**IoT Project:**

**RC Car for Surveillance or Data Retrieval**

*University of Cagliari*

*Cristian Manca 70/90/00584, Alberto Perra 70/90/00523*

An RC (Radio-Controlled) Car is a remote-controlled vehicle equipped with devices and sensors to collect useful data in specific contexts. Examples of applications include:

* **Surveillance:** Using cameras to monitor hard-to-reach areas or for security purposes.
* **Data Retrieval:** Collecting environmental data (such as temperature, humidity, gas) or mapping terrain in exploration contexts.

For these purposes, our RC Car will implement a Camera to retrieve real-time image data, enabling an operator to drive the RC Car remotely via the Internet. The control interface communicates with the Camera, allowing the operator to steer the wheels, control the car's speed and direction, turn lights on and off, and manage various onboard sensors, leaving room for additional features.

The system will provide a visual representation of the environment, enable vehicle control, allow for the saving of specific image frames, and support the analysis of those frames using automated object recognition techniques. Additionally, notifications can be sent to users based on the identified objects. This process can also be automated by recognizing specific labels (for instance, if the RC Car detects a dangerous object, it can notify bot subscribers of its presence).

**Functional Requirements**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Description | Input | Output |
| FR1 | Remote Vehicle Control | The system enables an operator to drive the RC Car remotely through the control interface. | User commands via control interface | RC Car movement |
| FR2 | Real-time Image Retrieval | The system retrieves real-time image data using the Camera and displays it on the interface | Camera stream from the RC Car | Real-time video stream on the user interface |
| FR3 | Sensor Management | The system allows monitoring of environmental and operational parameters via onboard sensors. Sensor readings are collected at a high sampling rate and periodically transmitted to the remote system for visualization | Sensor data | Sensor readings are displayed on the control interface |
| FR4 | Data Saving | The operator can save specific image frames for further analysis. | User command to save an image | Image frame saved |
| FR5 | Object Recognition | The system analyzes retrieved images using a machine learning algorithm to identify objects. | Saved image frames | Recognized objects and their labels |
| FR6 | Automated Notifications | The system sends notifications to the subscribers when specific objects are detected | Object labels identified | Notifications sent to the user |
| FR7 | GPS Module Integration | The RC Car provides its current position | Coordinates from the GPS module | Current location displayed in the control interface |

**Non-Functional Requirements**

|  |  |  |
| --- | --- | --- |
| ID | Name | Description |
| NFR1 | System Responsiveness | The control interface must respond to user commands within an acceptable waiting time |
| NFR2 | Battery Efficiency | The system should operate for at least 1 hour on a fully charged battery under standard conditions |
| NFR3 | Scalability | The system must allow integration of additional sensors and features without significant performance degradation |
| NFR4 | Durability | The RC Car must withstand outdoor use, including minor impacts and vibrations |
| NFR5 | Ease of Use | The control interface must be intuitive for operators with no technical expertise |
| NFR6 | Affordable | The device must be cost-effective to attract a wide range of customers |

1. **Scenario Definition**

This project is designed to address the increasing demand for mobile, adaptable, and user-friendly surveillance and data collection systems. Unlike traditional fixed-camera setups or static sensor networks, the radio-controlled (RC) car platform developed here provides a mobile solution that can be easily re-deployed, re-tasked, and adapted to a variety of environments and user needs.

The system’s flexibility stems from its modular hardware and software architecture: producers can customize the onboard detection algorithms or integrate new sensors to tailor the car for specific tasks. For example, in a residential scenario, the platform can patrol a house or garden, using person detection to send real-time alerts if an intruder is spotted. In industrial or commercial contexts, it can monitor facilities for unauthorized access, environmental hazards, or equipment anomalies, using both video and environmental sensors. In the context of wild exploration or scientific research, the car can be programmed to recognize animals instead of humans, enabling non-invasive wildlife observation or ecological data collection in areas difficult or unsafe for humans to access directly.

A key feature of the system is its accessibility. Both control and monitoring are managed via a web dashboard that operates in any modern browser, eliminating the need for specialized apps or technical setup. This makes the solution approachable even for non-technical users, allowing them to pilot the car, monitor the video feed, and review environmental and location data from laptops, tablets, or smartphones. Furthermore, the integration of a notification system extends accessibility: users receive instant notifications about critical detections directly on their mobile devices, ensuring they remain informed and able to act quickly even when away from the main dashboard.

The platform’s programmability and expandability are central to its value proposition. By simply adjusting the detection logic or adding different sensors, the same car can shift from home surveillance to being a tool for wildlife monitoring, or for inspecting changing conditions in industrial environments. GPS integration enables precise localization and path tracking, while environmental sensors provide supplementary data for safety and research applications.

From a technical perspective, the system is built for real-world usability and robustness. The mobile nature of the platform allows it to reach locations that are inaccessible to fixed systems, respond dynamically to changing conditions, and adapt its mission at a moment’s notice. The use of cost-effective, open-source hardware and software ensures that the barrier to entry remains low, making advanced mobile surveillance and data collection accessible to a broad audience.

Overall, this RC car platform enables a new class of surveillance and monitoring scenarios, scalable from home security to industrial and scientific use cases, thanks to its combination of mobility, modularity, programmability, and ease of use.

