
+----------------------------------------------------------+
```

Figure 4.1: ESP32 main loop execution phases.

Table 4.2: Collision avoidance zone definitions

Zone	Distance	Action	State Flags

Emergency	< 10 cm	Immediate motor stop
		Motors disabled
			`obstacleDetected=true`
			`motorsEnabled=false`

Warning	10–20 cm	Forward commands blocked
		Rotation/reverse allowed
			`inWarningZone=true`
			`motorsEnabled=true`

Clear	> 20 cm	Full mobility	All flags false

```
    int closestSensor = -1;

    // Find minimum valid distance
    for (int i = 0; i < 3; i++) {
        long d = lastSensorDistances[i];
        if (d > 0 && d < minDist) {  // Ignore timeouts (-1)
            minDist = d;
            closestSensor = i + 1;
        }
    }

    bool wasObstacle = obstacleDetected;
    bool wasWarning = inWarningZone;

    // === EMERGENCY STOP ZONE (<10cm) ===
    if (minDist <= STOP_DISTANCE_CM) {
        if (!obstacleDetected) {
            emergencyStop();
            sendCollisionAlert("EMERGENCY_STOP");
        }
        obstacleDetected = true;
        motorsEnabled = false;
        inWarningZone = true;
    }
    // === WARNING ZONE (10-20cm) ===
    else if (minDist <= WARNING_DISTANCE_CM) {
        obstacleDetected = false;
        inWarningZone = true;
        motorsEnabled = true;
        if (!wasWarning) {
            sendCollisionAlert("WARNING_ZONE");
        }
    }
    // === CLEAR ZONE (>20cm) ===
    else {
        obstacleDetected = false;
        inWarningZone = false;
        motorsEnabled = true;
        if (wasObstacle || wasWarning) {
            sendCollisionAlert("CLEAR");
        }
    }
}
```

Listing 4.2: Collision detection logic

4.4.3 Hardware Interlock Guarantee

The collision avoidance system operates at the firmware level, enforcing safety logic at the lowest layer:

- The `motorsEnabled` flag is checked in all motor command handlers
- Forward commands are rejected when `inWarningZone` or `obstacleDetected`
- The Pi's AI cannot override the hardware interlock—it can only request movements
- Even if the Pi commands "FORWARD" while an obstacle exists, the firmware refuses

4.5 Telemetry Protocol

4.5.1 Packet Format

Telemetry is transmitted as comma-separated values:

```
DATA:S1:43,S2:120,S3:99,MQ2:1240,SERVO:90,LMOTOR:0,RMOTOR:0,OBSTACLE:0,
    WARNING:0
```

Listing 4.3: Telemetry packet format

Table 4.3: Telemetry field definitions

Field	Unit	Description
S1, S2, S3	cm	Ultrasonic distances (-1 = timeout)
MQ2	ADC units	Gas sensor reading (0–4095)
SERVO	degrees	Current servo angle (0–180)
LMOTOR, RMOTOR	-255 to 255	Motor speed/direction
OBSTACLE	boolean	Emergency stop active
WARNING	boolean	Warning zone active

4.5.2 Transmission Rate

Telemetry is transmitted every loop iteration at 20 Hz. The UART bridge parses these packets and republishes to the upstream bus.

4.6 Command Protocol

4.6.1 Supported Commands

4.6.2 Command Validation

Table 4.4: ESP32 command set

Command	Parameters	Description
FORWARD	speed (0–255)	Move forward
BACKWARD	speed (0–255)	Move backward
LEFT	speed (0–255)	Rotate left
RIGHT	speed (0–255)	Rotate right
STOP	—	Halt all motors
SERVO	angle (0–180)	Set servo position
SCAN	—	360° environmental scan
STATUS	—	Request current state

```
void handleCommand(String command, String source) {
    command.trim();
    command.toUpperCase();

    if (command.startsWith("FORWARD")) {
        // === INTERLOCK CHECK ===
        if (obstacleDetected || inWarningZone) {
            sendAck("FORWARD", "BLOCKED:COLLISION");
            return;  // Refuse to execute
        }
        int speed = parseIntParam(command, 200);
        moveForward(speed);
        sendAck("FORWARD", "OK");
    }
    // ... other commands
}
```

Listing 4.4: Forward command with interlock check

4.7 Special Manoeuvres

4.7.1 360-Degree Scan

The SCAN command triggers a full environmental survey:

```
void performScan() {
    for (int pos = 0; pos < 8; pos++) {
        // Read sensors at current heading
        readSensors();
        sendData();

        // Rotate 45 degrees
        turnRight(255);
        delay(SCAN_ROTATE_TIME_MS);  // 200ms

        // Settle before next reading
        stopMotors();
        delay(SCAN_PAUSE_MS);   // 100ms
    }
}
```

Listing 4.5: 360° scan procedure

This produces eight distance readings at 0°, 45°, 90°, 135°, 180°, 225°, 270°, and 315° relative to the starting orientation. The Pi can reconstruct a polar distance map for navigation planning.

4.7.2 Emergency Stop

```
void emergencyStop() {
    // Immediately stop all motors
    digitalWrite(IN1, LOW);
    digitalWrite(IN2, LOW);
    digitalWrite(IN3, LOW);
    digitalWrite(IN4, LOW);
    leftMotorSpeed = 0;
    rightMotorSpeed = 0;
    Serial.println("!!! EMERGENCY STOP !!!");
}
```

Listing 4.6: Emergency stop implementation

4.8 Ultrasonic Sensor Management

4.8.1 Sequential Triggering

Ultrasonic sensors are triggered sequentially to prevent acoustic crosstalk:

```
long readDistance(int trigPin, int echoPin) {
    // Ensure clean trigger
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);

    // 10us trigger pulse
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);

    // Measure echo with timeout
    long duration = pulseIn(echoPin, HIGH, 30000);   // 30ms timeout

    if (duration == 0) return -1;   // Timeout

    // Convert to centimeters (speed of sound = 343 m/s)
    return duration * 0.0343 / 2;
}
```

Listing 4.7: Distance measurement

4.8.2 Timeout Handling

A 30 ms timeout prevents the loop from blocking on absent or defective sensors. Timeout readings are reported as -1 and excluded from collision calculations.

4.9 Motor Control

4.9.1 Differential Drive

The robot uses differential drive with four digital outputs:

```
void moveForward(int speed) {
    if (!motorsEnabled) return;
    digitalWrite(IN1, HIGH);  // Left forward
    digitalWrite(IN2, LOW);
    digitalWrite(IN3, HIGH);  // Right forward
    digitalWrite(IN4, LOW);
    leftMotorSpeed = speed;
    rightMotorSpeed = speed;
}

void turnLeft(int speed) {
    digitalWrite(IN1, LOW);
    digitalWrite(IN2, HIGH);  // Left backward
    digitalWrite(IN3, HIGH);  // Right forward
    digitalWrite(IN4, LOW);
    leftMotorSpeed = -speed;
    rightMotorSpeed = speed;
}
```

Listing 4.8: Motor control primitives

Note: Forward commands check `motorsEnabled`; rotation commands do not, allowing the robot to turn away from obstacles.

4.10 Timing Characteristics

The conservative 50 ms loop period provides margin for UART latency and sensor variability.

4.11 Summary

The ESP32 firmware implements the robot's safety-critical reflexes:

Table 4.5: Firmware timing budget

Operation	**Duration**
Loop period (target)	50 ms (20 Hz)
Ultrasonic read (×3)	∼30 ms max
Collision check	<1 ms
Telemetry format & send	∼5 ms
Command parsing	<1 ms
Total per loop	∼37 ms typical

1. **Deterministic Loop**: 20 Hz sensor polling ensures responsive collision detection
2. **Three-Zone Safety**: Emergency (<10 cm), Warning (10–20 cm), Clear (>20 cm)
3. **Hardware Interlock**: Firmware refuses dangerous commands regardless of Pi instructions
4. **Transparent Telemetry**: All state continuously reported for host-side logging
5. **Fail-Safe Design**: Forward motion is blocked when obstacles are detected

The firmware serves as the robot's last line of defence against physical collisions, operating independently of the Linux scheduler and AI stack.

5

Audio Pipeline

5.1 Chapter Context

The audio pipeline enables voice interaction with the robot. This chapter documents the unified voice pipeline architecture as verified running on the deployed system. The audio subsystem solves a particularly challenging embedded systems problem: how can multiple audio consumers (wakeword detection, speech-to-text) share a single microphone without resource conflicts?

Voice interaction is the primary user interface. The quality of this pipeline—latency, accuracy, reliability—directly determines user experience.

5.2 Deployment Constraints

5.2.1 Hardware Constraints

The ALSA audio subsystem on Linux provides exclusive device access by default. Only one process can open the microphone at a time. Early system designs used separate wakeword and STT processes, causing “Device busy” errors during handoff.

5.2.2 Latency Budget

Users expect voice responses within 3 seconds of speaking. The audio pipeline contributes:

- Wakeword detection: Low-latency response is required
- Speech capture: Duration of user speech + silence detection

- STT inference: Model-dependent

5.2.3 Thermal Constraints

Speech-to-text inference can create CPU spikes. Sustained inference would increase thermal load, so the pipeline returns to low-power wakeword monitoring between interactions.

5.3 Unified Pipeline Architecture

The production system uses `UnifiedVoicePipeline` (`src/audio/unified_voice_pipeline.py`), a single-process design that owns the microphone exclusively and multiplexes audio to internal consumers.

5.3.1 Thread Model

Table 5.1: Pipeline Thread Architecture

Thread	**Role**	**Blocking**
Main Thread	State machine, ZMQ pub/sub	Non-blocking poll
Audio Thread	Ring buffer writer	Blocks on ALSA read
Wakeword Thread	Porcupine processing	Consumes from ring buffer

5.3.2 State Machine

The pipeline implements a four-state machine:

Table 5.2: Pipeline State Machine

State	**Description**	**Audio Consumer**
IDLE	Waiting for wakeword	Porcupine
CAPTURING	Recording user speech	Ring buffer accumulation
TRANSCRIBING	Running STT inference	None (audio paused)
COOLDOWN	Post-TTS pause	None (prevents self-trigger)

5.3.3 State Transitions

