Silesian
University
of Technology

FINAL PROJECT

AI-Powered Mobile Robot

Yassine BENDIMERAD
Student identification number: yb308985

Programme: Informatics
Specialisation: Artificial Intelligence

SUPERVISOR
dr inż. Dariusz Myszor
DEPARTMENT of algorithms and software
Faculty of Automatic Control, Electronics and Computer Science

Gliwice 2026

Abstract

This thesis presents the design and implementation of **Botronka**, an AI-powered mobile robot built on Raspberry Pi 5. The robot combines computer vision, speech interaction, trust-based behavior, and mobile actuation in a modular thread-based software architecture. The goal of the project was to create a functional companion platform for entertainment that can recognize users, converse naturally, and execute safe movement commands in real time.

The implemented pipeline includes a face identity subsystem, an audio pipeline, and a motion layer with trust-aware policy constraints. The system was validated through unit tests and field trial, while solving the engineering challenges as they come such as networking, electronics failing, soldering mistakes and runtime robustness. The final result is a working robotic platform and a baseline for future improvements in autonomy, multimodal reasoning, efficiency and reliability.

Keywords

AI-powered robot; social companion robot; mobile robot; human–robot interaction; embedded AI; edge computing; computer vision; face recognition; speech interface; speech-to-text; text-to-speech; trust-aware control; multimodal interaction.

List of Abbreviations and Symbols

AI	Artificial Intelligence
LLM	Large Language Model
STT	Speech-to-Text
TTS	Text-to-Speech
VAD	Voice Activity Detection
UML	Unified Modeling Language
GPIO	General-Purpose Input/Output
FPS	Frames per Second
JSON	JavaScript Object Notation
YAML	YAML Ain't Markup Language
Wi-Fi	Wireless Fidelity
VPN	Virtual Private Network
SFace	OpenCV face recognition model used for embedding comparison
YuNet	OpenCV lightweight face detection model
OWNER/ GUEST/ FRIEND	Trust levels used by the robot policy layer

Acknowledgements

I would like to sincerely thank my supervisor **dr inż. Dariusz Myszor**, for his guidance, feedback, and support throughout this thesis project since the first contact while I was abroad.

I am also very grateful to the **SpaceCoffee student club** for helping me design the robotic platform and obtain the hardware needed to complete this work. I would like to also thank my professors, classmates, and study mates for their ideas, discussions, and encouragement during the development process and the whole program.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Mobile robots are becoming more capable thanks to lightweight AI models and affordable embedded platforms. In recent years, it became feasible to run speech and vision pipelines directly on a single-board computer.

I decided to develop the **Botronka** project to combine the multiple engineering outcomes in something I am passionate about : AI, and that in one coherent robotic platform, using face identity which was a project I did on my own in the past for attendance, so I wanted to add to it trust-aware interaction, voice conversation, and physical movement. The robot is intentionally designed as a practical prototype, where software decisions are continuously validated by hardware behavior.

1.2 Problem Statement

Many small robot projects work only as isolated demos (only vision, only voice, or only movement). The main challenge in this thesis is integration: combining those modules into a robust runtime architecture that can work in real-world conditions on the limited ressources that the Raspberry Pi hardware offers locally.

The engineering problem is therefore not only model inference, but also:

- event coordination between concurrent threads,
- trust and character enforcement before motion execution,
- automating the program,
- recovery from issues during runtime, electronics, connectivity...
- reproducibility of setup and testing.

1.3 Thesis Objectives

The main objective is to design and implement a coherent *AI-Powered Mobile Robot* that can interact with people naturally. The specific objectives are:

1. Building a modular software architecture with a shared message bus and independent worker threads.
2. Implement a vision identity pipeline for face detection, recognition, and trust-level assignment.
3. Implement an on-device audio pipeline using VAD, STT, LLM reasoning, and TTS output.
4. Integrate motion control with policy constraints based on user trust level and command safety.
5. Validate the complete system through unit tests, runtime checks, and field observations.

1.4 Scope and Limitations

The thesis focuses on a functional prototype and engineering integration quality. Using compact local models to challenge and test optimisation on Raspberry Pi resources limits, but introducing intentional trade-offs in language richness and recognition. The robot is not intended as a fully autonomous navigation system, instead, movement commands are short and controlled, with safety-aware gating.

1.5 Thesis Organization

Chapter 2 presents the problem analysis and introduces the system overview.
Chapter 3 defines project requirements and tools.
Chapter 4 describes the external specification, including hardware setup and integration.
Chapter 5 details the internal specification: software architecture, vision pipeline, audio interaction & agent reasoning, and user experience with sensor-driven behavior.
Chapter 6 covers verification and validation for testing and reproducibility.
Chapter 7 summarizes achieved results, encountered engineering difficulties, and possibilities for further development.

Chapter 2

Analysis of the problem

2.1 System Overview

Botronka is designed as an **AI-powered social companion mobile robot** with four main capabilities:

1. perceive nearby people and estimate identity/trust,

2. interact through speech in natural language,

3. execute movement commands under safety policy,

4. provide feedback through display and buzzer states.

his thesis treats Botronka as an integrated cyber-physical platform rather than an AI demo, making the system prioritize local processing and practical reliability over model size.

2.2 Functional Architecture

The runtime is built as a set of independent worker threads connected through a shared message queue. Each module publishes events and reacts to selected event types.

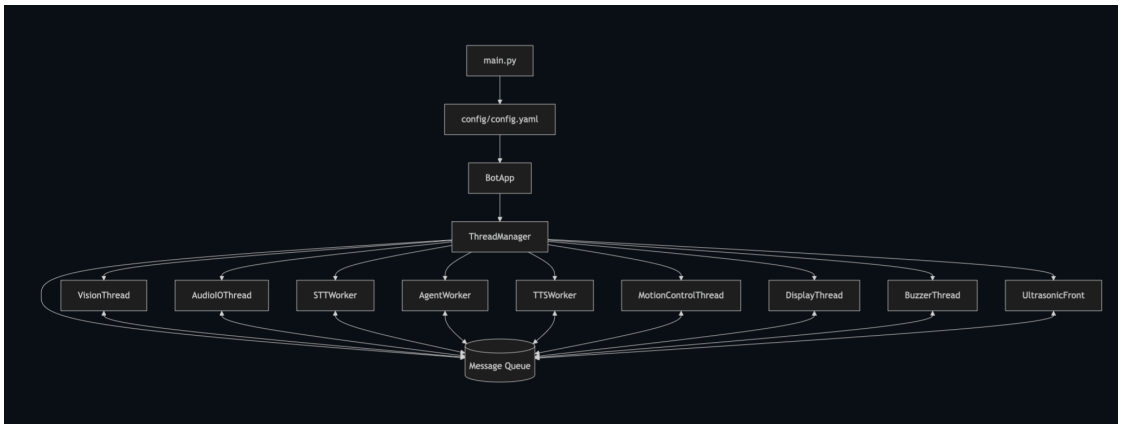


Figure 2.1: High-level software architecture (thread-based message bus design).

The event-driven approach improves modularity: vision, audio, motion, and UI can evolve independently while preserving a common communication contract.