1. **Assumptions:**

The hardware and software selection process for the RC car surveillance system was shaped not only by functional (FR) and non-functional requirements (NFR), but also by careful measurements and a comparison of technical specifications and real-world integration constraints.

**2.1. Hardware considerations:**

For the main processing and camera unit, we selected an ESP32-CAM module. This board integrates a 2MP OV2640 camera, WiFi (802.11 b/g/n), and a dual-core MCU, delivering all-in-one functionality for real-time video streaming and remote control. During testing, the ESP32-CAM consistently delivered 8–10 FPS at VGA (640x480) resolution, with a measured current draw of 80–180 mA during streaming (see [datasheet](https://randomnerdtutorials.com/esp32-cam-ov2640-camera-settings/)). Peak consumption reached up to 250 mA during WiFi reconnect or camera start, but remained very manageable for mobile battery operation. These values enabled over an hour of continuous operation on a 2200 mAh Li-ion battery or a standard power bank, satisfying the project's autonomy NFR.

Alternatives such as the Raspberry Pi were considered and quickly discarded. While the Pi can support a camera module and offers more CPU power, its board alone draws 400–700 mA (see [Raspberry Pi documentation](https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#power-supply)), not including the additional draw of a camera or WiFi dongle. Such an increase in consumption can lead to a significant reduction in battery life, and the addition of a separate camera module further increases both cost and space requirements, making the solution impractical for our RC car. Moreover, the complexity of managing a full Linux OS, slow boot times, and the risk of SD card corruption after sudden power loss conflicted with our robustness and maintainability goals.

Microcontroller-only boards such as Arduino Uno, Nano, or Pro Mini were also analyzed. While these platforms are popular for sensor integration, they lack the processing power and camera interface necessary for real-time video streaming (FR2), and would require external modules for WiFi and camera functionality, compounding both the power budget and the integration complexity.

With the ESP32-CAM established as the main node, we turned to the challenge of sensor integration. Initially, we attempted direct management of environmental and positional sensors (temperature, humidity, GPS) on the ESP32-CAM, but real-world tests revealed that the camera and networking tasks left insufficient resources for reliable polling and communication with peripherals. This led to sporadic loss of sensor data and, occasionally, dropped video frames, directly impacting FR3 and FR7.

To address this, we evaluated various sensor bridge options. Arduino-based boards (Uno, Nano, Pro Mini) again proved less suitable due to their larger dimensions (e.g., Uno: 68.6 x 53.4 mm). Additionally, the absence of built-in WiFi or convenient UART debugging further complicated integration. In contrast, the NodeMCU (ESP8266-based) board, with its 30 x 58 mm size, offered a compact, USB-powered solution that fit well in the limited space of the RC chassis. Its typical current draw was measured at 70–120 mA (see [NodeMCU datasheet](https://components101.com/development-boards/nodemcu-esp8266-pinout-features-and-datasheet)), and it could be powered directly from a USB power bank, greatly simplifying both wiring and recharging logistics. The ability to program and debug the NodeMCU via USB further enhanced development speed and maintainability, and its resources were more than sufficient for reading sensors and forwarding data over UART to the ESP32-CAM.

The choice of sensors was also informed by technical trade-offs. For temperature and humidity, the DHT11 was selected for its ultra-low current draw (0.3 mA during measurement, 60 μA standby; [DHT11 datasheet](https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT11.pdf)), sufficient accuracy for ambient monitoring (±2°C, ±5% RH), and update rate of 1 Hz. While the DHT22 offers higher accuracy and a wider range, our environmental requirements did not justify the increased cost and current (DHT22: up to 1.5 mA). For GPS, the DIYmall GPS 16E-TT module was adopted, a compact UART GPS with integrated patch antenna, based on a chipset compatible with the u-blox NEO-6M family ([DIYmall GPS 16E-TT reference](https://www.diymalls.com/products/diymall-gps-module-16e-tt)). This module offered reliable UART communication at 9600 baud, low current consumption (45–50 mA during tracking), and sufficient accuracy (~2.5 m CEP) for mobile surveillance. Other modules were considered (e.g., VK-162, or modules with external antennas), but were either too bulky or unnecessarily power-hungry for our use case.

In terms of actuation, the L298N dual H-bridge was selected after practical comparison. The module can supply up to 2A per channel (see [L298N datasheet](https://www.st.com/resource/en/datasheet/l298.pdf)), far exceeding the 0.8A limit of the L9110S. During full-speed movement or under load, our DC motors typically drew 200–400 mA each (stall currents measured above 1A), which the L298N handled without overheating or voltage sag, whereas the L9110S became unstable after prolonged use.

Power supply design was an iterative process. Initial tests with standard 9V alkaline batteries (500 mAh, <1A max) revealed that they could not sustain both logic and motors, especially during acceleration, leading to frequent resets. Supplementing with a USB powerbank (5V 2A) for the logic modules (ESP32-CAM, NodeMCU) provided stable and ample current, and allowed for simple, safe recharging in the field. The hybrid approach, however, requires more space inside the car and double charging (battery replacement and power bank charging) and is not the ideal solution for long-term implementation. A single 12V Li-ion pack, capable of delivering sustained high current and paired with a step-down regulator for the logic rails, would provide a cleaner and more robust solution for future revisions.