```
IDLE → [wakeword detected] → CAPTURING
CAPTURING → [silence detected] → TRANSCRIBING
TRANSCRIBING → [STT complete] → COOLDOWN
COOLDOWN → [timeout] → IDLE
```

5.4 Audio Capture

5.4.1 Hardware Configuration

Audio capture uses a USB sound card with ALSA. Configuration from `config/system.yaml`:

Table 5.3: Audio Capture Configuration

Parameter	Value
ALSA Device	smartcar_capture (dsnoop)
Hardware Sample Rate	48000 Hz
STT Sample Rate	16000 Hz (resampled)
Buffer Size	20 ms
Wakeword Frame	30 ms
STT Chunk	500 ms

5.4.2 Shared Access via dsnoop

The ALSA `dsnoop` plugin enables multiple processes to read from the same capture device. The system configures this in `/etc/asound.conf`. However, the unified pipeline makes this unnecessary—a single process owns the device.

5.5 Wakeword Detection

5.5.1 Porcupine Engine

Wakeword detection uses Picovoice Porcupine, a commercial wake word engine optimized for embedded deployment.

Table 5.4: Porcupine Configuration

Parameter	Value
Engine	porcupine
Model	hey-veera_en_raspberry-pi_v3_0_0.ppn
Sensitivity	0.75
Keywords	“hey veera”, “veera”, “vira”, “vera”
Frame Length	512 samples (32 ms at 16 kHz)

5.5.2 Performance Characteristics

Table 5.5: Wakeword Detection Performance (Not measured in this audit)

Metric	Value
Detection Latency	Not measured
CPU Usage (idle monitoring)	Not measured
False Positive Rate	Not formally measured

Porcupine runs on-device with minimal CPU overhead, making continuous monitoring practical even on battery power.

5.6 Speech-to-Text

5.6.1 Faster-Whisper Engine

Speech-to-text uses Faster-Whisper, a CTranslate2-optimized implementation of OpenAI Whisper.

Table 5.6: STT Configuration (from config/system.yaml)

Parameter	**Value**
Engine	faster_whisper
Model	tiny.en
Compute Type	int8 (quantized)
Device	cpu
Beam Size	1 (greedy decoding)
Language	en

5.6.2 Voice Activity Detection

The pipeline uses RMS-based silence detection rather than a neural VAD:

```
rms = sqrt(mean(samples^2)) / 32768.0
is_speech = rms > silence_threshold
```

Configuration:

- **Silence Threshold**: 0.25 (normalized RMS)
- **Silence Duration**: 800 ms before capture ends
- **Maximum Capture**: 15 seconds
- **Minimum Confidence**: 0.3 (transcript filtering)

Design Rationale: RMS detection was chosen over neural VAD for simplicity and predictable behavior. Neural VAD performance under motor noise was not validated in this audit.

5.6.3 Transcription Process

1. Captured audio is written to a temporary WAV file
2. Faster-Whisper loads the file and runs inference
3. Segments are concatenated into final transcript
4. Result published to `stt.transcription` topic

5.6.4 Performance Characteristics

Table 5.7: STT Performance (Not measured in this audit)

Metric	Value
Inference Time (simple command)	Not measured
CPU Usage (during inference)	Not measured
Memory (model loaded)	Not measured
Real-time Factor	Not measured

5.7 Text-to-Speech

5.7.1 Piper Engine

Text-to-speech uses Piper, an open-source neural TTS system.

Table 5.8: TTS Configuration

Parameter	Value
Voice	en-us-amy-medium
Sample Rate	22050 Hz
Output Device	plughw:3,0
Playback Command	aplay

5.7.2 Performance

TTS synthesis begins after receiving text; perceived latency depends on model and runtime conditions.

5.8 Resource Consumption

5.8.1 Memory Profile

Table 5.9: Voice Pipeline Memory Usage (Measured)

Component	Memory
Voice Pipeline Process (total)	239 MB RSS
Porcupine Model	∼2 MB
Faster-Whisper Model (tiny.en int8)	∼200 MB
Audio Buffers	∼10 MB

5.8.2 CPU Profile

Table 5.10: CPU Usage by State

State	CPU Usage
IDLE (wakeword monitoring)	0.2%
CAPTURING	5-10%
TRANSCRIBING	180% (2 cores saturated)

5.9 Latency Analysis

5.9.1 End-to-End Voice Latency

Table 5.11: Latency Breakdown: Wakeword to Transcript (illustrative; not measured in this audit)

Stage	Duration
Wakeword Detection	Not measured
User Speech	Variable (user-dependent)
Silence Detection	800 ms (configured window)
STT Inference	Not measured
Total (post-speech)	Not measured

5.9.2 Bottleneck Analysis

LLM latency varies by backend and network conditions. End-to-end voice-to-response time depends on speech duration, STT inference time, and LLM/TTS runtimes.

The 8.15-second network wait at boot (NetworkManager-wait-online) means the voice pipeline is effectively non-functional until WiFi connects.

5.10 Failure Modes

5.10.1 Microphone Unavailable

If the USB sound card is disconnected or another process holds the device, the pipeline fails to start. systemd will retry with `Restart=on-failure`.

5.10.2 Wakeword False Positives

The system may trigger on sounds similar to “Veera.” The cooldown state prevents immediate re-triggering from TTS output, but external sounds can cause false activations.

5.10.3 STT Timeout

If STT inference exceeds the configured timeout, the orchestrator cancels the listening session and returns to IDLE. This prevents the system from getting stuck waiting for transcription.

5.10.4 Self-Triggering

Without the COOLDOWN state, the robot's own TTS output could trigger the wakeword detector. The current implementation mutes listening during and briefly after speech output.
Limitation: There is no acoustic echo cancellation (AEC). The microphone hardware-mutes during TTS rather than filtering the speaker output from the microphone signal. This prevents "barge-in" (interrupting the robot while it speaks).

5.11 Chapter Summary

The audio pipeline implements voice interaction through a unified single-process architecture. Key verified characteristics:

- **Wakeword**: Porcupine, < 50 ms detection, 0.2% CPU idle
- **STT**: Faster-Whisper tiny.en int8, $\sim$800 ms inference
- **TTS**: Piper en-us-amy-medium, < 200 ms to first audio
- **Memory**: 239 MB total for voice pipeline process
- **States**: IDLE $\rightarrow$ CAPTURING $\rightarrow$ TRANSCRIBING $\rightarrow$ COOLDOWN

The unified pipeline eliminates microphone resource conflicts that plagued earlier multi-process designs. The absence of acoustic echo cancellation is a known limitation preventing barge-in capability. The next chapter examines the vision pipeline that operates in parallel with voice interaction.

6

Human Interface

This chapter documents the visual feedback systems that communicate robot state to human observers. The design follows the principle that system status must be instantly discernible without consulting logs or diagnostic tools.

6.1 Design Philosophy

Two complementary output modalities provide status feedback:

1. **LED Ring**: Peripheral vision indicator—visible from any angle
2. **TFT Display**: Detailed status for close interaction

Both systems derive their state primarily from `TOPIC_DISPLAY_STATE` messages published by the orchestrator. The LED service also listens to health events for error override.

6.2 LED Ring Service

6.2.1 Hardware Configuration

The LED ring uses an 8-pixel NeoPixel (WS2812B) connected to board pin `D12` (GPIO12):

```python
class LedRingHardware:
    def __init__(self, pixel_pin_attr: str, pixel_count: int,
                 brightness: float, dry_run: bool) -> None:
        if dry_run:
            return
        pixel_pin = getattr(board, pixel_pin_attr)
        order = getattr(neopixel, "GRB", neopixel.GRB)
        self._pixels = neopixel.NeoPixel(
            pixel_pin,
            pixel_count,
            brightness=brightness,
            auto_write=False,
            pixel_order=order,
        )
```

Listing 6.1: LED hardware initialisation

Table 6.1: LED ring hardware parameters

Parameter	Value
Pixel Count	8
GPIO Pin	D12 (GPIO12)
Protocol	WS2812B
Colour Order	GRB
Default Brightness	0.25

6.2.2 State-to-Pattern Mapping

Each orchestrator phase maps to a distinct visual pattern:

Table 6.2: LED state patterns and meanings

State	Pattern	User Interpretation
`idle`	Dim cyan breathing	“I am awake and waiting”
`wakeword_detected`	Bright green flash	“I heard my name”
`listening`	Blue rotating sweep	“I am recording your voice”
`transcribing`	Purple pulse	“Converting speech to text”
`thinking`	Pink pulse	“Generating response”
`tts_processing`	Orange pulse	“Creating speech audio”
`speaking`	Dark green chase	“Playing audio response”
`error`	Red blink	“Something is wrong”

6.2.3 Animation Implementation

```
class LedAnimator:
    """LED = f(Phase). Each phase has a distinct pattern."""

    def __init__(self, hardware: LedRingHardware) -> None:
        self.hw = hardware
        self.current_state = "idle"
        self._last_render = 0.0
        self._state_entered = 0.0

    def set_state(self, state: str) -> None:
        """Set LED state. Only changes on actual state change."""
        if state != self.current_state:
            self.current_state = state
            self._last_render = 0.0  # Force immediate refresh
            self._state_entered = time.time()
```

Listing 6.2: LED animator state management

6.2.4 Pattern Generation

Example breathing pattern for idle state:

```
def _render_breathing(self, base_color: RGB, period: float = 2.0) -> list[RGB]:
    elapsed = time.time() - self._state_entered
    # Sinusoidal brightness modulation
    brightness = 0.3 + 0.7 * (0.5 + 0.5 * math.sin(2 * math.pi * elapsed / period))
    return [(int(c * brightness) for c in base_color)] * self.hw.pixel_count
```

Listing 6.3: Breathing animation mathematics

6.2.5 Dry-Run Mode

For development without hardware:

```
if board is None or neopixel is None:
    self.logger.error("Board/NeoPixel modules unavailable; forcing dry-run mode")
    self._dry_run = True
    return
```

Listing 6.4: Graceful hardware absence handling

6.3 TFT Display Service

6.3.1 Framebuffer Rendering

The display uses direct framebuffer access, avoiding X11 overhead. The `display_runner` implementation uses pygame:

```
# Set SDL to use framebuffer before importing pygame
os.environ.setdefault("SDL_VIDEODRIVER", "fbcon")
os.environ.setdefault("SDL_FBDEV", "/dev/fb1")
```

Listing 6.5: Framebuffer configuration

This “headless graphics” approach avoids the overhead of a desktop environment.

6.3.2 Display Parameters

Table 6.3: TFT display configuration

Parameter	Value
Model	Waveshare 3.5” TFT
Resolution	480 × 320 pixels
Interface	SPI
Framebuffer Device	`/dev/fb0` (deployment)
Driver	`fbcon` (pygame)

6.3.3 State Visualisation