2.3 Main Runtime Flow

At runtime, the robot follows a multimodal loop:

1. Vision detects face presence and identity confidence.
2. Audio I/O opens or closes the microphone based on face/wake state.
3. Speech To Text (STT) converts speech to text.
4. Agent module decides response and optional command.
5. Policy layer verifies trust/safety.
6. Text To Speech (TTS) and motion threads execute allowed outputs.

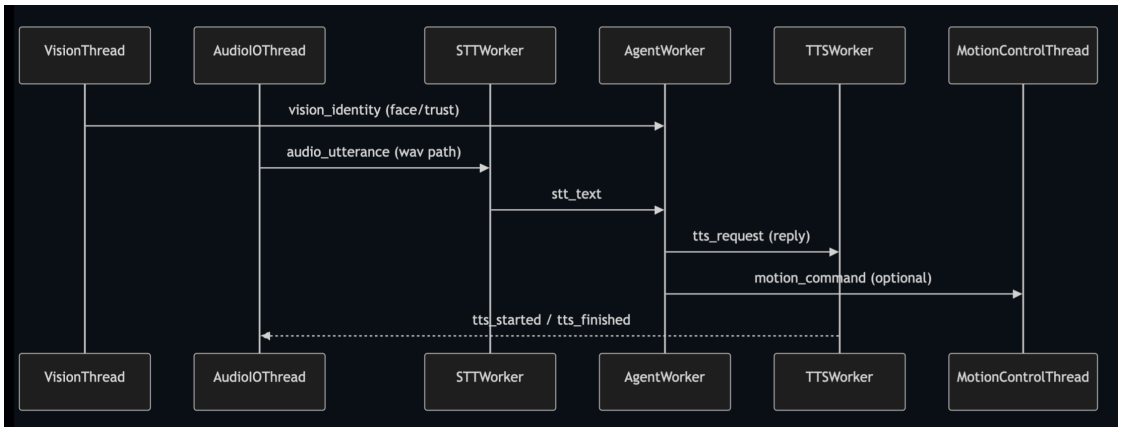


Figure 2.2: Runtime interaction flow from sensing to action.

2.4 Hardware-Software Design

The final platform prototype includes camera, microphone/speaker audio dongle, ultrasonic sensor, OLED display, buzzer, wheel motors, and steering motors. Using software interfaces to directly control hardware interfaces (GPIO mapping, frequencies, and safety thresholds).

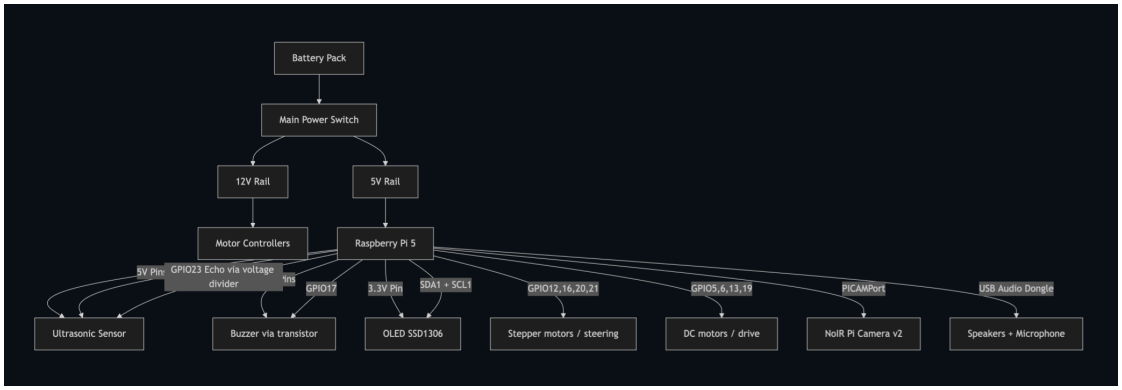


Figure 2.3: Hardware setup and key subsystem interconnections.

2.5 Operational States and Trust Logic

The robot behavior combines emotional display states and trust-aware permissions:

- **Emotion states**: `GREETING`, `HAPPY`, `SUSPICIOUS`, `LONELY`, `STUCK`, `ANGRY`, `SLEEPY`.
- **Trust levels**: `UNKNOWN` (0), `GUEST` (1), `FRIEND` (2), `OWNER` (3).

Movement-related commands are blocked for low-trust users, not only as a safety feature, especially for physical interaction in shared spaces, but also to test promotion and unlocking new controls.

2.6 Deployment Context

The robot runs on Raspberry Pi 5 with Debian-based software stack, with optional systemd auto-start. Local inference stack includes whisper.cpp for STT, llama.cpp for LLM reasoning, Piper for TTS, and YuNet + SFace models for Vision System. Keeping a setup with a manageable latency while preserving offline capability for all the interactions.

2.7 Summary

This chapter introduced Botronka as an integrated AI-mobile-robot system with message driven runtime architecture and trust aware behavior policy. The next chapter details hardware structure and electrical design decisions.

Chapter 3

Requirements and tools

3.1 Use case model

The interaction model is centered around user identity, trust level, and safe execution of actions. The main actors are:

- **Owner (admin)**,
- **Friend**,
- **Guest / Unknown User**,
- **Environment (obstacle)**.

In this model, **Unknown User** is merged with **Guest** because available actions are the same; the practical difference is reflected on Botronka's emotional/UI status rather than in new use cases.

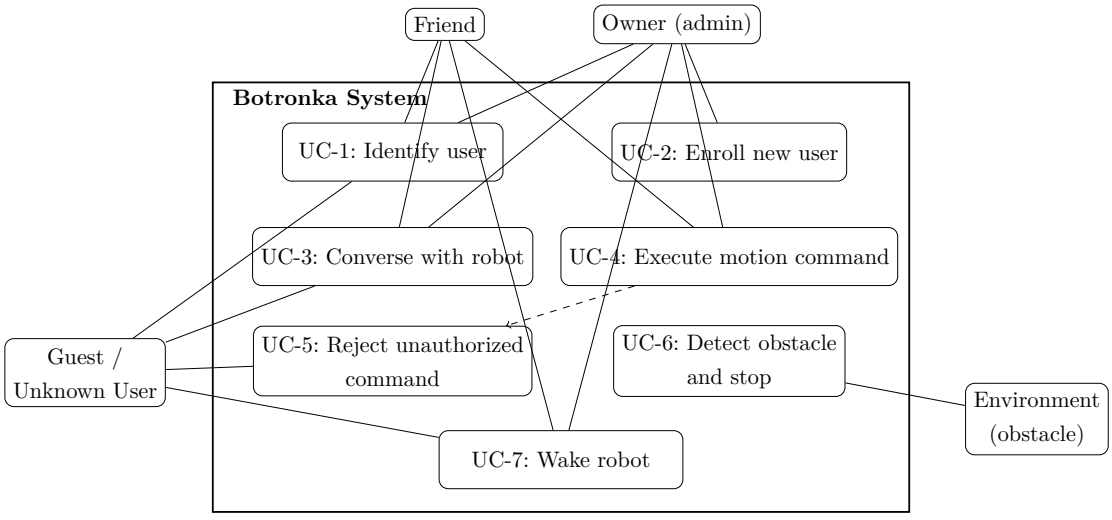


Figure 3.1: Use case diagram for trust-aware interaction and safe motion behavior.