The final hardware configuration, with ESP32-CAM for video and control, NodeMCU for sensor bridging, L298N for actuation, DHT11 and DIYmall GPS 16E-TT for environmental and positional data, and a hybrid power supply, was the result of both theoretical requirements mapping and direct measurement through repeated prototyping. Each choice was validated against the project's target of robust real-world operation, energy efficiency, and maintainable system integration.

**2.1. Software considerations:**

A fundamental architectural choice at the outset was the adoption of WiFi as the primary communication method between the ESP32-CAM, the backend server, and the user dashboards. Wireless alternatives like Bluetooth, Zigbee, or LoRa were seriously considered. Bluetooth (both Classic and BLE) was quickly ruled out due to its limited range and especially its insufficient bandwidth for low-latency video streaming (FR2) and real-time remote control (FR1). Zigbee and LoRa, while offering extended range and low power consumption, are optimized for low-data-rate telemetry and cannot support the continuous, high-throughput JPEG video required by our surveillance use case. WiFi, by contrast, is natively supported on the ESP32, provides ample bandwidth for video and control, and is universally compatible with standard infrastructure and user devices, ensuring both high responsiveness and ease of deployment for non-technical users (NFRs).

However, not all communication within the system leverages WiFi. The NodeMCU module, tasked with reading environmental and GPS sensors, does not use WiFi to communicate with the rest of the system. Here, too, several options were evaluated:

* I2C: A widely used protocol for inter-microcontroller and sensor communication, I2C allows multiple devices on a shared bus. However, it is susceptible to noise over longer distances and requires careful addressing and pull-up configuration. In practice, synchronizing master/slave roles between ESP32 and NodeMCU for reliable, bidirectional, and timely sensor data exchange proved error-prone, especially with frequent camera and WiFi activity on the ESP32-CAM.
* WiFi-to-WiFi (two stations): In theory, the NodeMCU could connect via WiFi to the same network as the ESP32-CAM and backend, sending sensor data over TCP/UDP or MQTT. However, this approach would require additional configuration, IP management, and more power consumption, with no real benefit given the data volume and the close physical proximity of the modules. It would also increase the risk of radio interference and complicate debugging.
* UART Serial: A simple, robust, and well-supported protocol for point-to-point communication. UART offers sufficient bandwidth for sensor and GPS data (at 9600 baud or above), is natively available on both MCUs, and is immune to most forms of network contention. In both our lab tests and real-world deployments, UART serial proved to be reliable and insensitive to the heavy network and camera activity on the ESP32-CAM, ensuring sensor readings were delivered on time for both local processing and transmission up to the backend.

Based on these comparisons, UART serial was selected as the optimal solution for connecting NodeMCU and ESP32-CAM. This choice satisfies the NFRs for reliability, simplicity, and power efficiency, while keeping the hardware footprint and firmware complexity minimal.

For the backend, we evaluated several mainstream technologies. Python with Flask was considered early due to its mature ecosystem for scientific computing and AI, its extensive documentation, and its ease of onboarding for new contributors. Node.js with Express and Socket.IO was also examined, offering excellent I/O performance and native support for real-time web protocols. However, integrating high-performance AI (like YOLOv3 object detection) in a Node.js stack would have required either native bindings to C++ or communication with a Python subprocess, introducing complexity and undermining the maintainability and responsiveness goals central to our non-functional requirements. Language like Go, while highly concurrent and efficient, lacked the rapid prototyping and AI library support essential for our development speed and future expansion. C++ was ultimately set aside due to the significantly higher development overhead for web and storage options integration, despite its performance advantages.

Through practical prototyping, Python with Flask emerged as the best choice for our use case. Its threading and async capabilities allowed us to maintain low-latency I/O, while its ecosystem simplified the integration of OpenCV for vision, PyMongo for data storage and access, and requests for the notification system. This directly fulfilled both the system responsiveness NFR and our requirement for maintainability and extensibility.

Choosing the right communication protocol for real-time and bidirectional data exchange between ESP32-CAM, the backend, and the dashboard was another critical area. We explore HTTP polling and REST endpoints, but these quickly showed their limits: command feedback was noticeably delayed, video was choppy, and overall latency was unacceptable for remote driving. MQTT was briefly considered for sensor telemetry, but its lack of native browser support and inefficiency when handling high-rate binary data (like JPEG frames) made it impractical as a unified solution. WebRTC, while state-of-the-art for browser-based video, is simply not viable on ESP32-class hardware due to resource constraints and lack of official libraries. In contrast, WebSocket provided the persistent, low-latency, bidirectional channel we needed. It is natively supported in both modern browsers and in the firmware stack (`WebSocketsClient.h` for Arduino/ESP32), and allowed us to deliver sub-100 ms command latency and smooth video at up to 10 FPS, directly supporting the project's responsiveness and user-friendliness targets.

For object detection, we required a backend solution that was not only accurate and performant but also easy to extend with new models or detection logic. OpenCV’s DNN module with YOLOv3 models was integrated as a result of both its strong performance in CPU-bound scenarios and its straightforward integration with Flask and the rest of our Python codebase. We compared this pipeline to TensorFlow/TFLite, but found that OpenCV/YOLOv3 offered faster deployment and more robust results on typical server hardware. Hardware accelerators (Edge TPU or Jetson) were considered but ruled out due to their cost, energy consumption, and the project's targeted hardware simplicity.