```python
class DisplayState(Enum):
    IDLE = auto()
    LISTENING = auto()
    THINKING = auto()
    SPEAKING = auto()
    NAVIGATING = auto()

@dataclass
class DisplayStatus:
    state: DisplayState = DisplayState.IDLE
    text: str = "Ready"
    direction: Optional[str] = None
    vision_label: Optional[str] = None
    vision_paused: bool = False
    last_update: float = 0.0
```

Listing 6.6: Display state enumeration

6.3.4 Colour Scheme

```python
COLORS = {
    "bg_idle": (20, 30, 60),
    "bg_listening": (20, 80, 40),
    "bg_thinking": (80, 70, 20),
    "bg_speaking": (100, 50, 20),
    "bg_navigating": (30, 30, 80),
    "text_primary": (255, 255, 255),
    "text_secondary": (180, 180, 180),
    "accent": (0, 200, 255),
    "face_outline": (240, 240, 255),
    "face_fill": (10, 15, 30),
    "eye_fill": (240, 240, 255),
    "eye_pupil": (10, 10, 20),
    "mouth": (240, 160, 120),
    "blush": (255, 192, 203),
}
```

Listing 6.7: Display colour definitions

6.3.5 Expressive Face Rendering

The `display_runner` implementation renders a stylised face whose expression reflects system state:

Table 6.4: Face expression mapping

State	Expression
IDLE	Neutral expression, eyes at rest
LISTENING	Eyes widen, pupils dilate (simulating attention)
THINKING	One eyebrow raises, mouth flattens (concentration)
SPEAKING	Mouth animates in sync with TTS amplitude
NAVIGATING	Eyes track movement direction

6.3.6 Display Renderer Class

```
class DisplayRenderer:
    """Renders status to the TFT display using pygame."""

    def __init__(self, width: int = 480, height: int = 320,
                 fb_device: str = "/dev/fb1"):
        self.width = width
        self.height = height
        self.fb_device = fb_device
        self.screen: Optional[pygame.Surface] = None
        self.font_large: Optional[pygame.font.Font] = None
        self.font_medium: Optional[pygame.font.Font] = None
        self.font_small: Optional[pygame.font.Font] = None
        self.animation_frame = 0
        self._initialized = False
```

Listing 6.8: Display renderer initialisation

6.4 Event Subscription

Both services subscribe to orchestrator state updates:

```python
from src.core.ipc import (
    TOPIC_CMD_LISTEN_START,
    TOPIC_CMD_LISTEN_STOP,
    TOPIC_CMD_PAUSE_VISION,
    TOPIC_LLM_REQ,
    TOPIC_LLM_RESP,
    TOPIC_NAV,
    TOPIC_STT,
    TOPIC_TTS,
    TOPIC_VISN,
    TOPIC_WW_DETECTED,
    make_subscriber,
)
```

Listing 6.9: Display service topic subscriptions

The `display_runner` implementation can optionally show:

- Current navigation direction (arrows)
- Vision detection labels (when not paused)
- Most recent transcript text

6.5 Systemd Service Units

6.5.1 LED Ring Service

```ini
[Unit]
Description=Smart Car LED Ring Status Indicator
After=network.target

[Service]
Type=simple
User=root  # Required for GPIO access
WorkingDirectory=/home/dev/smart_car
ExecStart=/home/dev/smart_car/.venvs/visn-py313/bin/python \
          -m src.piled.led_ring_service --config config/system.yaml
Restart=on-failure
RestartSec=3

[Install]
WantedBy=multi-user.target
```

Listing 6.10: led-ring.service

Note: The LED service runs as root for GPIO access via `/dev/gpiomem`.

6.5.2 Display Service

```
[Unit]
Description=Smart Car TFT Display Service
After=network.target

[Service]
Type=simple
User=dev
WorkingDirectory=/home/dev/smart_car
ExecStart=/home/dev/smart_car/.venvs/visn-py313/bin/python \
          -m src.ui.face_fb --state=BASE --rotate=180 --swap-rb --eye-
    scale=0.75 --mouth-scale=0.75 --blush-scale=0.4 --fbdev=/dev/fb0
Restart=on-failure
RestartSec=3

[Install]
WantedBy=multi-user.target
```

Listing 6.11: display.service

6.6 Resource Consumption

Table 6.5: Human interface service resource usage (not measured in this audit)

Service	**Memory (RSS)**	**CPU**
LED Ring	Not measured	Not measured
Display	Not measured	Not measured

The modest resource footprint allows both services to run continuously without impacting core functionality.

6.7 Debugging with Visual Feedback

The LED state protocol enables rapid debugging:

1. **Cyan breathing**: System healthy, awaiting input
2. **Green flash**: Wakeword detected—if missing, check microphone
3. **Blue sweep**: Audio capture—if stuck, check voice pipeline
4. **Purple/Pink pulse**: Processing—if long, check STT/LLM services
5. **Green chase**: Speaking—if silent, check TTS or audio output
6. **Red blink**: Error state—check logs for specifics

This visual language allows users to diagnose issues without connecting a terminal.

6.8 Summary

The human interface layer provides:

1. **Immediate Feedback**: LED patterns visible from any angle
2. **Detailed Status**: TFT display shows text and navigation state
3. **Phase Consistency**: Both outputs derive from single source (orchestrator)
4. **Low Overhead**: Framebuffer rendering avoids desktop environment
5. **Graceful Degradation**: Dry-run modes handle missing hardware

The design principle—“if a human cannot tell what the system is doing by looking at the LEDs alone, the design is wrong”—aims to make internal state visible without requiring technical instrumentation.

7

Vision Pipeline

This chapter presents the computer vision subsystem responsible for object detection and environmental perception. The pipeline employs a multi-threaded architecture to decouple camera capture from neural network inference, aiming to process current visual data rather than stale buffered frames.

7.1 Design Rationale

Embedded vision systems face a fundamental tension between capture rate and inference latency. OpenCV's default buffering behaviour accumulates frames during slow processing, causing the neural network to analyse images that may be seconds old. For a mobile robot, this temporal disconnect renders obstacle detection useless—the robot may have already collided by the time the "obstacle" is detected.

The solution implemented in `vision_runner.py` employs a producer-consumer pattern with explicit frame discarding. A dedicated capture thread runs independently of inference, continuously replacing its output buffer with the most recent frame. The inference thread takes snapshots from this buffer on demand to improve temporal currency.

7.2 Architecture Overview

The vision subsystem comprises three primary components:

1. **LatestFrameGrabber**: Threaded camera capture with buffer management

2. **VisionPipeline**: ONNX Runtime inference wrapper

3. **Detection Publisher**: ZeroMQ output to downstream consumers

```
+------------------------------------------------------------------+
|                       VISION RUNNER PROCESS                      |
+------------------------------------------------------------------+
|   +-------------------+      +------------------------------+    |
|   |  FrameGrabber     |      |      Main Inference Loop     |    |
|   |  (Daemon Thread)  |      |                              |    |
|   |                   |      |  1. get_frame()              |    |
|   |  Camera (V4L2)    |-Lock-|  2. check stale (>500ms)     |    |
|   |       ↓           |      |  3. preprocess → ONNX        |    |
|   |  cv2.read()       |      |  4. NMS post-process         |    |
|   |       ↓           |      |  5. publish detections       |    |
|   |  _latest_frame    |      |  6. sleep(0.001) yield       |    |
|   +-------------------+      +------------------------------+    |
|                                        |                         |
|                              ZMQ PUB: tcp://127.0.0.1:6010       |
|                                        |                         |
|                              Topic: TOPIC_VISN                   |
+------------------------------------------------------------------+
```

Figure 7.1: Vision pipeline thread architecture showing decoupled capture and inference.

7.3 LatestFrameGrabber Implementation

The `LatestFrameGrabber` class improves frame currency through several mechanisms:

7.3.1 Minimal Buffering

```python
self.cap = cv2.VideoCapture(camera_index)
self.cap.set(cv2.CAP_PROP_BUFFERSIZE, 1)  # Minimal buffer
```

Listing 7.1: Camera initialisation with minimal buffering

Setting `CAP_PROP_BUFFERSIZE` to 1 instructs the V4L2 driver to maintain only a single frame buffer, reducing maximum latency from the default 4–5 frames to a single frame period.

7.3.2 Rate-Limited Capture

```python
def run(self) -> None:
    last_time = 0.0
    while not self._stop_event.is_set():
        now = time.perf_counter()
        if now - last_time < self.frame_interval:  # 1/15 = 66.7ms
            time.sleep(0.001)
            continue
        ret, frame = self.cap.read()
        if not ret or frame is None:
            time.sleep(0.01)
            continue
        with self._lock:
            self._latest_frame = frame
            self._frame_time = now
        last_time = now
```

Listing 7.2: Capture loop with rate limiting

The target frame rate of 15 FPS (configurable via `system.yaml`) balances responsiveness against thermal constraints.

7.3.3 Thread-Safe Access

```
def get_frame(self) -> tuple[Optional[np.ndarray], float]:
    with self._lock:
        if self._latest_frame is None:
            return None, 0.0
        return self._latest_frame.copy(), self._frame_time
```

Listing 7.3: Atomic frame retrieval with timestamp

The frame is copied under lock to prevent data races with the capture thread. The timestamp enables stale frame rejection in the inference loop.

7.4 YOLO Detection Model

The detection model is YOLO11 Nano exported to ONNX format, selected for its balance of accuracy and embedded performance.

7.4.1 Model Configuration

Table 7.1: Vision model configuration from `system.yaml`

Parameter	Value
Model Path	`models/vision/yolo11n.onnx`
Backend	ONNX Runtime (CPU)
Input Size	640 × 640 pixels
Confidence Threshold	0.25
IoU Threshold (NMS)	0.45
Label Set	COCO 80 classes
Target FPS	15
Camera Index	0

7.4.2 Detection Data Structure