3.2 Functional requirements

Botronka is designed as an **AI-powered social companion mobile robot**. The key functional requirements are:

1. **FR-1**: System shall detect at least one face in the camera frame at ≥ 5 FPS.
2. **FR-2**: System shall assign trust level to recognized identity.
3. **FR-3**: System shall block motion commands if `trust_level < 2`.
4. **FR-4**: System shall provide spoken feedback within ≤ 15 s after user utterance.
5. **FR-5**: System shall support enrolling a new user identity and persisting it in local storage.
6. **FR-6**: System shall detect a front obstacle and stop motion commands for collision prevention.
7. **FR-7**: System shall support wake-word based activation when no face is currently detected.
8. **FR-8**: System shall publish state updates for display and buzzer feedback.

3.3 Non-functional requirements

The software architecture was designed around practical quality goals:

1. **NFR-1 (Modularity)**: Each subsystem (vision, audio, motion, UI) should be independently maintainable and testable.
2. **NFR-2 (Responsiveness)**: Sensing and actuation loops should run concurrently with bounded latency.
3. **NFR-3 (Safety)**: Unsafe or unauthorized commands must be rejected deterministically.
4. **NFR-4 (Robustness)**: Runtime errors in one worker should not terminate the whole robot process.
5. **NFR-5 (Edge deployability)**: Core interaction pipeline should work on Raspberry Pi without mandatory cloud services.
6. **NFR-6 (Reproducibility)**: Build, run, and tests should be executable from documented commands.

3.4 Tools and technology stack

The robot runs on Raspberry Pi 5 with Debian-based software stack, NetworkManager-managed Wi-Fi profiles, and optional systemd auto-start [12, 1, 5, 10].

The full stack is local and optimized for edge deployment:

- VAD segmentation with `webrtcvad`,
- STT with `whisper.cpp`,
- LLM reasoning via `llama.cpp` server,
- speech synthesis with `Piper`.

These choices were motivated by practical efficiency and available support for Raspberry Pi-class hardware [3, 2, 6, 4, 11, 14].

For the vision and camera integration, I also relied on documented APIs and deployment guides for OpenCV face APIs and Picamera2 documentation [7, 8, 13].

3.5 Engineering and implementation approach

To achieve this, the project uses an event-driven multi-threaded architecture with a central messaging bus.

All workers communicate through a shared queue with a common message envelope:

Listing 3.1: Core message model used by worker threads.

```
@dataclass
class Message:
    sender: str
    type: str
    content: str
    sent_at: float
```

In addition to raw event flow, the system maintains a thread-safe runtime state object that tracks face presence, trust level, distance estimate, audio mode, and mute constraints due to TTS, motion, or buzzer activity.

Chapter 4

External specification

4.1 Deployment and startup procedure

A reproducible setup and execution flow is summarized below, as for autonomous startups, systemd service scripts are included in `scripts/`:

Listing 4.1: Minimal reproducible setup and run flow.

```
# setup virtual environment and install dependencies
python -m venv .venv
source .venv/bin/activate
pip install -e .



# run some unit tests
python tests/unit/test_motion.py
python tests/unit/test_wake_mode.py

# run robot
python main.py
```

4.2 Hardware Setup and Integration

4.2.1 Design Intent

The hardware platform was designed to be compact, affordable, and easy to iterate. The goal was not to maximize speed, but to create a stable physical base that supports AI perception, audio interaction, and controlled motion.

4.2.2 Core Components

Table 4.1 summarizes the main components selected for the final prototype.

Table 4.1: Main hardware components used in Botronka.

Subsystem	**Component**	**Role in system**
Compute	Raspberry Pi 5 (8GB RAM)	Main runtime host for all software threads and local AI inference
Vision	NoIR Pi Camera v2	Face detection/recognition input stream
Audio	USB audio dongle + microphone + speakers	Speech capture and spoken responses
Distance sensing	HC-SR04-style ultrasonic sensor	Front obstacle distance estimation
UI feedback	SSD1306 OLED + buzzer	Emotional display and acoustic alerts
Locomotion	DC motors + stepper steering drivers	Forward/backward movement and steering
Power	3-cell Li-ion battery pack + rails	12V motor rail + 5V logic/compute rail

4.2.3 Power and Electrical Topology

The battery output is split into two electrical domains:

- **12V rail**: motor controllers and drive power,
- **5V rail**: Raspberry Pi and low-voltage peripherals.

Power and signal routing are summarized in Figure 4.1. This separation improved stability by reducing electrical interference between motor transients and control electronics.

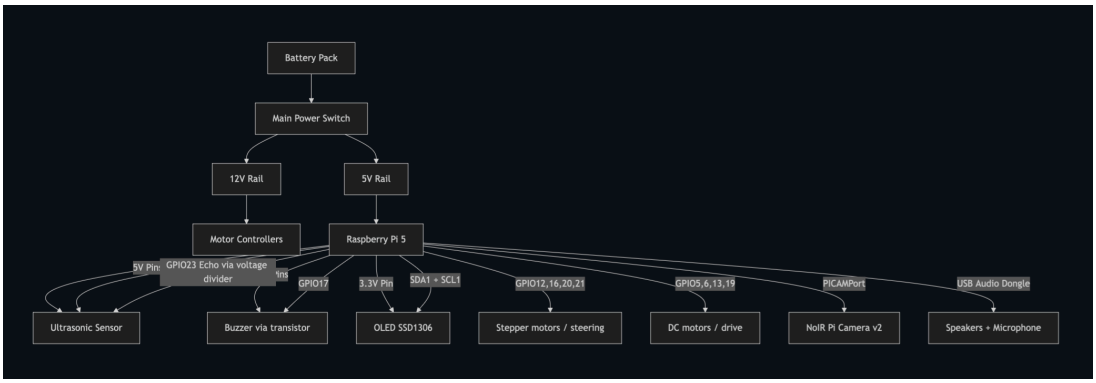


Figure 4.1: Hardware and GPIO integration map used in the final prototype.

4.2.4 GPIO Mapping

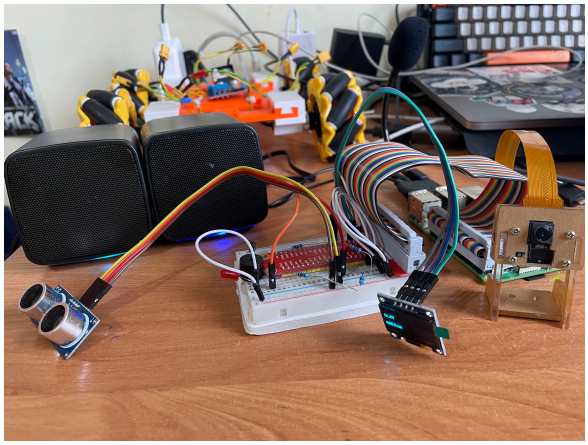
Table 4.2 presents the practical GPIO allocation used in the software configuration.

