UGANDA CHRISTIAN UNIVERSITY

A Centre of Excellence in the Heart of Africa

UGANDA CHRISTIAN UNIVERSITY

FACULTY OF ENGINEERING, DESIGN AND TECHNOLOGY

Department of Computing and Technology

Roadwise Traffic Management Prototype: System Architecture and Implementation

Group Members

Mokili Promise Pierre	B20848
Geno Owor Joshua	B20233
Mbeiza Rachel	B23293

Contents

1 Abstract

This document details the Roadwise prototype: a low-cost AI and IoT traffic sensing and actuation system that emphasizes cyclist and vulnerable road user priority in mixed-traffic urban environments. The prototype couples a YOLO-based detection pipeline with a lightweight Python backend, microcontroller-based actuation (ESP8266/Arduino), and a minimal interface for operator situational awareness. The report includes a technical architecture, hardware integration details, API formats, control state machine, model invocation patterns, testing methodology, and practical deployment guidance for small pilots.

2 Introduction

Roadwise is a practical demonstration of how commodity sensing and open-source machine learning can be combined to deliver intersection-level situational awareness and simple actuation policies in resource-constrained urban contexts. The prototype targets three immediate goals:

1. Achieve reliable frame-level detection of vehicles and cyclists using YOLO family models.
2. Translate perception outputs into safe, auditable actuation commands via microcontrollers.
3. Provide a minimal operator interface and logging for evaluation and monitoring.

This documentation is written for developers and engineers who will reproduce, extend, or pilot the Roadwise prototype.

3 System Architecture

The Roadwise prototype uses a layered architecture with clear separation between sensing, intelligence, actuation, and interfaces.

- **Layer 1: Sensing & Hardware** – USB UVC camera(s) and optional local sensors (e.g., ultrasonic, IR) capture raw environmental information.
- **Layer 2: Intelligence & Cloud/Edge** – A Python backend (Flask) runs the YOLO detection inference and implements aggregation, simple decision logic, and persistent logging. Optionally this runs on an edge PC or Raspberry Pi; development uses a GPU-backed workstation for model tuning.
- **Layer 3: Action** – Microcontroller firmware (ESP8266/ESP32/Arduino) receives actuation commands (HTTP/MQTT) and toggles relays or LED arrays representing traffic lights.
- **Layer 4: Interfaces** – A lightweight officer dashboard displays recent counts and live snapshots; the backend exposes REST endpoints to support simple integrations.