Our requirements for robust, scalable image and metadata storage led us to consider filesystem storage, SQLite, and MongoDB. While direct filesystem access allowed for rapid prototyping, it proved unmanageable as the gallery grew, especially regarding filename collisions and metadata search. SQLite improved metadata handling but struggled with binary data and concurrent access under load. MongoDB paired with GridFS, on the other hand, provided a seamless way to store and retrieve large binary objects alongside rich metadata, supporting both high performance and the ability to run geospatial queries for GPS-tagged frames. This approach also maximized future extensibility for analytics and data mining.

Instant user notifications were essential for user experience. We compared email, SMS, custom mobile app notifications, and Telegram Bot integration. Email and SMS were either too slow or too limited; a custom app would have greatly increased development and maintenance effort. Telegram Bot, by contrast, offered instant, media-rich notifications, required minimal user setup, and integrated easily with our Python backend using the HTTP API. This solution proved both reliable and highly usable during field tests.

On the dashboard/frontend side, we considered using frameworks like React or Vue for building a modern SPA, but the project’s requirements, focused on real-time streaming, responsive control, and broad compatibility, meant that a lightweight solution using vanilla HTML, CSS, and JavaScript was optimal. This decision minimized dependencies, ensured fast load times, and made the system accessible to users with a wide range of devices and browsers, directly addressing our ease-of-use NFR.

1. **Analysis**
   1. **Analysis of Non-Functional Requirement:**

The implementation of each functional requirement (FR) in this system is guided by a set of cross-cutting Non-Functional Requirements (NFRs). This section describes each NFR, taking into consideration the specific architectural choices and components adopted to satisfy it.

* + 1. **NFR1 - System Responsiveness**

To guarantee an immediate and interactive experience for the operator, the system has been designed with a strong focus on real-time responsiveness across all its main communication channels. The video streaming and control plane rely on a persistent WebSocket-based client/server architecture, which allows for bi-directional, asynchronous communication with minimal overhead; this enables live video and control commands to be delivered in near real-time, far outperforming traditional HTTP polling or stateless REST APIs. For the aggregation of sensor data, the NodeMCU microcontroller bridges environmental and GPS sensor readings, continuously updating values in memory and periodically transmitting them via a local UART serial connection to the ESP32-CAM. This ensures that sensor updates are always fresh when delivered over the WebSocket channel to the backend and dashboard.

The notification subsystem for object detection events adopts a different architectural pattern: here, the backend acts as a publisher, asynchronously notifying all registered Telegram users (subscribers) of critical detections such as a person in the camera view. This publish/subscribe approach, built atop the Telegram Bot API, ensures that alerts are delivered instantly to multiple users on their mobile devices, even if they are not actively watching the dashboard, without blocking the flow of real-time video or sensor data.

* + 1. **NFR2 - Battery Efficiency**

Maximizing battery life is essential to ensure the system remains operational for at least one hour per charge, even during intensive use. To meet this requirement, computationally demanding tasks such as object detection, image storage, and notification logic are offloaded from the embedded ESP32-CAM to the backend server. This division of labor allows the ESP32-CAM to focus solely on capturing and transmitting video frames, relaying commands, and forwarding sensor data, all of which are optimized for energy efficiency. Sensor readings are processed and transmitted only at necessary intervals, with the NodeMCU handling local aggregation to avoid redundant communication. Firmware routines on both microcontrollers are streamlined to minimize idle power draw, and the system architecture supports flexible adjustment of sampling and transmission rates to balance performance and autonomy according to mission needs.

* + 1. **NFR3 - Scalability**

A key objective is to future-proof the system, allowing for easy integration of additional sensors, new AI modules, or multi-vehicle support without major reengineering. To support this, the hardware and software are structured in a modular fashion: the NodeMCU acts as a bridge for all environmental and GPS sensors, decoupling sensor logic from the ESP32-CAM. This means new sensors or protocols can be added by simply updating the NodeMCU firmware, with no impact on the main video/control logic. On the backend, the architecture is designed to manage multiple sessions and users, and MongoDB/GridFS offers scalable, concurrent-safe storage for large datasets. Such modularity and scalability ensure that the project can evolve and expand with minimal friction.

* + 1. **NFR4 - Durability**

Durability is key for a surveillance vehicle intended for outdoor use. The system’s hardware choices and mechanical design have been selected to withstand typical outdoor conditions, including minor impacts and vibrations. The chassis and mounting for the ESP32-CAM, NodeMCU, and sensor modules are reinforced to provide physical protection, while cable management minimizes the risk of accidental disconnection during operation. Connectors and electronic components are chosen for their robustness and, where possible, for resistance to dust and moisture. Power distribution is protected against short circuits and voltage spikes, and software watchdogs monitor for system hangs or communication errors, triggering safe recovery or shutdown procedures if needed. These combined measures ensure the RC car can operate reliably in real-world, uncontrolled environments.