Table 4.2: GPIO pin mapping for major peripherals.

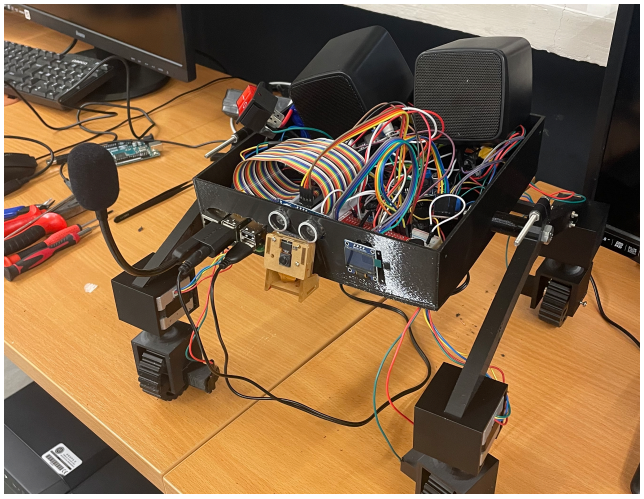
Module	Pins	Voltage	Notes
OLED (I2C)	SDA1, SCL1	3.3V	Display of emotional state and mic indicator
Buzzer	GPIO17 (via transistor)	5V	Alerts for obstacle/wake/-countdown
Ultrasonic	GPIO24 (Trig), GPIO23 (Echo divider)	5V	Front distance events at periodic rate
Stepper steering	GPIO12, GPIO16, GPIO20, GPIO21	12V	Steering range constrained in software
DC wheel control	GPIO5, GPIO6, GPIO13, GPIO19	12V	Directional control via L298N inputs

4.2.5 Mechanical Build and Iteration

The robot chassis and structural pieces were 3D printed and iteratively refined. Figures 4.2a and 4.2b show transition from development assembly to final integrated layout on the platform.



(a) Development phase, partially disassembled build.



(b) Final laboratory assembly.

Figure 4.2: Mechanical prototyping evolution.

4.2.6 Power Stability Considerations

Because the platform can be powered from GPIO 5V pins, power quality becomes critical. During deployment, undervoltage events were treated as a risk factor for unstable Wi-Fi, sensor errors, and service drops. A practical diagnostic step was added: periodic checks of system throttling status (e.g., `vcgencmd get_throttled`) to confirm stable operation under load.

4.2.7 Summary

This setup provided a good group to focus on the software stack presented in the next chapters. The architecture balances component availability, mechanical flexibility, and enough electrical robustness for continuous multimodal runtime.

Chapter 5

Internal specification

5.1 Software Architecture

The full project repository (source code, configuration files, and supplementary material) is available at: bm-Yassine/botronka-ThesisProject.

5.1.1 Architecture Goals

The software architecture was designed around three practical goals:

1. **Modularity**: each hardware/AI subsystem should remain independently maintainable.

2. **Real-time responsiveness**: sensing and action threads should run concurrently.

3. **Safety and robustness**: command execution must pass policy checks.

To acheive this, the project uses an event-driven multi-threaded architecture with a central messaging bus.

5.1.2 Application Composition

The entry point creates a global `BotApp` object, loads `config/config.yaml`, instantiates runtime components, and registers all workers in the thread manager. The key workers are: vision, audio I/O, STT, agent, TTS, motion, display, buzzer, and ultrasonic sensing.

Table 5.1: Main software threads and responsibilities.

Thread	Responsibility
`VisionThread`	Captures camera frames, detects/recognizes faces, publishes identity/trust events
`AudioIOThread`	Opens/closes microphone, records VAD utterances, handles greeting and wake probing
`STTWorker`	Converts WAV utterances to text and validates wake phrase candidates
`AgentWorker`	Interprets user text, applies fast-path rules or LLM inference, emits replies/commands
`TTSWorker`	Generates and plays speech responses, emits speaking state events
`MotionControlThread`	Parses motion commands and controls wheels
`UltrasonicFront`	Measures front distance and broadcasts obstacle events
`DisplayThread`	Maintains emotional UI state and renders OLED expressions
`BuzzerThread`	Acoustic feedback for alerts, wake confirm, and countdown patterns

5.1.3 Message-Driven Runtime

All workers communicate through a shared queue with a common message envelope:

Listing 5.1: Core message model used by worker threads.

```
@dataclass
class Message:
    sender: str
    type: str
    content: str
    sent_at: float
```

The thread manager consumes queue messages and broadcasts them to all workers. Each worker selectively processes only relevant message types and ignores others. This design decouples producers from consumers and simplifies subsystem replacement.

5.1.4 Shared Runtime State Store

In addition to raw event flow, the system maintains a thread-safe runtime state object that tracks:

- face presence and trust level,

- current distance estimate,
- audio mode (IDLE, ENGAGED, LISTENING, THINKING, SPEAKING),
- mute constraints due to TTS, motion, or buzzer activity.

This shared state prevents inconsistent behavior across threads. For example, the microphone remains closed while the robot is speaking or moving, reducing feedback loops and accidental self-triggering.

5.1.5 Command Path and Policy Enforcement

The command path is intentionally layered:

1. `STTWorker` emits `stt_text`.
2. `AgentWorker` decides between *chat* and *command* JSON response.
3. Policy module checks trust threshold and forbidden actions.
4. Allowed commands are sent to motion/buzzer threads; denied commands are converted to spoken refusal.

This approach separates language understanding from safety-critical execution and improves explainability in debugging.

5.1.6 Configuration Strategy

System behavior is controlled by structured YAML configuration:

- model paths and inference parameters,
- GPIO mappings and motion timings,
- trust thresholds and prompts,
- wake mode and greeting behavior,
- boot/runtime options.

Centralized configuration is important to re implement this project with new hardware or new setup, and keep an easy way to improve, add, and controle sensors.

5.1.7 Fault Tolerance Considerations

Each critical loop catches runtime exceptions and emits error messages (e.g., `vision_error`, `stt_error`, `tts_error`). This keeps the overall runtime alive even when a subsystem experiences transient failure.

The architecture therefore follows a practical resilience model: degrade gracefully, report errors through the message channel, and keep remaining capabilities available.

5.1.8 Summary

The software architecture provides a clear, modular foundation for multimodal robot behavior under constrained embedded hardware. The next chapter details the computer vision and identity-trust pipeline built on top of this runtime.

5.2 Vision and Identity Pipeline

5.2.1 Purpose of the Vision Module

The vision subsystem is responsible for continuous face presence detection and identity estimation. To give its output importance, I made it directly affect trust-aware interactions and movement permissions in downstream modules, and also to turn on the microphone for interactions.

The implemented pipeline is optimized for Raspberry Pi constraints and uses OpenCV Zoo models (YuNet detector and SFace recognizer) with a lightweight inference workflow [9].