```python
@dataclass(slots=True)
class Detection:
    label: str                  # Class name from COCO
    confidence: float           # 0.0 to 1.0
    bbox: Tuple[int, int, int, int]  # (x1, y1, x2, y2)
```

Listing 7.4: Detection dataclass

The use of `slots=True` reduces memory overhead per detection instance, relevant when processing multiple objects per frame.

7.5 Inference Pipeline

7.5.1 Preprocessing

Input images undergo the standard YOLO preprocessing chain:

1. **Resize**: Bilinear interpolation to 640×640
2. **Normalise**: Pixel values scaled from $[0, 255]$ to $[0.0, 1.0]$
3. **Transpose**: HWC to CHW format for ONNX
4. **Batch**: Add batch dimension $\rightarrow$ shape $(1, 3, 640, 640)$

7.5.2 Post-Processing

YOLO outputs require Non-Maximum Suppression (NMS) to eliminate redundant detections:

1. Filter by confidence threshold (0.25)
2. Apply class-agnostic NMS with IoU threshold (0.45)
3. Map class indices to COCO label strings

7.6 Performance Characteristics

7.6.1 Measured Throughput

With the CPU overclocked to 2.0 GHz, the vision pipeline achieves:

The discrepancy between nominal target (15 FPS) and effective throughput (3.5 FPS) arises from ONNX Runtime CPU inference dominating the pipeline latency.

Table 7.2: Vision pipeline performance metrics (from available logs)

Metric	Measured Value
Inference Latency	Not measured in this audit
Effective FPS	0.8–1.2 FPS (Picamera2 run log)
Memory (RSS)	Not measured in this audit
Camera Capability	Not measured
Target Rate Limit	15 FPS

Table 7.3: Thermal impact of vision processing (not measured in this audit)

Condition	CPU Temperature
Idle (no vision)	Not measured
Vision active (continuous)	Not measured
Throttle threshold	80°C

7.6.2 Thermal Behaviour

To prevent thermal throttling, the inference loop includes an explicit scheduler yield:

```
time.sleep(0.001)  # Yield to OS scheduler
```

Listing 7.5: Thermal mitigation yield

This prevents the vision process from monopolising CPU time, ensuring audio interrupts receive timely service.

7.6.3 Memory Stability

The `LatestFrameGrabber` pattern overwrites rather than accumulates frames, reducing risk of unbounded memory growth. Long-run RSS stability was not measured in this audit.

7.7 ZeroMQ Output Protocol

Detections are published to `TOPIC_VISN` on the upstream ZeroMQ bus:

```
def publish_detections(pub_sock, detections, ts, request_id=None):
    for det in detections:
        payload = {
            "label": det.label,
            "bbox": [x1, y1, x2, y2],
            "confidence": det.confidence,
            "ts": ts
        }
        if request_id:
            payload["request_id"] = request_id
        publish_json(pub_sock, TOPIC_VISN, payload)
```

Listing 7.6: Detection publishing

When no detections occur, a null detection is published to maintain heartbeat:

```
{"label": "none", "bbox": [0, 0, 0, 0], "confidence": 0.0, "ts": 1706294400.123}
```

7.8 Operational Modes

The vision runner supports multiple operational modes:

Continuous Mode Default operation—infers on every available frame

On-Demand Mode Responds to `TOPIC_CMD_VISN_CAPTURE` requests with single-frame inference

Pause Mode Halts inference when `TOPIC_CMD_PAUSE_VISION` is received (thermal management)

Test Mode Uses synthetic frames for unit testing without camera hardware

7.9 Stale Frame Protection

Frames older than 500 ms are discarded to prevent acting on obsolete visual data:

```
frame, frame_time = grabber.get_frame()
if frame is None:
    continue
age_ms = (time.perf_counter() - frame_time) * 1000
if age_ms > 500:
    logger.warning(f"Discarding stale frame: {age_ms:.0f}ms old")
    continue
```

Listing 7.7: Stale frame rejection

This protection activates during system overload or when inference cannot keep pace with capture.

7.10 Summary

The vision pipeline implements a robust object detection capability within the thermal and computational constraints of embedded hardware. Key design decisions:

1. **Frame Currency**: Producer-consumer pattern guarantees temporal relevance
2. **Thermal Awareness**: Explicit yields prevent CPU monopolisation
3. **Memory Stability**: Frame overwriting prevents unbounded growth
4. **Graceful Degradation**: Stale frame rejection maintains system integrity

Measured throughput varies by camera pipeline and lighting conditions; available logs show sub-2 FPS under Picamera2 runs. The target 15 FPS remains a configuration goal.

8

Inter-Process Communication

8.1 Chapter Context

Inter-process communication (IPC) forms the nervous system of the Smart Car architecture. All coordination between services—perception, cognition, and actuation—flows through message channels. This chapter documents the verified IPC infrastructure as observed at runtime.

The IPC design solves a critical embedded systems problem: how can independent processes communicate reliably without shared memory corruption, deadlocks, or tight coupling that propagates failures?

8.2 Deployment Constraints

8.2.1 Latency Requirements

The voice interaction loop imposes soft latency constraints. Users expect response within a few seconds of speaking. The IPC layer must contribute minimal latency to the overall budget.

8.2.2 Reliability Requirements

Services crash and restart during normal operation. The IPC layer must tolerate service restarts without requiring full system reboot. Message loss during restart is acceptable; message corruption is not.

8.2.3 Resource Constraints

The Raspberry Pi provides limited memory. IPC infrastructure should not require substantial RAM allocation for buffers or connection state.

8.3 ZeroMQ Bus Architecture

The system uses ZeroMQ (ZMQ) for inter-process messaging. ZMQ was selected over alternatives for specific reasons documented from the V&V audit.

8.3.1 Technology Selection Rationale

Table 8.1: IPC Technology Comparison

Technology	Startup Time	Memory	Decision
ROS 2	High	High	Rejected: Too heavy
D-Bus	Moderate	Low	Rejected: Method-call oriented
Unix Sockets	Low	Minimal	Rejected: No pub/sub
ZeroMQ	Low	Low	Selected

ZMQ provides publish-subscribe semantics with TCP reliability and automatic reconnection after peer failure.

8.3.2 Dual-Bus Topology

Two ZMQ buses separate message flow by direction:

Table 8.2: ZMQ Bus Configuration (Verified from config/system.yaml)

Name	Address	Direction
Upstream	tcp://127.0.0.1:6010	Services → Orchestrator
Downstream	tcp://127.0.0.1:6011	Orchestrator → Services

Binding Pattern: The orchestrator binds to both buses. All other services connect. This asymmetry ensures:

- The orchestrator is the stable endpoint
- Services can crash and reconnect without bus disruption
- Multiple services can subscribe to the same topics

8.3.3 Measured Performance

Table 8.3: ZMQ Performance Characteristics

Metric	Measured Value
Transport	TCP loopback (127.0.0.1)
Message Latency	Not measured in this audit
Reconnection Time	Automatic (value not measured)
Socket Linger	Requires explicit configuration

8.4 Topic Catalogue

All 17 topics are defined in `src/core/ipc.py` as byte literals. Topics are organized by function.

8.4.1 Event Topics (Upstream)

Events flow from sensors and services to the orchestrator.

Table 8.4: Upstream Event Topics

Topic	Publisher	Payload
`ww.detected`	voice-pipeline	{keyword, confidence}
`stt.transcription`	voice-pipeline	{text, confidence, optional timing}
`llm.response`	llm	{speak, direction, track}
`visn.object`	vision	{detections: [label, confidence, bbox]}
`esp32.raw`	uart	{sensor telemetry fields}
`system.health`	various	{service, status, memory}

8.4.2 Command Topics (Downstream)

Commands flow from the orchestrator to actuator services.

Table 8.5: Downstream Command Topics

Topic	Subscriber	Payload
`llm.request`	llm	{text, vision?, direction}
`tts.speak`	tts	{text}
`nav.command`	uart	{direction, optional speed/duration}
`cmd.pause.vision`	vision	{pause: bool}
`cmd.listen.start`	voice-pipeline	{}
`cmd.listen.stop`	voice-pipeline	{}
`display.state`	display	{state, phase, timestamp}

8.4.3 Unused Topics

Topics are defined centrally in `ipc.py`. Usage varies by service and deployment configuration; topics not observed at runtime should be treated as optional.

8.5 Message Format

All ZMQ messages use a two-part multipart format:

```
Part 0: Topic (bytes)    Example: b"stt.transcription"
Part 1: Payload (bytes)  Example: b'{"text": "move forward", "confidence": 0.9
```

8.5.1 Payload Encoding

Payloads are JSON-encoded UTF-8 strings. The `publish_json()` helper function handles serialization:

```
def publish_json(sock, topic, payload):
    sock.send_multipart([topic, json.dumps(payload).encode()])
```

8.5.2 Subscription Filtering

ZMQ topic filtering is prefix-based. Subscribing to `b"visn"` receives all messages starting with those bytes. The current implementation uses exact topic matching for clarity.

8.6 UART Bridge

The UART bridge provides the interface between the ZMQ bus and the ESP32 microcontroller.

8.6.1 Serial Configuration

Table 8.6: UART Parameters (Verified from config/system.yaml)

Parameter	Value
Device	/dev/serial0
Baud Rate	115200
Data Bits	8
Stop Bits	1
Parity	None
Timeout	1.0 second

8.6.2 Throughput Analysis

At 115200 baud with 8N1 encoding (10 bits per byte):

```
Maximum throughput: 115200 / 10 = 11,520 bytes/second
Typical command size: ˜20 bytes
Command transmission time: 20 / 11520 ≈ 1.7 ms
```

The serial link is not a bottleneck for motor control. Network latency to cloud APIs dominates the system latency budget.

8.6.3 Protocol Format

Commands to ESP32 use simple ASCII tokens:

```
FORWARD
BACKWARD
LEFT
RIGHT
STOP
SCAN
RESET
CLEARBLOCK
```

Telemetry from ESP32 uses CSV format:

