Realization Document

Summary

The Smart Traffic Light System is a fully working small-scaled traffic management solution that optimizes traffic flow through real-time sensor data analysis, The system uses ultrasonic sensors to detect vehicle presence across four traffic lanes, processes this data through a logic algorithm and automatically adjust traffic light timing to minimize congestion.

Key Achievements:

- Fully functional **ESP32** system with 4 lanes car detection.
- Complete **backend** with API endpoints for the dashboard and the ESP32.
- Realtime **Database** integration using Firebase.
- **Dashboard** displaying the latest real-time data, traffic status and environmental data.
- Complete **system integration** with bidirectional communication.

1.Implementation & Codebase

1.1 Project Structure

```
Smart-Roads/
├── ESP32/src/main.cpp          (Embedded firmware)
├── backend/
│   ├── server.js               (Express server entry point)
│   ├── config/firebase.js      (Database configuration)
│   ├── controllers/
│   │   ├── sensorController.js  (Data processing and state management)
│   │   └── decisionController.js (Decision logic algorithm)
│   ├── middleware/validation.js  (Validates sensor data before storing)
│   └── routes/
│       ├── sensorRoutes.js     (API endpoints for sensor data)
│       └── decisionRoutes.js   (API endpoints for decisions)
└── dashboard/
    ├── app.py                  (Flask server)
    └── templates/index.html    (Dashboard UI)
```

1.2 Backend Implementation

API Endpoints

Method	Endpoint	Description
POST	/api/sensor-data	Saves sensor data & makes traffic decision
GET	/api/sensor-data	Retrieves sensor history
GET	/api/sensor-data/latest	Gets most recent sensor reading
GET	/api/decision/latest	Gets latest traffic decision
GET	/api/decisions	Gets decision history for dashboard.
GET	/health	Server health check

Decision Logic

The decision logic in **decisionController**.js implements the following priority algorithm:

1. **Filter active lanes:** Only consider lanes with carCount > 0
2. **Sort by priority:** Primary priority is sorted by car count (descending order), Secondary is sorted by first triggered time (ascending order)
3. **Determine traffic density:** Heavy(9 or more cars), Moderate (5 or more), Light (less than 5 cars)
4. **Set green duration:** Fixed 10 seconds for the green light + 3 seconds for the yellow light transition
5. **Cycle management:** Prevent new decision during the active 13 seconds cycle

1.3 Embedded Software (ESP32)

The ESP32 firmware (main.cpp) implements a non-blocking state logic for real-time car detection and traffic light control

Key features:

- **Car counting state logic:** Tracks the transitions between IDLE and CAR_PRESENT. A car is only counted when it leaves the detection zone.
- **Non-Blocking traffic light control:** Handles the timing sequence of the traffic light without blocking the main loop.
- **WIFI Communication:** Sends a JSON to the backend every 2 seconds using a HTTP POST.
- **Environmental Monitoring:** BME280 Sensor for temperature, humidity and pressure.
- **Brownout Protection:** Disabled brownout detector for stable WIFI transmissions.

Pin Configurations

Lane	TRIG Pin	ECHO Pin	LEDs (R/Y/G)
Lane 1	5	34	13 / 12 / 14
Lane 2	19	35	15 / 2 / 4
Lane 3	25	36	18 / 23 / 26
Lane 4	32	39	33 / 27 / 16

2.Installed Software & Dependencies

2.1 Backend Dependencies (package.json)

- Express – Web framework for Node.js, handles the routing and makes setting API easy
- Firebase – Firebase Admin SDK for database access, lets the backend communicate with the database
- Cors – Cross-origin sharing, enables requests between different domain/ports.
- Dotenv – Environment variable management, keeps sensitive information like the firebase credentials out of the code for protection

2.2 ESP32 Dependencies (platformio.ini)

- WiFi.h – Handles connecting to the WIFI
- HTTPClient.h – HTTP request handling, enables the ESP32 to make a HTTP POST to the backend
- ArduinoJson – Handles JSON building and parsing process, ESP32 needs to build JSON (send data) and parse JSON (read logic decision)
- Adafruit_BME280 & Adafruit-Sensor - Libraries for the environmental sensor. Makes reading values simple