* + 1. **NFR5 - Ease of Use**

Ease of use has been a decisive factor in all interface and system integration choices. The primary control and monitoring interface for the RC car is a modern web application, accessible from any device with a browser, be it a laptop, tablet, or smartphone, without requiring the installation of proprietary software or complex configuration. This approach drastically lowers the entry barrier for non-technical operators, as they can simply connect to the car’s dashboard through a familiar web environment. The dashboard aggregates all critical functionalities, vehicle control, live video streaming, sensor monitoring, gallery management, and effect toggling into an intuitive and single-page interface, employing clear icons, straightforward controls, and real-time feedback to guide the user at every step.

Moreover, the integration with Telegram for notifications further enhances usability. Telegram is a widely adopted, cross-platform messaging service that most users already have on their smartphones. By leveraging Telegram’s Bot API, the system can deliver instant alerts about critical detections directly to the operator’s phone, even when they are away from the dashboard. This ensures operators remain informed and able to act promptly, without the need to learn new tools or constantly monitor the web interface. Both the web app and Telegram channel were selected specifically because they require no technical setup, are accessible from virtually any device, and present a minimal learning curve, making the overall system truly user-friendly for anyone, regardless of technical background.

* + 1. **NFR6 - Affordability**

Cost-effectiveness has guided every component and technology selection, to keep the overall device affordable and accessible to a wide range of users. The ESP32-CAM and NodeMCU were chosen for their excellent price-performance ratio and broad community support, enabling advanced features such as video streaming and wireless communication without the expense of more specialized hardware. The backend is built on open-source software (Flask, MongoDB, OpenCV), eliminating licensing costs and allowing for future customization without additional investment. By leveraging widely available sensors and employing efficient design patterns, the system delivers professional-grade surveillance capabilities at a fraction of the cost of commercial solutions, making it attractive for both educational and practical deployments.

* + 1. **Security and Access Protection**

Although security was not originally listed among the main non-functional requirements, the preliminary analysis and the definition of the usage scenario made it clear that protecting access to the system is essential. Since the platform allows remote control of a surveillance vehicle and access to sensitive data such as live video, sensor readings, and GPS location, it is crucial to ensure that only authorized operators can use the system.

For this reason, a user authentication mechanism has been implemented: access to the dashboard and all critical functions is protected by a login system with a single user account. This choice intentionally avoids multi-user management, ensuring there is always and only one set of credentials active at any time. The unique user model prevents conflicts and race conditions in vehicle control, guaranteeing that all commands and actions originate from a single authenticated operator.

In line with the intended usage model, it is assumed that each device is shipped or delivered with its unique default credentials, which are different for each RC car. These credentials are provided upon purchase or deployment and are intended to be changed by the user after first access for improved security. Credentials can be updated securely (e.g., to change the default password), but it is not possible to create multiple accounts or assign simultaneous roles. All web and API sessions require authentication, and credentials are stored hashed according to best practices. This approach, while simple, is well aligned with the typical use cases of the system and ensures that access and control remain strictly reserved to the legitimate operator, eliminating ambiguity and increasing overall system safety.

* 1. **Analysis of Functional Requirement:**

This section analyzes each Functional Requirement for the Radio-Controlled Car and discusses the rationale behind the implemented hardware and software solutions. Each Functional Requirement is analyzed concerning the Non-Functional Requirements of the project.

* + 1. **FR1 – Remote Vehicle Control**

The ability to control the RC Car remotely forms one of the principal foundations of the entire project. This function enables an operator to drive the car from any location within WiFi reach, sending commands for steering and direction through a user-friendly interface.

* **Hardware Analysis and Choice:**

For implementing remote control, the central hardware component selected is the ESP32-CAM board. This board integrates a WiFi-enabled microcontroller and a camera, making it exceptionally suitable for IoT robotics where both connectivity and visual feedback are required. The ESP32-CAM was chosen for several reasons. First, it offers a low cost and reduced power consumption. Second, its compact size fits easily within the RC chassis, and its sufficient processing power allows it to handle basic control logic and camera streaming simultaneously.

To actuate movement, a PonteH L298N dual H-bridge motor driver was chosen to interface between the ESP32-CAM and the four DC motors. It manages four standard DC motors (two per side, for differential drive), enabling the car to move forward, backward, and turn. The L298N was selected primarily for its robustness and simplicity; while more efficient and modern drivers exist, their added cost and marginal gains were not justified in the context of our prototype. The four motors, each with an attached wheel, are controlled in pairs to enable the car to move forward, backward, and turn using differential steering. The L298N board accepts a wide range of supply voltages (up to 12V), and in our setup, it was powered by a dedicated 9V battery pack. This battery was chosen for ease of integration and availability, but it was found to be insufficient to reliably power both the motors and the ESP32-CAM simultaneously.

To address this, the power architecture was split: the L298N and the motors were powered exclusively by the 9V battery pack, The L298N and the DC motors are powered exclusively by the 9V battery pack, while the ESP32-CAM and the NodeMCU (ESP8266) are powered by a common USB power bank (5V), which offers a stable and reliable voltage source for the logic components.

In our design, the NodeMCU is directly connected to the power bank and to the breadboard. The ESP32-CAM does not use its onboard micro-USB port for power—doing so would block access to its GPIO pins and make sensor connections impractical. Instead, the ESP32-CAM receives regulated 5V power from the breadboard, which is itself supplied by the NodeMCU’s VV, effectively distributing the power from the power bank to both boards.