5.2.2 Pipeline Stages

Each recognition cycle in `VisionThread` follows these steps:

1. capture RGB frame from Pi Camera,
2. convert to BGR and detect faces,
3. select the primary face (largest bounding box),
4. compute embedding and compare with local database,
5. apply similarity threshold and temporal stability gate,
6. publish `vision_identity` event.

The design intentionally includes a stability gate (windowed identity consensus) before assigning a known identity, reducing transient misclassification.

5.2.3 Identity and Trust Data Model

Two local JSON files are used:

- `face_db.json`: name →embedding vector,
- `trust_map.json`: name →trust level (Guest/Friend/OWNER).

Trust levels are integrated in real time and converted to a numerical score used by policy logic:

`UNKNOWN=0, GUEST=1, FRIEND=2, OWNER=3`

5.2.4 Event Payload Structure

The `vision_identity` message includes both immediate and temporal context: identity name, trust level, similarity score, face count, last-seen timestamps, and owner-seen deltas.

Table 5.2: Key fields in `vision_identity` payload.

Field	Type	Meaning
`name`	string	Recognized identity or `UNKNOWN`
`trust_level`	string	`UNKNOWN`/`Guest`/`Friend`/`OWNER`
`similarity`	float	Embedding cosine similarity to best match
`stable`	bool	Temporal stability result over recent frames
`face_detected`	bool	Presence signal for behavior state machine
`faces`	int	Number of faces detected in frame
`last_seen_ts`	float/null	Last stable timestamp for reported identity
`seconds_since_owner_seen`	float/null	Time since any owner identity was seen

5.2.5 Enrollment Workflow

In addition to passive recognition, the thread supports active enrollment. When a registration request is received:

1. multiple samples are captured from camera stream,
2. embeddings are normalized and averaged,
3. database and trust-map files are atomically updated,
4. in-memory state is refreshed without restarting the app.

This was my way to enable practical in-field onboarding of new users as `Guest` identities, and a possibility to promote them.

5.2.6 Latency and Robustness Considerations

The vision loop runs at configurable recognition FPS (lower than camera FPS) to balance CPU load with responsiveness. The `presence_hold_s` parameter prevents immediate face-loss flicker when short frame drops occur.

Exceptions are converted into `vision_error` messages instead of terminating the whole robot runtime.

5.2.7 Summary

The vision subsystem provides both perception and social context by linking face recognition to trust semantics. This makes it a central input for the audio pipeline and motion safety policy described in the next chapter.

5.3 Audio Interaction and Agent Reasoning

5.3.1 Objective of the Audio Stack

The audio subsystem transforms Botronka from a passive mobile platform into an interactive companion robot. The implemented pipeline captures speech, transcribes it, reasons about intent, applies trust-aware policy, and produces spoken feedback.

The full stack is local-first and optimized for edge deployment:

- VAD segmentation with `webrtcvad`,
- STT with `whisper.cpp`,
- LLM reasoning via `llama.cpp` server,
- speech synthesis with `Piper`.

These choices were motivated by practical efficiency and available support for Raspberry Pi-class hardware [3, 2, 6, 4].

5.3.2 Audio State Machine

The runtime uses explicit audio modes to avoid race conditions and feedback loops:

```
IDLE →ENGAGED →LISTENING →THINKING →SPEAKING
```

Mode transitions are controlled by face presence, wake phrase detection, and mutual exclusion conditions (e.g., microphone muted while TTS or motors are active).

Table 5.3: Practical meaning of audio runtime modes.

Mode	Behavior
IDLE	Mic closed; no active face and no wake override
ENGAGED	Mic can open when allowed by state constraints
LISTENING	VAD recording in progress
THINKING	STT text available; agent is computing decision
SPEAKING	TTS output active; mic blocked to prevent self-capture

5.3.3 Speech Capture and Wake Handling

`AudioIOThread` records utterances only when `RuntimeStateStore.can_open_mic()` is true. When no face is visible, a short wake probing mode is used. Candidate wake audio is sent to `STTWorker` with dedicated matching logic.

Wake phrase acceptance includes exact and fuzzy variants (e.g., "hello botronka", "wake up botronka", "listen botronka"). A successful match temporarily opens a wake override window and triggers audible confirmation.

5.3.4 Speech-to-Text Processing

STT is implemented through `whisper-cli`. The selected model in the project is `ggml-small.en.bin`, good transcription quality and inference time on Raspberry Pi. Normal and wake-candidate paths share the same backend but apply different constraints and acceptance criteria.

The STT worker also maintains queue hygiene by dropping stale wake candidates, which prevents latency growth from backlog accumulation.

5.3.5 Agent Decision Layer

After transcription, text is handled by `AgentWorker`. The decision strategy is hybrid:

1. **Fast local rules**: greetings, simple motion shortcuts, and utility prompts,
2. **LLM path**: structured JSON response for general dialogue and command interpretation.

This two-stage design reduces average latency while keeping flexibility for open-ended interactions.

Listing 5.2: Target decision schema emitted by the agent.

```
{"type":"chat|command",
 "speak":"string",
```

```
"command":"string|null",
"requires_trust":0-3}
```

5.3.6 Policy and Trust Enforcement

Before execution, commands are checked against:

- forbidden tokens (unsafe actions),
- minimum trust threshold for motion commands.

If policy denies a command, the agent converts this into a spoken refusal instead of silent failure to improve user transparency.

5.3.7 Text-to-Speech Output

TTS is implemented with Piper using `en_US-lessac-medium` voice model. The worker pre-generates common phrases at startup and caches synthesized WAV files by text hash, reducing repeated response latency.

During playback, `tts_started`/`tts_finished` events are emitted so that other modules (especially audio input) can coordinate correctly.

5.3.8 Performance-Oriented Design Choices

Several practical choices significantly improved responsiveness:

- local fast-path decisions for common intents,
- persistent HTTP session for LLM requests,
- phrase caching for TTS,
- short cooldown and bounded wake-candidate lifetime,
- explicit timing logs (VAD/STT/LLM/TTS stages).

5.4 User Experience and Sensor-Driven Behavior

5.4.1 Design Philosophy

Beyond technical functionality, Botronka was designed to feel understandable to a human user. The robot should communicate internal state clearly, avoid sudden unsafe actions, and provide immediate feedback when it is listening, thinking, or blocked.

This chapter focuses on the bridge between sensing and user perception: OLED expressions, buzzer cues, and distance-triggered behavior.

5.4.2 Emotion-Oriented Display States

The display thread maps perception and runtime context to emotional states:

- `GREETING`: face just appeared,
- `HAPPY`: normal operation,
- `SUSPICIOUS`: face detected but unknown trust,
- `LONELY`: no face for prolonged time,
- `STUCK`: obstacle too close for sustained period,
- `ANGRY`: subsystem error,
- `SLEEPY`: reserved for extended idle behavior.

This emotional abstraction is practical: it gives users immediate context without exposing raw technical variables.

5.4.3 Microphone Awareness in UI

The OLED interface also reflects microphone activity. This is important because a social robot should indicate when it is listening and when audio capture is blocked.

Mic status is driven by runtime events:

- `audio_listening_started` →mic-on indicator,
- `audio_listening_finished`/`tts_started` →mic-off,
- motion or buzzer activation →mic-off to reduce feedback.