2.3 Dashboard Dependencies (requirements.txt)

- Flask – Python web framework, serves the html page and handles the API calls to the backend.
- Requests – Standard HTTP library for API calls, helps to fetch data from the backend API

3.Hardware Implementation

3.1 Components List

Component	Quantity	Purpose
ESP32-WROOM32	1	Main microcontroller with WIFI
HC-SR04 Ultrasonic Sensor	4	Vehicle detection per lane
BME280 Environmental Sensor	1	Temperature, humidity, pressure
LED (Red, Yellow, Green)	12	Traffic light indicators (3 per lane)
Breadboard	2	Component mounting and connections
Resistors (220Ω)	12	LED current limiting
Jumper Wires	-	Component interconnections

3.2 Wiring Configuration

Lane	Ultrasonic Trigger	Ultrasonic Echo	Traffic Light LEDs (R / Y / G)
Lane 1	GPIO 5	GPIO 34	13 / 12 / 14
Lane 2	GPIO 19	GPIO 35	15 / 2 / 4
Lane 3	GPIO 25	GPIO 36	18 / 23 / 26
Lane 4	GPIO 32	GPIO 39	33 / 27 / 16

---

4. Quality Criteria

4.1 Code Quality

- The backend was organized using a **modular structure** with separate controllers, routes, and middleware to keep everything clean and easy to maintain.
- Proper **error handling** was added using try–catch blocks to make the errors easier to interpret and so that the server does not crash unexpectedly.
- All incoming API requests are checked using **validation middleware** to avoid bad or incomplete data.
- Console logs were used throughout the system to help with **debugging** and to **monitor** what the server and ESP32 are doing in real time.

4.2 System Reliability

- The system includes **cycle protection** to make sure the traffic lights never receive conflicting commands.
- If the ESP32 loses WIFI, it automatically tries **reconnecting**, so the system keeps running without manual restarts.

- All HTTP requests have a 5 second **timeout**, which prevents the program from hanging when the server doesn't respond.
- Sensor data and system decisions are **saved** in Firebase, so even if something restarts, the state is not lost.

4.3 Testing Approach

- When the ESP32 starts up, all LEDs run through red/yellow/green to check that the hardware is working.
- I tested the API using a script called test-api.js, which sends different requests to make sure the endpoints are behaving correctly and are up and running.
- I also created simulate-esp32.js, which simulates the ESP32 data so I could test the backend and dashboard without needing the actual hardware.

5.Setup Guide

5.1 Prerequisites

- Node.Js v18+
- Python 3.10+
- PlatformIO IDE
- Firebase account with Realtime Database
- ESP32 DevKit V1
-

5.2 Backend Setup

```
# 1. Clone the repository
git clone https://github.com/AjCodes/Smart-Roads.git
cd Smart-Roads-main/backend

# 2. Install dependencies
npm install

# 3. Configure environment variables
cp .env.example .env
# Edit .env with your Firebase credentials:
# - FIREBASE_PROJECT_ID
# - FIREBASE_PRIVATE_KEY
# - FIREBASE_CLIENT_EMAIL

# 4. Start the server
npm start
# Server runs on http://localhost:5000
```

5.3 ESP32 Setup

```
# 1. Open ESP32 folder in PlatformIO
cd Smart-Roads-main/ESP32

# 2. Edit WiFi credentials in src/main.cpp
# const char* ssid = "YOUR_WIFI_NAME";
# const char* password = "YOUR_WIFI_PASSWORD";

# 3. Update server IP address
# String serverName = "http://YOUR_PC_IP:5000/api/sensor-data";

# 4. Build and upload
pio run --target upload

# 5. Monitor serial output
pio device monitor
```