The NodeMCU, as we can see well in the following, serves a dual purpose: it acts as a power distributor for the ESP32-CAM and as the main interface for environmental sensors (such as the DHT11 and GPS module). Sensor readings are collected by the NodeMCU and sent to the ESP32-CAM via UART.

* **Communication Technology Analysis (Software Analysis and Choice):**

The implementation of remote vehicle control requires a communication protocol that ensures low latency and reliability between the operator’s web interface and the onboard hardware. In our system, this communication involves three main components: the browser-based frontend, the Python Flask backend (which also hosts a WebSocket server), and the ESP32-CAM microcontroller mounted on the vehicle.

Initially, a RESTful HTTP approach was considered for transmitting movement commands from the web interface to the backend and then to the ESP32. While this method is simple and widely supported, it introduces non-negligible latency and is not suitable for continuous or real-time applications due to its stateless nature, making it incompatible with other requirements. To address these limitations and to guarantee a responsive and robust control channel, the final system architecture adopts a WebSocket-based protocol for real-time communication between the frontend and backend. The user interface establishes a persistent WebSocket connection to the backend, through which commands such as steering, acceleration, and stopping are transmitted instantaneously. The backend then relays these commands to the ESP32-CAM, either via a persistent WebSocket connection or, in scenarios where that is not available, by updating a shared state variable that the ESP32 polls through HTTP requests. This means that, while WebSocket is the primary channel for both frontend-backend and backend-ESP32 communication, the HTTP-based mechanism is retained as a fallback for emergency or degraded conditions.

The adoption of WebSocket as the main protocol is not determined solely by the requirements of the remote control functionality, but is also the result of a broader architectural choice. Real-time video streaming (FR2), sensor data acquisition (FR3), and advanced features all benefit from a unified, bi-directional, low-latency channel. Protocols such as MQTT were also evaluated during the design phase, but their additional infrastructure and reduced browser compatibility made them less suitable for seamless integration in this context.

Consequently, the communication flows for remote control are handled by the web frontend, sending commands via WebSocket to the backend, which then forwards them to the vehicle, where the ESP32-CAM executes the received commands. This design ensures that the system can maintain control even in the event of network disruptions or partial system failures, while maintaining a unified and efficient communication infrastructure across all functional requirements.

* **User Interface Analysis:**

A web-based dashboard developed using HTML, CSS, and JavaScript serves as the main user interface for remote vehicle control. This interface is accessible from any device with a modern browser, including PCs, tablets, and smartphones, without the need for installing additional software. The design focuses on clarity and ease of use, presenting directional controls and real-time feedback in an intuitive layout.

The choice of a browser-based interface offers several advantages. It ensures cross-platform compatibility and enables rapid deployment and updates, as all logic and visuals are centralized on the server. Compared to native mobile applications or physical remotes, the web dashboard lowers the entry barrier for operators and supports seamless integration of additional features.

To further improve the operator’s comfort and responsiveness, the dashboard supports keyboard controls: the RC Car can be driven using the computer’s arrow keys, providing a fast and intuitive alternative to on-screen buttons.

* + 1. **FR2 - Real-time Image Retrieval**

Real-time image acquisition is essential for effective remote operation and situational awareness. The system should stream live video from the onboard camera directly to the web interface, allowing the operator to monitor the car’s surroundings in real time.

* **Hardware Analysis and Choice**

The real-time image acquisition system is based on a standard ESP32-CAM module, chosen for its integrated camera interface, onboard JPEG encoding, and built-in WiFi connectivity. The ESP32-CAM is capable of capturing high-resolution images and supports hardware JPEG compression, making it suitable for streaming applications. The module’s compact form factor and low power consumption make it ideal for mobile and battery-powered robotics applications. The camera is directly interfaced with the ESP32’s native pins, and the microcontroller autonomously manages image capture, encoding, and network transmission. This hardware configuration ensures that the RC Car can provide a continuous video stream without the need for additional processing boards.

* **Communication Technology Analysis (Software Analysis and Choice)**

The implementation of real-time video streaming in this system is entirely based on native WebSocket communication, optimized for low latency and robust operation. The main components involved are the ESP32-CAM microcontroller, the Flask-based backend (with an integrated WebSocket server), and the browser-based web dashboard.

The ESP32-CAM establishes a persistent WebSocket connection with the backend server at startup. Video frames are captured, encoded as JPEG images, base64-encoded, and sent as JSON messages directly over this WebSocket channel. This approach eliminates the need for traditional HTTP POST or polling mechanisms, reducing both transmission latency and server load. The persistent connection allows the ESP32 to push frames as soon as they are available, ensuring smooth and timely delivery even under variable network conditions.

Upon receiving each frame, the backend decodes and processes the image, applies any requested visual effects (such as negative filtering or object detection overlays), and immediately broadcasts the processed frame to the connected web client via WebSocket connection. The web dashboard receives the frames in real time, decoding and displaying them on a canvas element without the need for plugins or refresh cycles.

The choice of WebSocket for both device-to-backend and backend-to-UI streaming was motivated by the need for a unified, bi-directional, and low-overhead communication channel. Unlike HTTP-based MJPEG streaming or RESTful frame uploads, the WebSocket approach guarantees lower latency, supports efficient binary data transfer, and enables the implementation of additional interactive features such as real-time control commands, sensor data updates, and event notifications over the same channel.