5.4.4 Ultrasonic Safety Sensing

The ultrasonic front sensor continuously publishes `distance_cm` events. These values are consumed by multiple modules:

- display state transitions (e.g., `STUCK`),
- buzzer near-obstacle alerts,
- follow-mode correction in motion control.

This message-based fan-out is a strong example of modular system design having one sensor stream feeds multiple behaviors

5.4.5 Buzzer Feedback Patterns

Buzzer signaling complements visual feedback, especially in noisy or bright environments where display reading is less reliable.

Implemented buzzer patterns include:

- near-obstacle warning,
- countdown cue before identity enrollment,
- short acknowledgement after wake detection.

The buzzer thread emits `buzzer_state` messages so the audio subsystem can temporarily mute microphone capture during acoustic output.

5.4.6 Behavior Timing and Human Perception

Small timing parameters have large effect on perceived quality:

- `lonely_after_s` avoids immediate mood flips,
- `stuck_after_s` prevents false stuck alarms,
- greeting delay and minimum-open windows improve conversational flow.

These values were tuned experimentally during iterative testing. The resulting behavior feels less mechanical and more consistent for users.

5.4.7 Summary

Botronka's user experience is built up from coordinated sensor interpretation and multimodal feedback. OLED emotions, buzzer cues, and distance awareness make internal decisions visible to users and improve trust in robot behavior.

Chapter 6

Verification and validation

6.1 Verification

6.1.1 Validation Strategy

Validation was performed at three complementary levels:

1. **Unit tests** for deterministic logic (parsers, policy, state transitions),

2. **Integration checks** for message flow between worker threads,

3. **Field tests** to verify behavior on physical hardware.

This layered strategy was necessary because Botronka is a complex physical system with different electronics so software correctness must be confirmed under real sensor noise, actuators and processing delays, and giving the ability to test each module at a time.

6.1.2 Unit-Level Verification

The project includes focused tests for core runtime behavior:

Table 6.1: Some representative unit test files and intent.

Test file	Validated logic
`tests/unit/test_motion.py`	Parsing of motion commands, steering limits, follow-mode distance behavior
`tests/unit/test_wake_mode.py`	Wake phrase matching and microphone opening constraints
`tests/unit/test_agent_quick_path.py`	Low-latency local NLU shortcuts and response format
`tests/unit/wheels_test.py`	Hardware-oriented wheel control smoke checks

6.1.3 Integration Checks

Integration validation focused on message-chain continuity:

```
audio_utterance →stt_text →agent_reply →tts_request/motion_command
```

Additionally, runtime state synchronization was checked for concurrency events:

- microphone mute during TTS playback,
- microphone mute during motion and buzzer activity,
- face/wake transitions between IDLE and ENGAGED states,
- trust gating before motion command execution.

6.1.4 Performance Observation

The runtime logs include timing markers for VAD, STT, LLM, and TTS stages. These measurements were used to guide practical optimizations:

- fast-path local rules for common requests,
- persistent HTTP session for LLM requests,
- phrase caching in TTS worker,
- bounded wake-candidate queue handling.

The resulting behavior provided acceptable response times for short command interactions, lowering average LLM response time from 15s to 10s.

6.2 Reproducibility

6.2.1 Field Reproducibility Procedure

With this setup, all you need is turn on the power switch. If you want more control the reproducible setup and execution flow is summarized below as the only commands you need:

Listing 6.1: Minimal reproducible setup and run flow.

```
python -m venv .venv
source .venv/bin/activate
pip install -e .

# run tests
python tests/unit/test_motion.py
python tests/unit/test_wake_mode.py

# run robot
python main.py
```

For autonomous startups and control, systemd service scripts are included in `scripts/`.

6.2.2 Observed Outcomes

Practical testing confirmed:

- stable event-driven operation across parallel threads,
- successful trust-gated command behavior,
- successful enrollment and promotion of new people,
- robust fallback handling when individual subsystems fail,
- reproducible startup and operation from documented configuration.

6.3 Summary

Verification demonstrated that Botronka is functionally coherent across modules and runtime interactions, while reproducibility assets make the project transferable for future iterations or other engineers.

Chapter 7

Summary

7.1 Encountered difficulties and problems

7.1.1 Engineering Challenges

Developing Botronka as a full-stack robot required solving problems far beyond my expectations. The most demanding issues were related to waiting for creation/shipment and assembly of hardware, runtime reliability in a headless environment, network issues and recovery, and practical hardware constraints during integration. But that gave me time to prepare and plan a good structure for software deployement.

7.1.2 Headless Network Reliability Challenge

One major challenge was maintaining stable remote access during development and testing. Because the robot was usually operated without monitor/keyboard, SSH or VPN availability were essential for debugging and deployment.

The intended behavior was automatic roaming between:

- home Wi-Fi (WPA2-PSK),
- university `eduroam` (WPA2-Enterprise / PEAP + MSCHAPv2).

When the network failed, Tailscale became offline and remote administration was immediately lost.

`Failure Symptoms and Root Causes :`
During incident analysis, the following symptoms were observed:

- repeated disconnection cycles in NetworkManager logs,
- netplan security warnings about overly accessible configuration,

- missing/empty netplan files in `/etc/netplan/`,
- enterprise re-authentication requiring interactive secrets (not available headlessly).

Root causes were a combination of configuration corruption, strict permission enforcement, and non-interactive authentication constraints.

Table 7.1: Network challenge: root causes and corrective actions.

Issue	**Observed impact**	**Action taken**
Corrupted netplan files	Wi-Fi config became ineffective; loss of connectivity	Recreated clean netplan baseline from known-good settings
Permission policy violations	`netplan generate/apply` warnings and unstable apply behavior	Corrected ownership and restrictive permissions on flagged files
Headless EAP secret prompts	Enterprise Wi-Fi attempts failed without interactive agent	Verified full credential storage and validated with low-level logs
Unstable fallback strategy	Delayed recovery during field tests	Added practical hotspot fallback path for emergency access

7.1.3 Power and Stability Constraints

Another challenge came from mixed-power robotics operation. The platform can run via 5V GPIO power input, but this increases sensitivity to undervoltage under load. Brownouts can indirectly degrade Wi-Fi stability and service continuity, data corruption, or sensor failing.

For now to handle this, only electrical distribution are reviewed after each major hardware modification.

7.1.4 Mechanical Iteration Challenges

The physical platform underwent multiple revisions. Initial layouts were small and basically unpractical for this usage. At first I wanted to go with omniwheels for multidirection movement on a flatplatform. But after few consultations and redesigns with Engineering Students Club, we 3D-printed parts that gave space and control.