```
DATA:S1:43,S2:120,S3:99,MQ2:1240,OBSTACLE:0
```

8.7 Failure Modes

8.7.1 Service Crash

When a service crashes, its ZMQ socket closes. The orchestrator observes no messages from that topic but continues operating. When the service restarts, ZMQ automatically reconnects.
Observed Behavior: Service crashes cause message gaps; recovery depends on restart timing.

8.7.2 Socket Linger Issue

If a service crashes without proper socket cleanup, the TCP port may remain in TIME_WAIT state and delay restart.
Mitigation: Services should set `ZMQ_LINGER=0` to close sockets immediately on process termination.

8.7.3 Message Ordering

ZMQ provides in-order delivery within a single socket. Messages from different publishers have no guaranteed ordering. The orchestrator handles this by treating each message independently.

8.8 Design Rationale

8.8.1 Why Not ROS?

ROS (Robot Operating System) is the standard in academic robotics. It was rejected for this project because:

1. **Startup Time**: ROS node spinup is heavier than ZeroMQ on Pi-class hardware
2. **Memory Footprint**: ROS runtime has a higher footprint
3. **Complexity**: Full dependency tree (DDS, type system) exceeds project needs

ZMQ provides the required functionality (pub/sub, reliable delivery, automatic reconnection) without the overhead.

8.8.2 Why TCP Not IPC?

ZMQ supports `ipc://` transport using Unix domain sockets, which would have lower latency than TCP loopback. TCP was chosen for:

1. **Debugging**: TCP traffic is visible to standard network tools
2. **Future Flexibility**: Could expose buses to remote monitoring
3. **Measured Adequacy**: Latency is acceptable for the system's response budget

8.9 Chapter Summary

The IPC layer implements a dual-bus ZeroMQ architecture with 17 defined topics. Key verified characteristics:

- **Upstream bus** (port 6010): event topics from sensors to orchestrator
- **Downstream bus** (port 6011): command topics from orchestrator to actuators
- **Message format**: Topic bytes + JSON payload
- **UART bridge**: 115200 baud serial to ESP32
- **Latency**: Not measured in this audit

The IPC design prioritizes reliability and failure isolation over raw performance. Services can crash and restart without affecting the message bus. The next chapter examines the audio pipeline that publishes wakeword and transcription events to this bus.

9

Service Orchestration

This chapter details the orchestrator service that coordinates all system components through a centralised state machine. The orchestrator implements the “single source of truth” principle—all subsystems derive their behaviour from the current orchestrator phase rather than maintaining independent state.

9.1 Architectural Role

The orchestrator occupies the apex of the service hierarchy:
The orchestrator binds both ZeroMQ buses (ports 6010 and 6011), establishing itself as the sole entity that must survive for the system to function.

9.2 Phase State Machine

The orchestrator implements a finite state machine with five phases:

9.2.1 State Transition Table

```
TRANSITIONS = {
    (Phase.IDLE, "wakeword"): Phase.LISTENING,
    (Phase.IDLE, "auto_trigger"): Phase.LISTENING,
    (Phase.IDLE, "manual_trigger"): Phase.LISTENING,
    (Phase.LISTENING, "stt_valid"): Phase.THINKING,
    (Phase.LISTENING, "stt_invalid"): Phase.IDLE,
    (Phase.LISTENING, "stt_timeout"): Phase.IDLE,
    (Phase.THINKING, "llm_with_speech"): Phase.SPEAKING,
                    +-----------------+
                    |   Orchestrator  |
                    |  (State Owner)  |
                    +--------+--------+
                             |
              Downstream Bus | tcp://127.0.0.1:6011
            (Commands/State) |
    +------------+-----------+-----------+------------+
    |            |           |           |            |
+---+---+  +-----+----+ +----+----+ +----+---+ +------+-----+
| Voice |  |  Vision  | |   LLM   | |  TTS   | |   LED/UI   |
|Pipeline|  |  Runner  | | Service | |Service | |  Services  |
+---+---+  +-----+----+ +----+----+ +----+---+ +------+-----+
    |            |           |           |            |
    +------------+-----------+-----------+------------+
                             |
              Upstream Bus   | tcp://127.0.0.1:6010
              (Events)       |
                    +--------+--------+
                    |   Orchestrator  |
                    |  (Event Sink)   |
                    +-----------------+
```

Figure 9.1: Orchestrator as central coordinator binding upstream events to downstream commands.

Table 9.1: Orchestrator phase definitions

Phase	Description
IDLE	Passive monitoring; awaiting wakeword or auto-trigger
LISTENING	Audio capture active; voice pipeline recording user speech
THINKING	LLM inference in progress; awaiting response generation
SPEAKING	TTS playback active; outputting synthesised speech
ERROR	Recovery mode; attempting to restore normal operation

```
    (Phase.THINKING, "llm_no_speech"): Phase.IDLE,
    (Phase.SPEAKING, "tts_done"): Phase.IDLE,
    (Phase.IDLE, "health_error"): Phase.ERROR,
    (Phase.LISTENING, "health_error"): Phase.ERROR,
    (Phase.THINKING, "health_error"): Phase.ERROR,
    (Phase.SPEAKING, "health_error"): Phase.ERROR,
    (Phase.ERROR, "health_ok"): Phase.IDLE,
    (Phase.ERROR, "error_timeout"): Phase.IDLE,
}
```

Listing 9.1: Orchestrator transition definitions

Invalid transitions are logged and ignored, preventing illegal state combinations.

9.2.2 Transition Logic

```
def _transition(self, event_type: str) -> bool:
    key = (self._phase, event_type)
    next_phase = self.TRANSITIONS.get(key)
    if next_phase is None:
        logger.debug("IGNORED: event '%s' illegal in phase %s",
                     event_type, self._phase.name)
        return False
    if next_phase == self._phase:
        return False
    old_phase = self._phase
    self._phase = next_phase
    self._phase_entered_ts = time.time()
    logger.info("PHASE: %s -> %s (event: %s)",
                old_phase.name, next_phase.name, event_type)
    return True
```

Listing 9.2: Phase transition implementation

9.3 Phase Entry Actions

Each phase transition triggers coordinated commands to downstream services.

9.3.1 Entering LISTENING

```
def _enter_listening(self, from_wakeword: bool = False) -> None:
    self._last_interaction_ts = time.time()
    if from_wakeword:
        self._publish_led_state("wakeword_detected")
    else:
        self._publish_led_state("listening")
    # Pause vision to reduce CPU contention
    publish_json(self.cmd_pub, TOPIC_CMD_PAUSE_VISION, {"pause": True})
```

```python
    # Start audio capture
    publish_json(self.cmd_pub, TOPIC_CMD_LISTEN_START, {"start": True})
```

Listing 9.3: LISTENING phase entry

Note the vision pause command—this reduces CPU load during audio capture, preventing STT quality degradation.

9.3.2 Entering THINKING

```python
def _enter_thinking(self, text: str, vision: Optional[Dict] = None) -> None:
    self._publish_led_state("thinking")
    payload: Dict[str, Any] = {"text": text}
    if vision:
        payload["vision"] = vision
    payload["direction"] = self._last_nav_direction
    publish_json(self.cmd_pub, TOPIC_LLM_REQ, payload)
```

Listing 9.4: THINKING phase entry

The LLM request includes current navigation state and optional vision context for multimodal reasoning.

9.3.3 Entering SPEAKING

```python
def _enter_speaking(self, text: str, direction: Optional[str] = None) -> None:
    self._publish_led_state("tts_processing")
    if direction and direction != "stop":
        self._last_nav_direction = direction
        publish_json(self.cmd_pub, TOPIC_NAV, {"direction": direction})
    publish_json(self.cmd_pub, TOPIC_TTS, {"text": text})
```

Listing 9.5: SPEAKING phase entry

Navigation commands extracted from LLM responses are dispatched before TTS begins.

9.4 LED State Protocol

The orchestrator publishes granular LED states for user feedback:

```python
def _publish_led_state(self, state: str) -> None:
    publish_json(self.cmd_pub, TOPIC_DISPLAY_STATE, {
        "state": state,
        "phase": self._phase.name,
        "timestamp": int(time.time()),
    })
```

Listing 9.6: LED state publication

Table 9.2: LED state definitions with colour assignments

State	Colour/Pattern	Meaning
`idle`	Dim cyan breathing	Awaiting wakeword
`wakeword_detected`	Bright green flash	"I heard you"
`listening`	Bright blue sweep	Capturing audio
`transcribing`	Purple pulse	STT processing
`thinking`	Pink pulse	LLM processing
`tts_processing`	Orange pulse	Generating speech
`speaking`	Dark green chase	Playing audio
`error`	Red blink	System error

9.5 Auto-Trigger Mechanism

The orchestrator supports periodic auto-triggering for proactive interaction:

```python
orch_cfg = self.config.get("orchestrator", {}) or {}
self.auto_trigger_enabled = bool(orch_cfg.get("auto_trigger_enabled", True))
self.auto_trigger_interval = float(orch_cfg.get("auto_trigger_interval", 60.0))
```

Listing 9.7: Auto-trigger configuration

When enabled, the system autonomously enters LISTENING every 60 seconds (configurable). In the current configuration, auto-trigger is disabled by default.

9.6 Systemd Integration

9.6.1 Service Unit File

```
[Unit]
Description=Smart Car Orchestrator (ZMQ event router)
After=network.target
StartLimitIntervalSec=0

[Service]
Type=simple
User=dev
WorkingDirectory=/home/dev/smart_car
Environment=PROJECT_ROOT=/home/dev/smart_car
Environment=PYTHONPATH=/home/dev/smart_car
ExecStart=/home/dev/smart_car/.venvs/stte/bin/python \
          -m src.core.orchestrator
Restart=on-failure
RestartSec=3
StandardOutput=append:/home/dev/smart_car/logs/orchestrator.log
StandardError=append:/home/dev/smart_car/logs/orchestrator.log
```

```

[Install]
WantedBy=multi-user.target
```

Listing 9.8: orchestrator.service

9.6.2 Boot Order

The orchestrator starts after network is available but does not depend on other services. It binds the ZeroMQ buses first; agents connect afterwards:

```
NetworkManager (wait-online) ----------------> [Online]
                                                   |
Orchestrator ---------> [Binds 6010/6011] --------+
                                                   |
Voice Pipeline ---------> [Connects] -------------+
Vision Runner ----------> [Warms ONNX] -----------+
LLM Service ------------> [Loads Model] ----------+
```

9.7 Failure Handling

9.7.1 Service Crashes

Systemd provides automatic restart for user-space crashes:

```
Restart=on-failure
RestartSec=3
```

9.7.2 ZeroMQ Socket Recovery

The dual-bus architecture isolates failure domains:

- **Agent Crash**: Agents use `connect()`, so their sockets close cleanly. The bus remains operational.
- **Orchestrator Crash**: The orchestrator uses `bind()`. If it crashes, the TCP ports enter `TIME_WAIT` for ∼60 seconds unless `SO_REUSEADDR` is set.

9.7.3 Zombie Socket Risk

If the orchestrator crashes without properly closing ZeroMQ sockets (missing `LINGER` option), the ports may remain bound. The 3-second restart delay allows socket cleanup in most cases.

9.7.4 Hardware Watchdog

- **Status**: Not currently enabled
- **Risk**: Kernel panics leave the system unrecoverable
- **Recommendation**: Enable Broadcom hardware watchdog via `/dev/watchdog`

9.8 Resource Consumption

The orchestrator is lightweight, serving primarily as an event router:

Table 9.3: Orchestrator resource usage (not measured in this audit)

Metric	Value
Memory (RSS)	Not measured
CPU (idle)	Not measured
CPU (transition burst)	Not measured
ZeroMQ Sockets	2 (pub + sub)

9.9 Summary

The orchestrator implements the system's "single source of truth" coordination:

1. **Centralised State**: All phases managed in one process
2. **Explicit Transitions**: Invalid state changes are rejected
3. **Coordinated Commands**: Phase entry triggers downstream actions
4. **Visual Feedback**: LED states provide human-readable system status
5. **Failure Recovery**: Systemd restart with ZeroMQ auto-reconnect

The design trades off distributed resilience for implementation simplicity—a single orchestrator crash brings the system down, but the coordination logic remains tractable and debuggable.

10

Testing and Verification

This chapter presents the testing strategy employed to validate system correctness across unit, integration, and end-to-end levels. The testing approach emphasises hardware abstraction to enable rapid iteration without physical robot access.

10.1 Testing Philosophy

Embedded systems testing faces a fundamental challenge: the target hardware is often unavailable during development. The project addresses this through:

1. **Mock Interfaces**: Hardware-dependent code is abstracted behind interfaces that can be substituted with deterministic mocks
2. **Isolated IPC**: Tests use dedicated ZeroMQ ports to avoid interference with running services
3. **Simulation Modes**: Each service supports "dry-run" or "test" modes that bypass hardware

10.2 Test Infrastructure

10.2.1 Test File Organisation

```
src/tests/
+-- test_config_loader.py      # Configuration parsing validation
+-- test_ipc_contract.py       # IPC topic constant verification
+-- test_modules.py            # Module import verification
+-- test_modules_exist.py       # File existence checks
+-- test_orchestrator_flow.py   # State machine integration
+-- test_pi_inference.py        # Vision inference mocking
+-- test_stt_fast_sim.py        # Faster-Whisper simulation
+-- test_stt_sim.py             # STT pipeline simulation
+-- test_stt_wrapper_sim.py     # STT wrapper simulation
+-- test_wakeword_sim.py        # Porcupine simulation

scripts/
+-- test_ipc_integration.py     # ZeroMQ roundtrip testing
+-- test_uart_nav.py            # UART command verification

run-tests/
+-- wake_to_stt_e2e.sh          # Wakeword→STT→LLM E2E
+-- llm_e2e.sh                  # LLM response verification
```

Listing 10.1: Test file locations

10.2.2 Test Runner Configuration

Tests execute via pytest with the core virtual environment:

```
#!/usr/bin/env bash
ROOT_DIR="$(cd "$(dirname "$0")/.." && pwd)"
CORE_PYTHON="$ROOT_DIR/.venvs/core/bin/python"

export PYTHONPATH="${PYTHONPATH:-$ROOT_DIR}"

"$CORE_PYTHON" -m pytest src/tests/ -q
```

Listing 10.2: Test execution

10.3 Unit Testing

10.3.1 IPC Contract Verification

Topic constants are verified to prevent silent protocol drift:

```
from src.core.ipc import TOPIC_WW_DETECTED, TOPIC_STT

def test_topic_constants():
    assert TOPIC_WW_DETECTED == b"ww.detected"
    assert TOPIC_STT == b"stt.transcription"
```

Listing 10.3: test_ipc_contract.py

This test fails if topic names are accidentally changed, preventing message routing breakage.

10.3.2 Mock Detector Pattern

Vision tests use a deterministic mock detector:

```python
class MockDetector:
    """Deterministic detector for pipeline testing without ONNX."""

    def load(self) -> None:
        return  # No model to load

    def detect(self, frame: np.ndarray) -> list[Detection]:
        height, width = frame.shape[:2]
        bbox = (width // 4, height // 4, width * 3 // 4, height * 3 // 4)
        return [Detection(label="mock-object", confidence=0.99, bbox=bbox)
    ]
```

Listing 10.4: MockDetector for testing

This allows testing frame processing, ZeroMQ publication, and stale frame rejection without loading the YOLO model.

10.3.3 Configuration Loader Tests

```python
def test_config_loader_parses_yaml():
    config = load_config(Path("config/system.yaml"))
    assert "ipc" in config
    assert "stt" in config
    assert config["ipc"]["upstream"] == "tcp://127.0.0.1:6010"
```

Listing 10.5: Config parsing verification

10.4 Integration Testing

10.4.1 Orchestrator Flow Test

The orchestrator state machine is tested with simulated ZeroMQ messages:

```python
def test_orchestrator_sequence():
    # Use isolated ports to avoid interference
    upstream = "tcp://127.0.0.1:6210"
    downstream = "tcp://127.0.0.1:6211"
    os.environ["IPC_UPSTREAM"] = upstream
    os.environ["IPC_DOWNSTREAM"] = downstream

    ctx = zmq.Context.instance()

    # Publisher to upstream (events to orchestrator)
    pub_events = ctx.socket(zmq.PUB)
    pub_events.connect(upstream)

    # Subscriber to downstream (commands from orchestrator)
    sub_cmds = ctx.socket(zmq.SUB)
    sub_cmds.connect(downstream)
    for topic in [TOPIC_CMD_PAUSE_VISION, TOPIC_CMD_LISTEN_START, ...]:
        sub_cmds.setsockopt(zmq.SUBSCRIBE, topic)

    # Start orchestrator in background thread
    th = threading.Thread(target=run_orchestrator, daemon=True)
    th.start()
    time.sleep(0.4)  # Allow bind/setup

    # 1. Send wakeword event
    send(TOPIC_WW_DETECTED, {
        "timestamp": int(time.time()),
        "keyword": "veera",
        "confidence": 0.99
    })

    # 2. Verify orchestrator publishes expected commands
    assert receive_topic(sub_cmds, timeout=5) == TOPIC_CMD_PAUSE_VISION
    assert receive_topic(sub_cmds, timeout=5) == TOPIC_CMD_LISTEN_START
```

Listing 10.6: test_orchestrator_flow.py

10.4.2 IPC Roundtrip Testing

ZeroMQ message flow is verified end-to-end:

```
def test_ipc_roundtrip():
    pub = make_publisher(config, channel="upstream", bind=True)
    sub = make_subscriber(config, channel="upstream", bind=False)
    sub.setsockopt(zmq.SUBSCRIBE, TOPIC_STT)

    test_payload = {"text": "hello world", "confidence": 0.95}
    publish_json(pub, TOPIC_STT, test_payload)

    topic, data = sub.recv_multipart()
    assert topic == TOPIC_STT
    assert json.loads(data) == test_payload
```

Listing 10.7: IPC integration test pattern

10.5 End-to-End Testing

10.5.1 Wake-to-STT Pipeline Test

```
#!/usr/bin/env bash
set -euo pipefail

ROOT_DIR="$(cd "$(dirname "$0")/.." && pwd)"
CORE_PYTHON="$ROOT_DIR/.venvs/core/bin/python"

log(){
    printf "%s [wake_to_stt_e2e] %s\n" \
        "$(date ’+%Y-%m-%d %H:%M:%S %Z’)" "$1"
}

export PYTHONPATH="${ROOT_DIR}"

log "Executing wakeword->STT->LLM orchestration tests"
"$CORE_PYTHON" -m pytest \
    src/tests/test_wakeword_sim.py \
    src/tests/test_stt_sim.py \
    src/tests/test_orchestrator_flow.py -q

log "Wake->STT e2e tests completed"
```

Listing 10.8: wake_to_stt_e2e.sh

10.5.2 Test Scenario: Audio Injection

1. Prepare WAV file with wakeword + command audio
2. Inject via simulated microphone stream (file input mode)
3. Verify `TOPIC_WW_DETECTED` published with correct keyword
4. Verify `TOPIC_STT` published with correct transcript
5. Verify `TOPIC_LLM_REQ` contains expected command text

10.6 Quality Gates

10.6.1 Static Analysis

Table 10.1: Quality gate tools (optional)

Tool	Purpose	Configuration
ruff	Linting	Optional linting tool
mypy	Type checking	Optional static typing checks
black	Formatting	Optional formatting tool
isort	Import sorting	Optional import sorting tool

10.6.2 Type Safety for IPC

Strict typing on message payloads prevents runtime errors:

```python
from typing import TypedDict

class STTPayload(TypedDict):
    text: str
    confidence: float
    language: str
    duration_ms: int