Alternative protocols such as RTSP, WebRTC, or HTTP polling were considered but discarded. RTSP and WebRTC are either too resource-intensive for the ESP32-CAM or require more complex solutions, while HTTP polling or MJPEG endpoints introduce unnecessary latency and are less efficient in continuous streaming scenarios.

This WebSocket-based architecture provides a robust foundation for future extensions, such as multi-user support, advanced analytics, or integration with additional sensors, all while maintaining the highest possible real-time performance for video streaming on resource-constrained hardware.

* **User Interface Analysis**

The user interface displays the live video stream in real-time directly within the dashboard on the browser. The interface automatically updates the image as new frames arrive, with no need for manual refresh or external plugins, providing continuous streaming. Additional controls allow the operator to apply real-time effects (such as negative inversion or object detection overlays) to the video stream and to save selected frames for later analysis. The design prioritizes immediate feedback and ease of use, ensuring that even non-technical users can effectively monitor and control the vehicle in real time.

* + 1. **FR3 - Sensor Management**

Integrated environmental sensors provide continuous monitoring of parameters such as temperature, humidity, and additional measurements. These readings are automatically collected and displayed on the control dashboard, enriching the operator’s understanding of the environment.

* **Hardware Analysis and Choice**

The architecture for sensor management is deliberately designed to maximize efficiency and scalability. The ESP32-CAM module is responsible for image streaming and vehicle control, while a separate NodeMCU board is dedicated to sensor integration and power distribution.

Direct connection of sensors to the ESP32-CAM was evaluated but ultimately deemed unfeasible for two main reasons. First, the ESP32-CAM’s available GPIO pins are almost entirely consumed by the camera interface and the dual H-bridge (L298N) required for motor control, leaving insufficient pins for reliable sensor expansion.

Instead, the NodeMCU is employed as a dedicated sensor hub. This board offers a greater number of accessible GPIOs, making it easy to connect and manage multiple sensors (such as temperature, humidity, GPS, and ultrasound), and provides a scalable foundation for future sensor expansion. The NodeMCU communicates sensor readings to the ESP32-CAM over UART serial, which then attaches this data as custom headers to each image frame sent to the Flask backend.

This separation of responsibilities not only ensures the robustness and responsiveness of the video streaming pipeline by allowing the ESP32-CAM to focus solely on image capture and network communication but also simplifies the integration of additional sensors. Furthermore, the NodeMCU helps manage the power supply for the sensors, increasing the system’s reliability during extended operation.

* **Communication Technology Analysis (Software Analysis and Choice)**

The system allows continuous monitoring of environmental and operational parameters via onboard sensors. Sensor readings, such as temperature, humidity, and GPS coordinates, are updated at a high sampling rate on the NodeMCU microcontroller, which serves as a dedicated sensor bridge. The NodeMCU continuously listens to the GPS module and reads environmental sensors, maintaining the most up-to-date values in memory. At fixed intervals (every 2 seconds by default, but configurable in the firmware), the NodeMCU transmits the latest set of sensor readings to the ESP32-CAM over a UART serial connection.

On the ESP32-CAM, the firmware parses the incoming serial data stream in real time, extracting and structuring the sensor values. Each set of readings is immediately packaged into a JSON-formatted message, which is then sent to the backend server using the same persistent WebSocket channel employed for both video streaming and control messages.

The backend Flask application receives, processes, and buffers incoming sensor updates, continuously maintaining the most recent environmental and GPS data in its global state. These live sensor values are then broadcast in real time to the web dashboard, enabling operators to visualize the current operational context alongside the video stream.

The choice to use a serial UART connection between NodeMCU and ESP32-CAM is motivated by its simplicity, low overhead, and reliability for short-range, high-rate sensor updates. This avoids the complexity and resource requirements of integrating additional wireless modules or implementing network-based IoT protocols at the sensor level.

Alternative solutions, such as connecting sensors directly to the ESP32-CAM or publishing readings via IoT protocols like MQTT, were considered but found less adaptable for modular system upgrades. The current approach allows new sensors to be integrated or sensor logic to be updated simply by modifying the NodeMCU firmware and the corresponding dashboard logic, with no changes required to the main communication pathway. This modular bridge design supports future expansion, such as adding new sensor types or preprocessing methods, without disrupting the rest of the system, thereby maximizing flexibility and maintainability.

* **User Interface Analysis**

The web dashboard presents real-time sensor readings in a dedicated status card, giving operators continuous feedback on the vehicle’s environmental conditions.

* Temperature and humidity values are displayed with clear icons and units, updated every few seconds via AJAX calls to the Flask backend.
* If available, additional sensor data as GPS position, can be shown similarly, allowing the operator to monitor safety-relevant parameters at a glance.
* The UI design ensures that these readings are always visible and easy to interpret, reducing the risk of missing critical environmental changes.

This approach enables non-technical users to benefit from advanced sensor capabilities with no need for specialized tools or software. It supports future expansion (e.g., sensor history, alerts) with minimal changes to the interface or API.

* + 1. **FR4 - Data Saving**

The ability to save specific image frames is crucial for documentation and further analysis. Operators can capture and store frames of interest from the live stream with a single command, ensuring important visual information is retained.