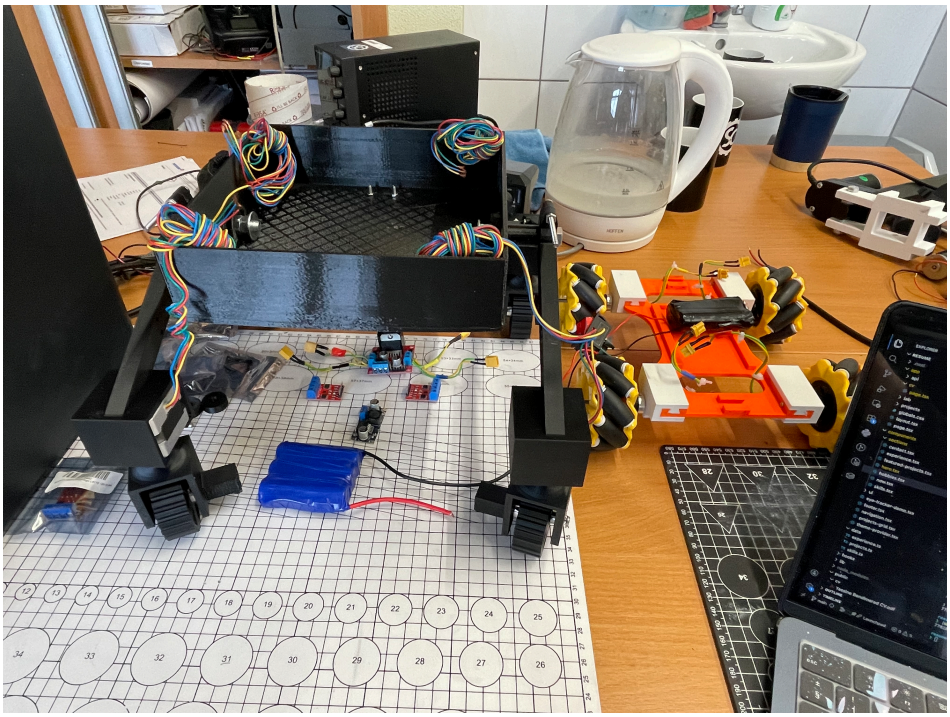


Figure 7.1: Comparison between earlier and improved platform organization.

7.1.5 Software Integration Complexity

The most persistent software challenge was concurrency coordination. Audio capture, TTS playback, movement control, and buzzer output had to be synchronized to avoid race conditions and self-interference. This was addressed through: a shared runtime state store, explicit audio modes, event-driven muting rules, consistent message contracts between threads.

7.1.6 Lessons Learned

The project highlighted practical lessons important in robotics: reliability engineering is as important as AI model selection, headless networking requires strict configuration discipline, clear state synchronization is essential in multimodal systems, always prepare fallback paths (hotspot, dry-run modes) to significantly reduce downtime.

7.1.7 Summary

The difficulties encountered were critical for shaping the final architecture. Solving them improved robustness, maintainability, and confidence in real-world operation.

7.2 Possibilities of further development

7.2.1 Current Risk Landscape

Although the prototype achieves its main objectives, several technical and operational risks remain for long-term deployment.

Table 7.2: Main risks identified in the current system state.

Risk category	**Severity**	**Description**
Power stability	High	Undervoltage can permenantly damage the pi or sensors
Network dependence for remote ops	Medium	Loss of Wi-Fi/VPN disables remote debugging and administration
Speech robustness in noise	Medium	STT quality degrades in outdoor/noisy acoustic conditions
Vision misidentification edge cases	Medium	Similar faces and lighting variation can reduce recognition reliability
Mechanical wear and cable stress	Medium	Mobile operation can degrade connectors and printed parts over time
Runtime contention on edge CPU	High	Concurrent AI tasks can increase latency under peak load

7.2.2 Model and Interaction Improvements

Potential AI-layer improvements include:

1. **Adaptive wake detection**: personalize threshold by environment noise profile.
2. **Context memory**: keep short conversation state for more natural dialogue continuity.
3. **Confidence-aware responses**: ask clarifying questions on uncertain STT/vision outcomes.
4. **Model benchmarking**: compare additional compact STT/LLM variants for latency-accuracy trade-offs.

7.2.3 Perception and Sensing Improvements

Single-front ultrasonic sensing limits obstacle awareness. Future revisions should consider:

- side/rear distance sensors,
- basic visual obstacle cues from camera stream,
- sensor fusion for smoother follow and stop decisions,
- confidence scoring for noisy sensor intervals.

7.2.4 Architecture and DevOps Improvements

The architecture is modular, but maintainability can be improved further by:

- formal message schemas/versioning,
- structured log aggregation and replay,
- simulation mode for CI-based integration tests,

7.2.5 Mechanical and Electrical Roadmap

Hardware-focused future work:

- more robust power regulation and battery telemetry,
- protected wiring channels and connector locking,
- modular service access for rapid maintenance.

7.2.6 Expected Impact of Improvements

Implementing the improvements above is expected to increase operational reliability in non-lab conditions, user trust through predictable behavior, maintainability and extensibility towards autonomous robotics research.

7.3 Conclusions

7.3.1 Thesis Summary

This thesis presented the design and implementation of **Botronka**, an AI-powered mobile robot developed as a complete student engineering project. The work integrated computer vision, speech interaction, trust-aware decision logic, and physical motion control into a single coherent runtime architecture.

The final platform demonstrates that meaningful multimodal interaction is feasible on a limited resources embedded hardware when software architecture is carefully designed around modular threads, message contracts, and explicit runtime state synchronization.

7.3.2 Main Achievements and Engineering Perspective

The key achievements of the project are:

1. end-to-end event-driven architecture spanning sensing, reasoning, and actuation,
2. face identity pipeline with trust-level semantics and dynamic user enrollment,
3. local audio stack (VAD + STT + LLM + TTS) with wake-mode support,
4. trust-gated command execution and policy-based safety filtering,
5. documented and unit tested workflow and reproducible project structure.

An important outcome of this work is that system reliability challenges (networking, power quality, state synchronization) were as significant as AI-model integration itself. Solving these practical challenges transformed the project from a set of isolated demos into an operational robotic system.

The thesis therefore contributes not only implementation details, but also a realistic development methodology: iterate, instrument, validate on hardware, and keep failure-recovery paths available.

7.3.3 Research and Development Outlook

With the following extensions, the platform can evolve from companion prototype toward a more advanced human-aware mobile robotics research testing platform, where Botronka can serve as a strong baseline for:

- improved multimodal context memory,
- richer obstacle perception and sensor fusion,
- stronger command formalization and safety envelopes,
- long-duration autonomy and deployment tooling.

7.3.4 Final Remark

In conclusion, the project fulfills the thesis objective of delivering an *AI-Powered Mobile Robot*. It combines practical hardware integration with modern local AI tooling while providing a clear base for further academic and engineering progress.

Bibliography

[1] Debian Project. *Debian 12 (Bookworm) Documentation*. https://www.debian.org/doc/. Accessed: 2026-02-28. 2025.

[2] Georgi Gerganov and contributors. *llama.cpp: LLM inference in C/C++*. https://github.com/ggml-org/llama.cpp. Accessed: 2026-02-26. 2024.

[3] Georgi Gerganov and contributors. *whisper.cpp: Port of OpenAI Whisper model in C/C++*. https://github.com/ggml-org/whisper.cpp. Accessed: 2026-02-26. 2024.