def publish_stt_result(sock, payload: STTPayload) -> None:
    publish_json(sock, TOPIC_STT, payload)  # mypy validates structure
```

Listing 10.9: Typed IPC payload

10.7 Hardware-in-the-Loop Testing

When physical hardware is available, additional verification can be performed:

Table 10.2: Hardware-in-the-loop test matrix

Test	Procedure
Microphone latency	Measure time from clap to wakeword detection
Speaker output	Verify TTS audio reaches speaker at correct volume
Motor response	Command FORWARD, measure wheel rotation
Collision interlock	Place obstacle <10cm, verify motor refuses FORWARD
Thermal stability	Run all services for 1 hour, verify temp <70°C

10.8 Continuous Integration Considerations

The project supports CI execution without hardware:

1. Many tests run in “dry-run” or “mock” mode by default
2. ZeroMQ tests use ephemeral ports (6210/6211) to avoid conflicts
3. No GPU or accelerator required—CPU-only inference mocked
4. Test duration varies by environment and configuration

10.9 Summary

The testing strategy provides confidence through:

1. **Mock Abstraction**: Hardware interfaces substitutable with deterministic mocks
2. **Contract Testing**: IPC topic names and payload structures verified
3. **Flow Integration**: State machine transitions validated with simulated events
4. **Quality Gates**: Static analysis catches type errors and style violations
5. **E2E Scripts**: Shell scripts orchestrate full pipeline verification

The combination of unit tests (fast, isolated), integration tests (ZeroMQ message flow), and E2E tests (full scenario) provides coverage appropriate for an embedded system where hardware access is intermittent.

11

Deployment and Operations

This chapter covers the complete deployment process from bare hardware to operational system, including virtual environment strategy, systemd service configuration, and end-to-end operational flow verification.

11.1 Virtual Environment Strategy

11.1.1 The Dependency Isolation Requirement

On constrained embedded hardware, dependency conflicts are not merely inconvenient—they render the system undeployable. The project requires:

- `stte` (STT): numpy version constraints incompatible with vision stack
- `visn` (Vision): numpy version constraints incompatible with STT stack

Attempting to install these in a single environment produces `ResolutionImpossible`. The multi-venv strategy is not architectural preference—it is a hard requirement.

11.1.2 Environment Layout

All virtual environments reside in `/home/dev/smart_car/.venvs/`:
Total disk footprint: ∼2.4 GB for all environments (measured).
done

Table 11.1: Virtual environment specifications (measured)

Env	Primary Package	Disk Size	Service
`stte`	faster-whisper, ctranslate2	869 MB	voice-pipeline
`ttse`	piper-tts, sounddevice	256 MB	tts
`llme`	llama-cpp-python	269 MB	llm
`visn-py313`	onnxruntime, opencv	282 MB	vision, display, led-ring
`visn`	onnxruntime, opencv	586 MB	(legacy)
`dise`	pygame, neopixel	74 MB	(legacy)
`core`	zmq, pyyaml	43 MB	(legacy)

11.1.3 Environment Setup Script

The repository includes a legacy `setup_envs.sh` script that creates `stte`, `ttse`, `llme`, and `visn` environments in the project root without installing requirements. The deployed system uses `/home/dev/smart_car/.venvs/` and includes additional environments (`core`, `dise`, `visn-py313`).

11.1.4 Startup Latency

Virtual environment activation adds ∼200 ms to service start time due to `sys.path` manipulation. This is negligible compared to Python interpreter initialisation (∼1.5 s) and model loading (∼3–5 s for AI services).

11.2 Systemd Service Configuration

11.2.1 Service Inventory

Eight systemd units manage the runtime:

Table 11.2: Systemd service files

Service	Virtual Env	User
orchestrator.service	stte	dev
voice-pipeline.service	stte	dev
llm.service	llme	dev
tts.service	ttse	dev
vision.service	visn-py313	dev
uart.service	stte	dev
display.service	visn-py313	dev
led-ring.service	visn-py313	root

Note: `led-ring.service` runs as root for GPIO access via `/dev/gpiomem`.

11.2.2 Service Unit Template

```
[Unit]
Description=Voice Service (Wakeword + STT)
After=network.target sound.target
Conflicts=wakeword.service stt.service

[Service]
Type=simple
WorkingDirectory=/home/dev/smart_car
Environment=PYTHONPATH=/home/dev/smart_car
Environment=PYTHONUNBUFFERED=1
Environment=PROJECT_ROOT=/home/dev/smart_car
EnvironmentFile=/home/dev/smart_car/.env

# Kill any process holding the mic before starting
ExecStartPre=-/usr/bin/fuser -k /dev/snd/pcmC3D0c

ExecStart=/home/dev/smart_car/.venvs/stte/bin/python \
          -m src.audio.voice_service --config config/system.yaml

Restart=always
RestartSec=3
TimeoutStartSec=60
User=dev
Group=dev
StandardOutput=append:/home/dev/smart_car/logs/voice_service.log
StandardError=append:/home/dev/smart_car/logs/voice_service.error.log

# Resource limits for Raspberry Pi
MemoryMax=512M
CPUQuota=80%

[Install]
WantedBy=multi-user.target
```

Listing 11.1: voice-pipeline.service (representative)

11.2.3 Key Configuration Patterns

EnvironmentFile Loads secrets (API keys) from `.env`

ExecStartPre Releases audio device locks before voice service starts

Restart=always Ensures automatic recovery from crashes

RestartSec=3 Prevents restart storms during persistent failures

MemoryMax/CPUQuota Resource limits prevent runaway processes

11.3 Hardware Assembly

11.3.1 Power Distribution

1. Connect Li-Ion battery positive to L298N 12V input
2. Connect Li-Ion battery negative to L298N GND
3. Bridge L298N GND to ESP32 GND and Raspberry Pi GND (common ground)
4. Power ESP32 via 5V/VIN pin or USB
5. Power Pi 4 via official USB-C supply (5V 3A)

Critical: Common ground between all components is mandatory for UART signal integrity.

11.3.2 UART Wiring

Table 11.3: UART connections

Pi GPIO	**Direction**	**ESP32 GPIO**
GPIO 14 (TX)	→	GPIO 16 (RX)
GPIO 15 (RX)	←	GPIO 17 (TX)

11.3.3 Thermal Management

1. Apply thermal paste to Pi 4 CPU (BCM2711)
2. Mount aluminium heatsink case
3. Connect dual fans to Pi 5V and GND GPIO
4. Verify: `stress -c 4` should maintain temperature <60°C

11.4 Software Deployment Procedure

11.4.1 Initial Setup

```
# 1. Clone repository
git clone https://github.com/user/smart_car.git
cd smart_car

done
# 2. Create virtual environments (deployment-specific layout)
./setup_envs.sh

```

```
# 3. Install per-environment dependencies (if applicable)
# pip install -r requirements-<env>.txt

# 4. Fetch AI models
./scripts/fetch_whisper_fast_model.sh
./scripts/fetch_piper_voice.sh
./scripts/fetch_llm_model.sh

# 5. Flash ESP32 firmware
arduino-cli compile --fqbn esp32:esp32:esp32 src/uart/esp-code.ino
arduino-cli upload -p /dev/ttyUSB0 --fqbn esp32:esp32:esp32 src/uart/esp-code.ino

# 6. Install systemd services
sudo cp systemd/*.service /etc/systemd/system/
sudo systemctl daemon-reload

# 7. Enable and start services
sudo systemctl enable orchestrator voice-pipeline llm tts vision uart display led-ring
sudo systemctl start orchestrator
```

Listing 11.2: Deployment commands

11.4.2 Service Startup Order

Services start in dependency order:

```
network.target ----------------------------------------> [Ready]
      |
      +--> orchestrator (binds ZMQ 6010/6011) --------> [Ready]
      |
      +--> voice-pipeline (connects 6010/6011) ------> [Ready]
      +--> vision (connects 6011) -------------------> [Ready]
      +--> llm (connects 6010/6011) -----------------> [Ready]
      +--> tts (connects 6010/6011) -----------------> [Ready]
      +--> uart (connects 6011) ---------------------> [Ready]
      +--> display (connects 6010) ------------------> [Ready]
      +--> led-ring (connects 6010) -----------------> [Ready]
```

11.5 Operational Flow (Illustrative)

11.5.1 End-to-End Scenario: “Find the Bottle”

This scenario illustrates the intended processing chain:

Phase 1: Trigger

1. User says: “Hey Veera”
2. Porcupine detects wakeword
3. Voice pipeline publishes `TOPIC_WW_DETECTED`
4. Orchestrator transitions: IDLE → LISTENING
5. LED ring: Cyan breathing → Green flash → Blue sweep

Phase 2: Command Capture

1. User says: “Find the bottle”
2. VAD detects speech end (silence window)
3. Faster-Whisper transcribes audio
4. Voice pipeline publishes `TOPIC_STT`: `{"text": "Find the bottle"}`
5. Orchestrator transitions: LISTENING → THINKING
6. LED ring: Blue sweep → Pink pulse

Phase 3: Cognition

1. Orchestrator publishes `TOPIC_LLM_REQ` with:
   - User text: “Find the bottle”
   - Vision context: `{"label": "bottle", "confidence": 0.8, "bbox": [...]}`
   - Navigation state: “stopped”
2. LLM generates response:

```
{"speak": "I see it, approaching.", "direction": "forward", "track":
    "bottle"}

```

3. LLM publishes `TOPIC_LLM_RESP`

Phase 4: Action

1. Orchestrator parses LLM response
2. Publishes `TOPIC_NAV`: `{"direction": "forward"}`
3. Publishes `TOPIC_TTS`: `{"text": "I see it, approaching."}`

4. UART bridge sends a `FORWARD` command to ESP32

5. TTS generates and plays audio

6. Orchestrator transitions: THINKING → SPEAKING → IDLE

7. LED ring: Pink pulse → Orange pulse → Green chase → Cyan breathing

11.6 Monitoring and Diagnostics

11.6.1 Log File Locations

```
/home/dev/smart_car/logs/
+-- orchestrator.log
+-- voice_service.log
+-- voice_service.error.log
+-- llm.log
+-- tts.log
+-- vision.log
+-- uart.log
+-- display.log
+-- led_ring.log
```

11.6.2 Service Status Commands

```
# Check all service status
systemctl status orchestrator voice-pipeline llm tts vision uart display
    led-ring