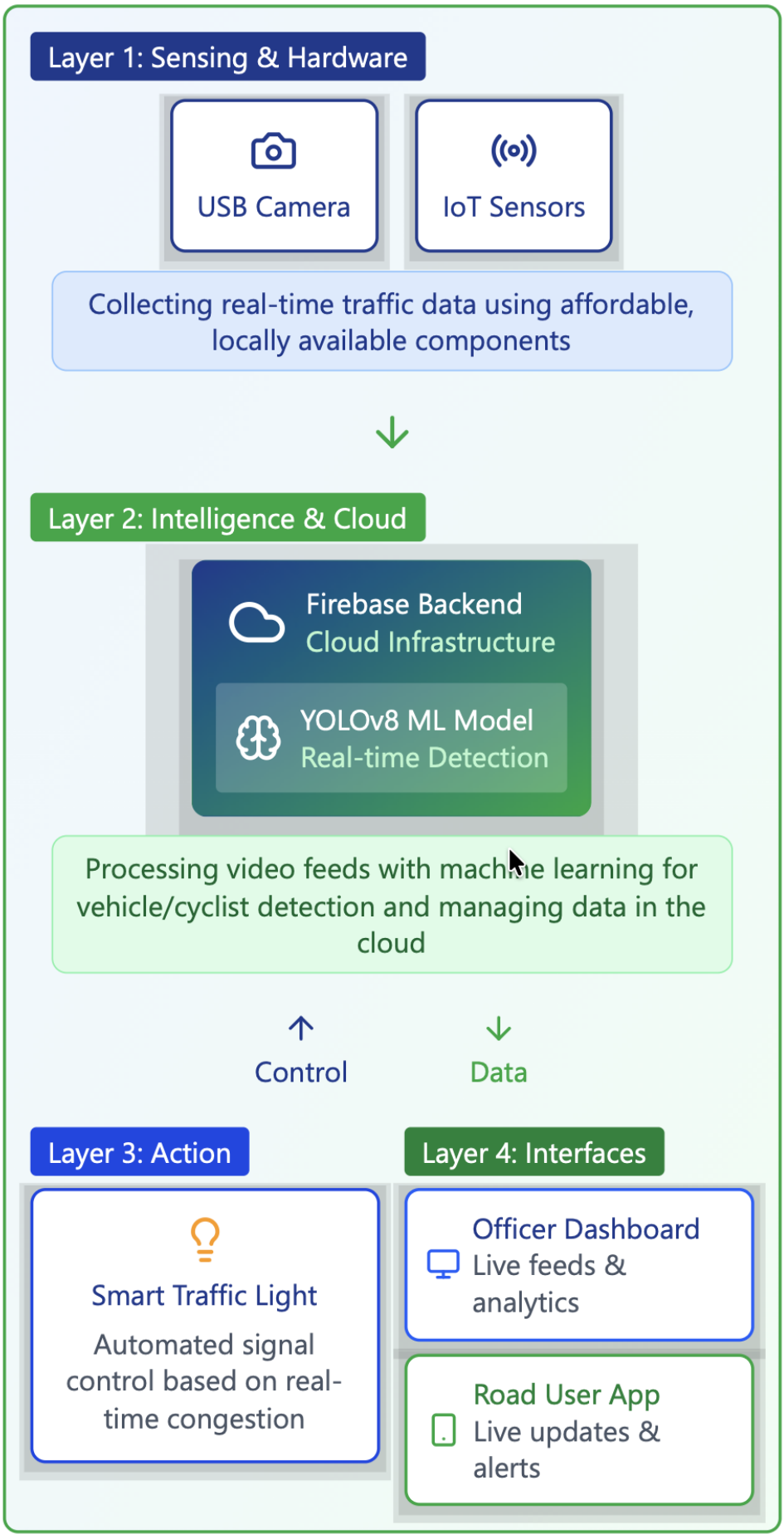

Figure 1: Roadwise prototype system architecture. The image file `roadwise_architecture.png` must be in the project root.

4 Materials and Physical Setup

The prototype emphasizes locally available components and modular integration. The BOM below lists the component types used in the prototype (no prices included).

- USB UVC camera (720p or 1080p) for real-time capture.
- Edge compute device (Raspberry Pi 4 or equivalent) or workstation for model inference and backend hosting.
- Microcontroller: ESP8266 or ESP32 dev board (NodeMCU / Wemos) or Arduino-based board for local actuation.
- LED arrays or a traffic-light style kit for prototyping signal outputs, or relay modules to interface to larger signals.
- Power supplies, wiring harnesses, connectors, and mounting hardware.
- Optional: GPU-equipped workstation for model development and batch inference experiments.

4.1 Physical notes and best practices

- Share ground between microcontrollers and any interfaced host to avoid floating-level issues.
- Use proper relay drivers or optoisolators when switching mains equipment; the prototype uses LEDs for safety during testing.
- Camera mounting should minimize vibration and provide clear sightlines to the detection zone. Use enclosures or weatherproof housings for outdoor runs.

5 Hardware Integration and Pin Mapping

This section maps the microcontroller GPIOs to the prototype outputs. The example pin assignments reflect the Arduino-style sketches in the repository.

5.1 Example pin assignment

Listing 1: Representative pin assignments (adapt to your board)

```
#define RED_N 2
#define YELLOW_N 3
#define GREEN_N 4
#define RED_E 6
#define YELLOW_E 7
#define GREEN_E 8
#define RED_S 10
#define YELLOW_S 11
#define GREEN_S 12
```

Adjust pins to avoid conflicts with serial or I2C pins on your chosen microcontroller.

6 Software Stack and Key Files

Primary repository files and their roles:

- `app.py` – Flask backend and detection orchestrator. Loads YOLO model, handles camera capture, inference loop, aggregation and REST endpoints.
- `esp8266.cpp` – Firmware for ESP8266 that polls backend endpoints and sets GPIO outputs accordingly.

- `main.cpp`, `Test_code.cpp` – Arduino sketches implementing traffic light state machines and local display logic.
- `detection.ipynb`, `count.ipynb` – Notebooks for prototype model evaluation and counting heuristics.
- `linearregressionmodel.ipynb` – Early regression experiment to suggest green times based on arrival rates.
- `requirements.txt` – Python package list for backend and detection environment.
- `roadwise_architecture.png` – architecture figure used above.