5.4 Dashboard Setup

```
# 1. Navigate to dashboard folder
cd Smart-Roads-main/dashboard

# 2. Install Python dependencies
pip install -r requirements.txt

# 3. Start Flask server
python app.py
# Dashboard runs on http://localhost:5001
```

5.5 Network configuration

1. Ensure PC and ESP32 are on the same WIFI.
2. Find your PC's ipv4 local IP, using **ipconfig** for windows and **ifconfig** for mac
3. Update ESP32 code with the IP address
4. Allow Node.js through windows firewall on port 5000 (by default)

---

6. Challenges & Solutions

There are always challenges to building something. Here's what we ran into and how we fixed it:

6.1 ESP32 WIFI disconnecting instantly

Problem: The ESP32 disconnected from WIFI immediately every time WIFI initialization started. When WIFI was turned off, the ESP32 worked completely fine.

Why it happened: After adding extra error handling to track where the issue came from, we found out it wasn't a software mistake at all. The WIFI chip on the ESP32 was faulty and kept spiking, which caused the disconnects.

Solution: We replaced the ESP32 with a new one, since the original board had a hardware defect.

6.2 New ESP32 not sending data to backend

Problem: After switching to the new ESP32, the WIFI finally connected normally, but it still couldn't send data to the backend.

Why it happened: After a lot of troubleshooting, we learned that the ESP32 and my backend must be on the same WIFI network. I was on a different network, so the ESP32 couldn't reach the server. Also, I forgot to update the IP address in the code and uploading it after switching networks.

Solution: I connected my laptop to the same WIFI as the ESP32 and updated the backend IP in the code. After that, everything worked smoothly and the backend started receiving data.

6.3 Traffic Lights flickering constantly.

Problem: The lights kept switching every 2 seconds instead of staying green.

Why it happened: The ESP32 sends data every 2 seconds, and the backend kept making new decisions each time. So it kept interrupting itself

Solution: We added a cycle lock so once a lane turns green, it stays green for the full duration before a new decision is made.

6.4 Ultrasonic sensors randomly stopping

Problem: Sometimes the sensors gave 0 or timeout values even though nothing was blocking them.

Why it happened: We powered them through the breadboard rails, and the rails had bad connections/ unstable connections.

Solution: We connected the sensors directly to the ESP32 power pins, so that the power is more stable and reliable.

6.5 Dashboard getting CORS errors

Problem: The dashboard couldn't load any data from the backend. The console kept showing CORS errors, even though the backend worked when I tested it directly.

Why it happened: The backend (port 5000) and the dashboard (port 5001) run on different ports, and when the dashboard tried fetching from the backend, the browser blocked the request since it was from a different port for security reasons unless specified in the code. I didn't realise

this at first, so it looked like the backend was the problem when it was actually a browser security rule.