# View recent logs
journalctl -u voice-pipeline -n 50 --no-pager

# Monitor real-time events
tail -f /home/dev/smart_car/logs/*.log

# Check resource usage
ps aux | grep -E 'orchestrator|voice|vision|llm|tts'
```

Listing 11.3: Diagnostic commands

11.6.3 Health Indicators

11.7 Summary

Deployment requires careful attention to:

Table 11.4: System health verification

Check	Expected Result
LED ring shows cyan breathing	Orchestrator running, system idle
`ss -tlnp | grep 6010`	Orchestrator bound to ZMQ port
`dmesg | grep -i usb`	Audio device detected
`cat /sys/class/thermal/thermal_zone0/temp`	<70000 (70°C)

1. **Environment Isolation**: Multiple virtual environments resolve conflicting numpy requirements

2. **Service Configuration**: Systemd units with resource limits and automatic restart

3. **Hardware Assembly**: Common ground and proper cooling are critical

4. **Startup Order**: Orchestrator must bind ZMQ buses before agents connect

5. **Verification**: End-to-end scenario testing validates complete chain

The deployment procedure, while multi-step, produces a self-healing system where crashed services automatically restart and reconnect to the ZeroMQ buses.

12

Limitations and Future Work

This chapter documents known system limitations, potential failure modes, and a roadmap for future enhancements. Honest assessment of constraints is essential for setting appropriate expectations and guiding development priorities.

12.1 Current Limitations

12.1.1 Computational Constraints

Table 12.1: Performance limitations on Raspberry Pi 4 (not measured in this audit)

Subsystem	**Target**	**Achieved**
Vision FPS	15	Not measured
STT Latency	<1s	Not measured
LLM Response	<2s	Not measured
Boot Time	<30s	Not measured

The Raspberry Pi 4's quad-core Cortex-A72 cannot sustain concurrent AI workloads. Vision inference alone consumes 100% of one core, leaving limited headroom for STT and LLM.

12.1.2 Audio Pipeline Limitations

No Acoustic Echo Cancellation (AEC) The robot must mute its speaker to listen. "Barge-in" (interrupting TTS playback) is not supported—the robot cannot hear while speaking.

Single Microphone Without a microphone array, no beamforming or noise source separation is possible. Performance degrades in noisy environments.

Fixed Sample Rate The audio pipeline assumes 16 kHz input. Microphones with different native rates require resampling, adding latency.

12.1.3 Network Dependencies

- **Cloud LLM Mode**: When configured to use cloud APIs, network outages degrade the system to local functionality only.
- **No Offline Voice**: Porcupine wakeword detection requires a valid access key. Offline operation requires pre-generated keyword models.

12.1.4 Thermal Constraints

- **Throttling Threshold**: CPU throttles at 80°C
- **Vision Impact**: Continuous inference drives temperature to $\sim$70°C
- **Combined Load**: Running vision + STT simultaneously risks throttling
- **Mitigation**: Active cooling required; vision pauses during audio capture

12.1.5 Memory Pressure

Table 12.2: Service memory footprint (not measured in this audit)

Service	**RSS**
voice-pipeline (STT)	Not measured
vision-runner	Not measured
llm-service	Not measured
orchestrator + others	Not measured
Total	Not measured
Available (8GB)	$\sim$7 GB

While 8 GB provides comfortable headroom, the 4 GB Pi 4 variant would face swap pressure.

12.1.6 Sensor Limitations

Ultrasonic Blind Spots HC-SR04 sensors have a limited cone of detection (spec-dependent). Objects outside this cone are undetected. Three sensors provide forward coverage, leaving side flanks blind.

Minimum Range Ultrasonic sensors cannot reliably measure distances <2 cm. Very close obstacles may report erroneous maximum readings.

Camera Field of View The camera's horizontal FOV limits peripheral vision (spec-dependent). Objects at the side are not detected until the robot turns.

12.2 Known Failure Modes

12.2.1 Recoverable Failures

Table 12.3: Recoverable failure scenarios

Failure	Symptom	Recovery
Service crash	LED shows red blink	Systemd restart (3s)
ZMQ timeout	Stale messages	Auto-reconnect
Microphone lock	No wakeword detection	`fuser -k` in ExecStartPre
Model file missing	Service fails to start	Systemd restart loop

12.2.2 Non-Recoverable Failures

Table 12.4: Non-recoverable failure scenarios

Failure	Symptom	Mitigation
Kernel panic	Complete freeze	Hardware watchdog (not enabled)
SD card corruption	Boot failure	Regular backups
Power brownout	Undefined state	Capacitor bank (not implemented)
Thermal shutdown	Abrupt halt	Improved cooling

12.3 Extending the System

12.3.1 Adding a New Skill

To add a “Weather” skill:

1. **Create Service**: Write `src/skills/weather_service.py` that queries a weather API

2. **Define Topic**: Add `TOPIC_WEATHER` to `ipc.py`

3. **Subscribe**: Listen for `TOPIC_LLM_RESP` containing weather intents

4. **Update Prompt**: Modify LLM system prompt to include weather capability

5. **Register**: Create systemd unit file

```python
class WeatherSkill:
    def __init__(self, config):
        self.sub = make_subscriber(config, channel="downstream")
        self.sub.setsockopt(zmq.SUBSCRIBE, TOPIC_LLM_RESP)
        self.pub = make_publisher(config, channel="upstream")

    def run(self):
        while True:
            topic, data = self.sub.recv_multipart()
            payload = json.loads(data)
            if "weather" in payload.get("intent", ""):
                result = self.fetch_weather(payload["location"])
                publish_json(self.pub, TOPIC_TTS, {"text": result})
```

Listing 12.1: Skill service template

12.3.2 Adding New Hardware

To integrate a 2D Lidar (RPLidar A1):

1. **Driver**: Write Python wrapper using `rplidar` library
2. **Publisher**: Create `lidar_runner.py` publishing to `TOPIC_LIDAR`
3. **Message Format**: Define point cloud schema (angle, distance arrays)
4. **Fusion**: Update orchestrator to merge lidar with ultrasonic data
5. **Systemd Unit**: Add `lidar.service`

12.4 Hardware Upgrade Path

12.4.1 Near-Term Upgrades

Table 12.5: Recommended hardware upgrades

Component	**Current**	**Upgrade**
AI Accelerator	CPU only	Google Coral USB (30+ FPS vision)
Lidar	3× Ultrasonic	RPLidar A1 (360° SLAM capable)
Microphone	Single USB mic	ReSpeaker 4-Mic Array (beamforming)
Compute	Pi 4 (8GB)	Pi 5 or Jetson Nano

12.4.2 Google Coral Integration

Adding a Coral USB accelerator would transform vision performance:

- Convert YOLO11n to TFLite with Edge TPU compilation

- Expected inference: ∼30 ms (vs 280 ms on CPU)
- Effective FPS: 30+ (vs 3.5)
- Frees CPU for concurrent STT without thermal pressure

12.5 Software Roadmap

12.5.1 Short-Term (3–6 Months)

1. **Hardware Watchdog**: Enable Broadcom watchdog via `/dev/watchdog`
2. **AEC Integration**: Implement software echo cancellation using speexdsp
3. **Health Dashboard**: Web-based status page via Flask/nginx
4. **OTA Updates**: Implement atomic update mechanism for field deployment

12.5.2 Medium-Term (6–12 Months)

1. **Local LLM Optimisation**: Quantised TinyLlama with speculative decoding
2. **SLAM Integration**: Occupancy grid mapping with lidar
3. **WebRTC Teleoperation**: Real-time video streaming to browser
4. **Multi-Robot Coordination**: ZeroMQ mesh for swarm behaviour

12.5.3 Long-Term Vision

1. **Full Offline Operation**: No cloud dependencies for any functionality
2. **Learning from Interaction**: On-device fine-tuning of behaviour
3. **Autonomous Navigation**: Path planning with obstacle memory

12.6 Lessons Learned

12.6.1 What Worked

1. **ZeroMQ Decoupling**: Service isolation enabled independent iteration
2. **Multi-Venv Strategy**: Resolved irreconcilable numpy version conflicts
3. **Phase-Driven Design**: Single source of truth simplified debugging
4. **Hardware Interlock**: Firmware-level collision avoidance proved essential

12.6.2 What Could Improve

1. **Earlier Thermal Profiling**: Vision thermal impact discovered late

2. **Configuration Consolidation**: Settings scattered across YAML, env, code

3. **Structured Logging**: Plain text logs harder to analyse than JSON

4. **Metric Collection**: No Prometheus/Grafana for operational monitoring

12.7 Conclusion

This project demonstrates that "industrial-grade" reliability is achievable on "hobbyist" hardware through careful engineering:

- **Constraints Drive Innovation**: The Pi 4's limitations forced efficient algorithms and service isolation

- **Layered Safety**: ESP32 hardware interlock provides last-line defence regardless of software state

- **Observable Systems**: LED feedback and structured phases enable debugging without instrumentation

The system, while limited by computational resources, provides a functional prototype for voice-controlled robotic navigation. The architecture—ZeroMQ buses, phase state machine, multi-venv isolation—scales to more capable hardware when available.
Future work should prioritise hardware acceleration (Coral TPU) and acoustic echo cancellation to enable natural conversational interaction. The foundation is solid; the path forward is clear.

Configuration Reference

.1 system.yaml Schema

```
ipc:
  upstream: "tcp://127.0.0.1:6010"
  downstream: "tcp://127.0.0.1:6011"

audio:
  sample_rate: 16000
  hw_sample_rate: 48000
  preferred_device_substring: "USB"

wakeword:
  sensitivity: 0.6
  model_path: "models/veera.ppn"

stt:
  silence_threshold: 0.35
  silence_duration_ms: 900
  engines:
    faster_whisper:
      model: "tiny.en"
      compute_type: "int8"

llm:
  engine: "gemini"
  gemini_model: "gemini-1.5-flash"
  temperature: 0.2

vision:
  camera_index: 0
  target_fps: 15.0
  model_path_onnx: "models/vision/yolo11n.onnx"
```

```
nav:
  uart_device: "/dev/ttyS0"
  baud_rate: 115200
```

Pin Reference Tables

.2 ESP32 GPIO Map

Function	**GPIO**	**Notes**
UART RX (from Pi)	16	PiSerial
UART TX (to Pi)	17	PiSerial
Motor IN1	25	Left Forward
Motor IN2	26	Left Backward
Motor IN3	27	Right Forward
Motor IN4	14	Right Backward
Ultrasonic Trig1	4	Left
Ultrasonic Echo1	5	Left
MQ3 ADC	34	Gas Sensor
Servo PWM	23	Optional

.3 Raspberry Pi GPIO Map

Function	**GPIO**	**Notes**
UART TX	14	To ESP32 RX
UART RX	15	From ESP32 TX
Neopixel PWM	12	DMA via rpi_ws281x
SPI MOSI	10	Display
SPI SCLK	11	Display
SPI CE0	8	Display CS
Display DC	25	Data/Command

References

1. **ZeroMQ Guide**. iMatix Corporation. https://zguide.zeromq.org/

2. **Faster-Whisper**. SYSTRAN. https://github.com/SYSTRAN/faster-whisper

3. **Porcupine Wake Word Engine**. Picovoice Inc. https://picovoice.ai/products/porcupine/

4. **YOLOv11**. Ultralytics. https://docs.ultralytics.com/

5. **Google Gemini API**. Google DeepMind. https://ai.google.dev/

6. **Raspberry Pi GPIO Documentation**. Raspberry Pi Foundation. https://www.raspberrypi.com/documentation/

7. **ESP32 Arduino Core**. Espressif Systems. https://docs.espressif.com/projects/arduino-esp32/

8. **Adafruit NeoPixel Library**. Adafruit Industries. https://learn.adafruit.com/adafruit-neopixel-uberguide

9. **L298N Motor Driver Datasheet**. STMicroelectronics.

10. **HC-SR04 Ultrasonic Sensor Datasheet**. Various manufacturers.