6.1 Backend: model invocation pattern

The backend uses the Ultralytics YOLO (v8) wrapper or the appropriate YOLO library. Typical invocation pattern:

Listing 2: YOLO inference call (conceptual)

```
from ultralytics import YOLO
model = YOLO('path/to/yolov8n.pt') # small model for edge use
results = model.predict(source=frame, imgsz=640, conf=0.35, iou=0.45)
# parse results for bounding boxes, classes and confidences
```

7 Data Flow and Runtime Sequence

This section describes the runtime sequence and messaging between the components.

7.1 Sequence diagram (Camera → Backend → MCU → Dashboard)

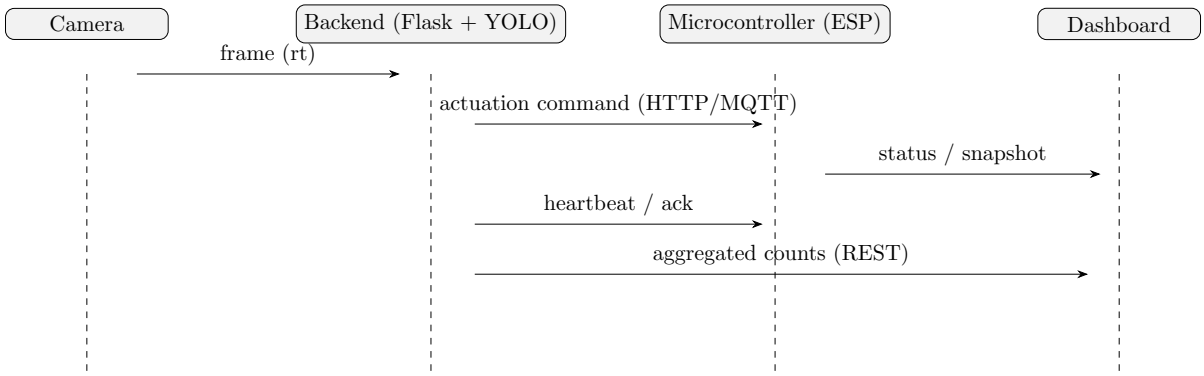

Figure 2: High-level runtime sequence and message flow (simplified).

8 Control Logic and State Machine

The prototype uses a deterministic state machine to ensure safety and predictability. The state transitions are implemented on the controller and coordinated by the backend decisions.

8.1 State machine diagram

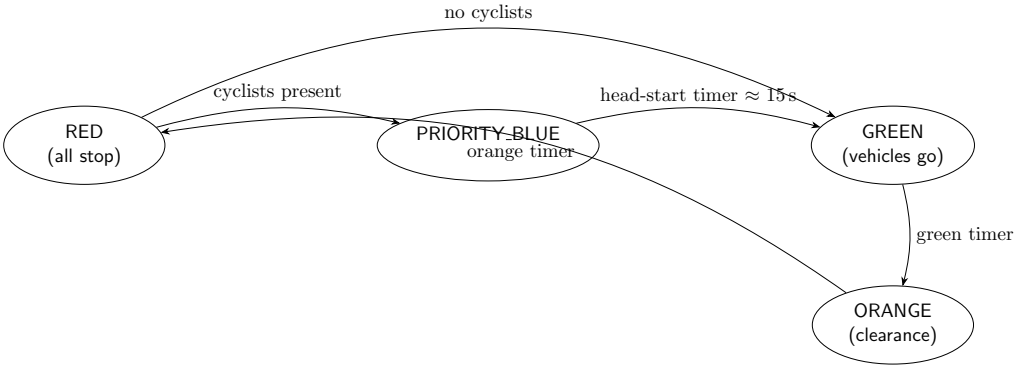

Figure 3: Simplified intersection phase state machine (per approach).

8.2 Decision rules (summary)

- **Cyclist priority**: if cyclist count within window T exceeds threshold X, request a priority blue phase at the next safe gap.
- **Safety fallback**: if connectivity fails or device misses N polls, MCU enters a safe fixed-time cycle.
- **Emergency stop**: backend may send `ALL_RED` to set all approaches to RED immediately.

9 APIs, Message Formats and Examples

All messages use JSON. Below are canonical examples.