[4] Google and contributors. *WebRTC Voice Activity Detector (Python bindings)*. https://github.com/wiseman/py-webrtcvad. Accessed: 2026-02-26. 2024.

[5] NetworkManager Developers. *NetworkManager Documentation*. https://networkmanager.dev/docs/. Accessed: 2026-02-28. 2025.

[6] Open Home Foundation Voice Team and contributors. *Piper: Fast local neural text to speech system*. https://github.com/OHF-Voice/piper1-gpl. Accessed: 2026-02-26. 2024.

[7] OpenCV Team. *cv::FaceDetectorYN Class Reference*. https://docs.opencv.org/4.x/df/d20/classcv_1_1FaceDetectorYN.html. Accessed: 2026-02-28. 2025.

[8] OpenCV Team. *cv::FaceRecognizerSF Class Reference*. https://docs.opencv.org/4.x/da/d09/classcv_1_1FaceRecognizerSF.html. Accessed: 2026-02-28. 2025.

[9] OpenCV Team. *OpenCV Zoo: Pre-trained models for OpenCV*. https://github.com/opencv/opencv_zoo. Accessed: 2026-02-26. 2024.

[10] Lennart Poettering and contributors. *systemd Documentation*. https://www.freedesktop.org/software/systemd/man/latest/. Accessed: 2026-02-28. 2025.

[11] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey and Ilya Sutskever. 'Robust Speech Recognition via Large-Scale Weak Supervision'. In: *arXiv preprint arXiv:2212.04356* (2023). URL: https://arxiv.org/abs/2212.04356.

[12] Raspberry Pi Foundation. *Raspberry Pi Documentation*. https://www.raspberrypi.com/documentation/. Accessed: 2026-02-28. 2025.

[13] Raspberry Pi Ltd. *Picamera2 Manual and Examples*. https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf. Accessed: 2026-02-28. 2025.

[14] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale et al. 'Llama 2: Open Foundation and Fine-Tuned Chat Models'. In: *arXiv preprint arXiv:2307.09288* (2023). URL: https://arxiv.org/abs/2307.09288.

Appendix A

Project File Map

This appendix summarizes the most relevant directories and files used to build and evaluate Botronka.

A.1 Top-Level Repository Structure

Listing A.1: Simplified repository layout.

```
botfriend/
 config/
    config.yaml
 data/
    people/
        face_db.json
        trust_map.json
 scripts/
    botfriend.service
    botfriend_boot.sh
    install_boot_service.sh
 src/
    app.py
    agent/
    audio/
    core/
    hardware/
    threads/
    vision/
 tests/
     unit/
     integration/
```

A.2 Core Runtime Files

Table A.1: Main runtime files and purpose.

File	Purpose
`main.py`	Program entry point and configuration loading
`src/app.py`	Composition root for all worker threads
`src/threads/threadManager.py`	Queue dispatch and thread lifecycle control
`src/core/message.py`	Shared message envelope definition
`src/core/state.py`	Thread-safe shared runtime state and audio-mode logic
`config/config.yaml`	Runtime parameters (models, GPIO, behavior, trust)

A.3 Subsystem Implementation Map

Table A.2: Implementation modules by subsystem.

Subsystem	Key files
Vision	`src/threads/vision.py`, `src/vision/face_service.py`
Audio I/O + VAD	`src/threads/audioIO.py`, `src/audio/vad.py`
STT	`src/threads/STTworker.py`, `src/audio/STT.py`
Agent + NLU + Policy	`src/threads/AgentWorker.py`, `src/agent/nlu.py`, `src/agent/policy.py`, `src/agent/llm_client.py`
TTS	`src/threads/TTSworker.py`, `src/audio/TTS.py`
Motion control	`src/threads/motion.py`
Display + buzzer + ultrasonic	`src/threads/display.py`, `src/threads/buzzer.py`, `src/threads/ultrasonic.py`

A.4 Test Files of Interest

- `tests/unit/test_motion.py`
- `tests/unit/test_wake_mode.py`
- `tests/unit/test_agent_quick_path.py`
- `tests/unit/wheels_test.py`

Appendix B

Message Model and Event Contracts

This appendix documents the message-based contracts used for inter-thread communication in Botronka.

B.1 Base Envelope

All events use a common envelope:

Listing B.1: Base message envelope shared by all workers.

```
@dataclass
class Message:
    sender: str
    type: str
    content: str # JSON payload encoded as string
    sent_at: float
```

B.2 Core Event Types

Table B.1 lists the most important message types used in runtime execution.

Table B.1: Key runtime message types and semantics.

Type	Producer	Purpose / payload highlights
`vision_identity`	VisionThread	Face presence, name, trust level, similarity, temporal identity context
`vision_error`	VisionThread	Non-fatal vision runtime exception details
`distance_cm`	UltrasonicFront	Front distance measurement value
`audio_utterance`	AudioIOThread	Path to recorded user utterance WAV

Type	Producer	Purpose / payload highlights
`audio_wake_candidate`	AudioIOThread	Short wake-probe WAV for wake phrase detection
`audio_wake_detected`	STTWorker	Wake phrase accepted, includes wake-open duration
`audio_listening_started`	AudioIOThread	Listening phase start marker
`audio_listening_finished`	AudioIOThread	Listening phase end marker
`stt_text`	STTWorker	Transcribed user text + source wav reference
`stt_error`	STTWorker	STT failure details
`llm_thinking`	AgentWorker	Boolean status indicating LLM processing in progress
`agent_reply`	AgentWorker	Final decision payload (chat/command + spoken text)
`tts_request`	AgentWorker / AudioIOThread	Request to synthesize and play speech
`tts_started`	TTSWorker	TTS playback start (used to mute mic)
`tts_finished`	TTSWorker	TTS playback end (audio flow resumes)
`tts_error`	TTSWorker	TTS synthesis or playback error
`motion_command`	AgentWorker	Motion instruction text for parser/execution
`motion_state`	MotionControlThread	Boolean moving flag for cross-thread coordination
`buzzer_countdown`	AgentWorker / STTWorker	Trigger countdown or chirp pattern
`buzzer_state`	BuzzerThread	Boolean active flag for mic mute coordination
`vision_register_request`	AgentWorker	Request identity enrollment in vision subsystem
`vision_register_result`	VisionThread	Enrollment result (success/error)

B.3 Typical End-to-End Event Sequence

The nominal conversation/action sequence can be summarized as follows:

1. `vision_identity` indicates face presence and trust context.
2. `audio_utterance` is emitted after VAD capture.

3. `stt_text` carries transcribed text.
4. `agent_reply` and `tts_request` are produced.
5. Optional `motion_command` is sent if policy allows execution.
6. `tts_started`/`tts_finished` and `motion_state` coordinate shared runtime behavior.

B.4 Contract Design Notes

The message model follows practical principles:

- human-readable JSON payloads for easy debugging,
- explicit event typing for selective consumption,
- non-fatal error events instead of thread crashes,
- state-bearing messages for synchronization (audio, motion, buzzer).

These contracts made it possible to evolve each subsystem independently while preserving stable integration behavior.

List of Figures

List of Tables