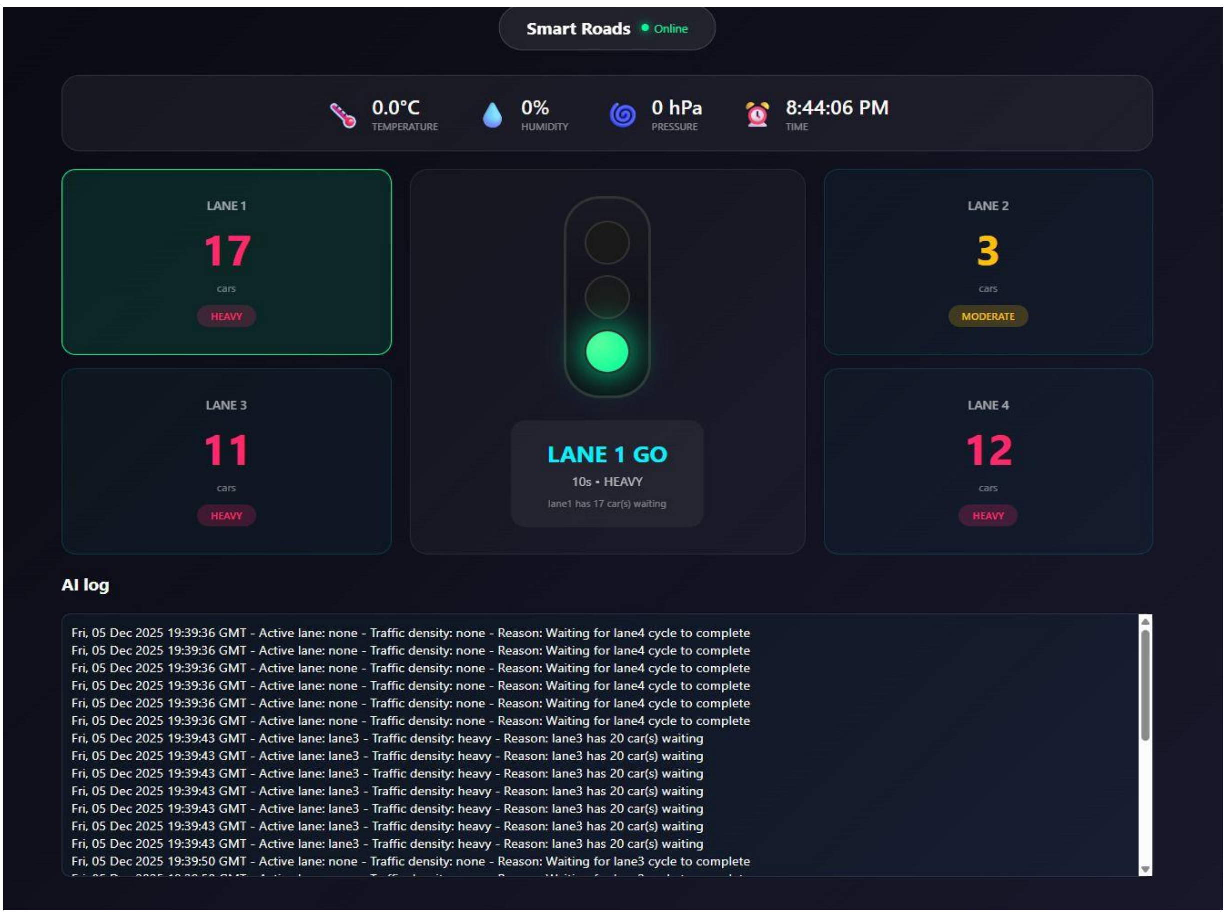
Solution: I enabled CORS in the Express server with app.use(cors()). After that, the dashboard was able to fetch the data normally.

7.Demo Evidence

7.1 Software Demo

The dashboard shows everything happening in real-time. Here's what you can see:

- **Environmental readings:** At the top, temperature, humidity, and pressure coming straight from the BME280 sensor
- **Four lane panels:** Each one displays the current car count
- **Traffic light visualization:** Sits in the center, shows which light is currently active (red, yellow, or green) and for which lane is it active and for how long.
- **System Status:** The "Online" indicator with a green dot lets you know that the ESP32 is connected and sending data.
- **Log Console:** Shows the thinking and decision making of the logic algorithm in which it decides which lanes goes.



(Dashboard screenshot showing the Smart Roads interface)

7.2 Hardware Demo

We recorded a video showing the full system in action. In the demo you can see:

- The ESP32 and sensors mounted on the breadboards
- Hand movements triggering the ultrasonic sensors (simulating cars)
- Car counts updating in real-time
- Traffic lights switching based on the logic decision
- Priority and green light is given to the lane that triggered the sensor first
- The full cycle green light > yellow transition > back to red

The video shows the complete data flow working, from sensor detection all the way to the LED response. (@Smart-Roads Video Demo)

8. Repository & Resources

GitHub Repository**: Smart-Roads** (https://github.com/AjCodes/Smart-Roads.git)

Key Files:

- **Backend**: backend/server.js, backend/controllers/
- **Firmware**: ESP32/src/main.cpp
- **Dashboard**: dashboard/app.py, dashboard/templates/index.html

(The full source code and documentation are available in the repository above)