9.1 Actuation command (backend → MCU)

```
{
  command: SET_PHASE,
  phase: NORTH_BLUE_PRIORITY,
  duration_s: 8,
  meta: {vehicles:12,cyclists:3,timestamp:2025-05-01T12:34:56Z}
}
```

9.2 Detection response (backend → dashboard)

```
{
  frame_id: 12034,
  vehicles: 12,
  cyclists: 3,
  detections: [
    {class:car,conf:0.86,bbox:[x,y,w,h]},
    {class:bicycle,conf:0.72,bbox:[x,y,w,h]}
  ]
}
```

9.3 Heartbeat (MCU → backend)

```
{device_id:esp_001,status:ok,uptime_s:12345,last_cmd_id:cmd_2025_05_01_1234}
```

10 Model Training, Data and Local Fine-tuning

The included notebooks demonstrate workflow for:

- converting frames and annotations to YOLO format,
- performing data augmentation and synthetic generation for underrepresented classes,
- fine tuning a YOLOv8n model (transfer learning) with a small learning rate for 10–30 epochs,
- evaluating per-class precision and recall to ensure cyclist recall is sufficiently high.

10.1 Recommended training hyperparameters (starting point)

- Model: yolov8n (small) for edge, yolov8s or larger for GPU hosts.
- Image size: 640 px.
- Batch size: depends on GPU memory; for edge training use small batches (4–16).
- Learning rate: 1e−3 to 1e−4 for transfer learning.
- Epochs: 10–30 (monitor for overfitting).
- Augmentation: random scale, horizontal flip, brightness jitter — avoid extreme distortions that break realistic views.

11 Empirical Observations and Performance

During prototype experiments:

- YOLOv8n on Raspberry Pi CPU: **3–6 FPS** at 640 px.
- YOLOv8n on consumer GPU: **20–40 FPS**.
- End-to-end latency (capture → decision → MCU): typically **200–800 ms** depending on hardware and network.
- Cyclist detection recall is the primary challenge due to small object size and occlusion; local retraining significantly improves recall.

12 Testing Methodology

Recommended evaluation protocol:

1. Partition collected data by time slots; ensure test set includes different lighting and traffic scenarios.
2. Evaluate per-class precision, recall and mAP.
3. Measure system-level metrics: detection FPS, end-to-end latency, command success rate.
4. Safety tests: simulate priority grants during heavy cross-traffic to verify no unsafe conflicts are introduced.

13 Deployment and Operational Considerations

- Provide a UPS or battery backup for the edge device and microcontroller to handle short power interruptions.
- Use secure provisioning for WiFi credentials and store secrets out of source control.
- For field pilots, obtain local permissions and post signage indicating video capture for research.
- Log only counts and anonymized metadata when possible to reduce privacy exposure.

14 Security and Reliability Recommendations

- Replace HTTP with HTTPS and use token-based device authentication.
- Implement exponential backoff and local fixed-time fallback logic on the MCU.
- Consider message signing (HMAC) to prevent unauthorized commands.

15 Limitations and Future Work

Limitations:

- Model performance in low light and heavy occlusion requires additional data and potentially hardware changes (IR illuminators).
- The rule-based controller is intentionally simple; it is not a substitute for rigorous traffic engineering design.
- Network reliability and real-time guarantees are not addressed in this prototype.

Future directions:

- Replace polling with MQTT or WebSocket push for lower latency and reduced server load.
- Integrate a micro-simulator (SUMO) to validate phasing strategies before field pilots.
- Implement OTA secure firmware updates and device provisioning portals.
- Collect local Kampala intersection data for tailored model fine-tuning.

16 Repository Mapping and Quick Start

File	Purpose
`app.py`	Backend server, endpoints `/detect-yolo` and `/predict`. Loads YOLO and handles frame loop.
`esp8266.cpp`	Microcontroller firmware; polls backend and applies commands.
`main.cpp`, `Test_code.cpp`	Arduino sketches: state machine and local display.
`detection.ipynb`, `count.ipynb`	Notebooks for model debugging and counting logic.
`linearregressionmodel`	Prototype regression model for timing suggestion.
`requirements.txt`	Python dependencies (use in virtualenv).
`roadwise_architecture.png`	Figure used in documentation.

Quick start commands

```
# Setup Python env (Linux / WSL / macOS)
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt

# Run backend (example)
python app.py --host 0.0.0.0 --port 5000

# Flash ESP8266 using Arduino IDE: open esp8266.cpp, set SSID/PASSWORD, compile and
    upload.
```