* **Hardware Analysis and Choice**

The data saving functionality is primarily handled on the software side, with the hardware architecture remaining unchanged from the real-time streaming setup. The ESP32-CAM continues to act as the video source, capturing JPEG frames and transmitting them to the backend server. No additional hardware components are required for saving image frames, as the process leverages the same camera and wireless infrastructure already in place for video streaming. This design choice ensures that the data saving feature does not impact the real-time performance or require changes to the RC Car’s physical setup.

* **Communication Technology Analysis (Software Analysis and Choice)**

The system provides the ability to save specific video frames for later analysis, archiving, or notification purposes. This feature is implemented by allowing the user, via the web dashboard, to trigger the capture and storage of the currently displayed frame from the live video stream. When a save request is initiated, the backend retrieves the most recent frame received from the ESP32-CAM, applies any active visual effects (such as negative filtering or object detection overlays), and then commits the processed image to persistent storage.

Frame saving is handled through integration with MongoDB’s GridFS, a scalable file storage system designed for efficiently storing and retrieving large binary files along with rich metadata. Each saved image is stored as a binary object in GridFS and is accompanied by a metadata record containing contextual information such as the timestamp, GPS coordinates, environmental sensor readings, and operator-provided tags or descriptions. Additionally, a thumbnail version of each image is generated and stored for efficient gallery browsing.

This approach ensures robust, concurrent-safe, and scalable storage of images, avoiding filename collisions and supporting quick retrieval or search based on metadata. The choice of GridFS over traditional filesystem storage was motivated by the need for atomic operations, consistent metadata management, and the ability to handle potentially large numbers of images from multiple sessions. The system’s API enables users to view, annotate, classify, or delete saved frames directly from the dashboard interface, and to send selected images via Telegram for instant notification.

By tightly integrating frame capture, effect processing, and metadata management, the system guarantees that each saved image accurately reflects the operational context at the moment of capture.

* **User Interface Analysis**

The web interface includes a dedicated control allowing users to save the current frame with a single button press or, for increased usability, via the keyboard shortcut 'C'. When the save command is triggered, the interface provides immediate feedback on the operation’s status (success, error, or progress) and displays the saved image in a gallery view for later review. Each saved image is enriched with metadata such as the time of capture, location, sensor data, and any effects applied, all visible from the gallery. Users can view, edit metadata by adding a description, or delete saved images directly from the dashboard, providing a comprehensive and user-friendly experience for managing surveillance data.

* + 1. **FR5 - Object Recognition**

Advanced object recognition capabilities enhance the system’s intelligence by automatically analyzing current and saved images for specific objects or threats. This feature leverages machine learning algorithms to identify and label items of interest within captured frames.

* **Hardware Analysis and Choice**

The object recognition feature leverages the same ESP32-CAM module for image acquisition as in previous requirements, ensuring hardware reuse and system simplicity. All computationally intensive machine learning tasks are offloaded to the backend server, which is typically a PC or single-board computer with sufficient processing power (CPU and RAM) to run YOLOv3-based object detection in real time. This design choice avoids overloading the ESP32-CAM, which is optimized for video capture and transmission, not for neural network inference. No additional onboard hardware is required, making the system cost-effective and easier to maintain.

* **Communication Technology Analysis (Software Analysis and Choice)**

The system enables automatic object recognition on the live video stream, leveraging a YOLOv3-based machine learning model for real-time detection. This feature is implemented entirely on the backend server, where each incoming video frame received from the ESP32-CAM is decoded and selectively passed to the detection pipeline, depending on the current system state and performance constraints.

Upon activation of object detection, the backend offloads frames from the live stream to a dedicated detection worker thread. To ensure real-time responsiveness and efficient resource usage, the system applies frame skipping, processing only a subset of incoming frames. Each selected frame is analyzed using OpenCV’s DNN module and the YOLOv3 model, yielding bounding boxes, class labels, and confidence scores for detected objects.

Detection results, including annotated images and structured detection data (bounding boxes, labels, confidences), are immediately displayed to web clients via WebSocket. This enables the dashboard to display detection overlays and relevant metadata in real time, synchronized with the live stream.

The decision to perform detection on the backend, rather than on the ESP32-CAM, is driven by the ESP32’s limited computational resources and memory. Running the YOLOv3 model on the backend enables support for complex models and rapid updates or improvements in the detection logic, without requiring changes to the embedded firmware. The modular design also enables on-demand or batch processing of saved images, supporting both real-time and retrospective analysis.

Alternatives such as edge-based detection (running directly on the camera) or lighter-weight models were considered, but would significantly constrain detection accuracy or scalability. The current architecture provides a robust, scalable, and easily upgradable solution, ensuring high detection accuracy and seamless integration with the system’s notification and data management features.

* **User Interface Analysis**

The web interface provides a simple toggle to activate or deactivate real-time object detection. When enabled, the dashboard overlays bounding boxes and labels onto the live video stream, using distinct colors to highlight dangerous or relevant objects (e.g., “person” in red). Detection results are updated in real time, with clear visual feedback and statistics (e.g., number of detected objects, processing time). Users can also view detection results for saved images in the gallery, including details such as object type, confidence score, and detection timestamp. Furthermore, from the gallery, the user can simply request the classification of the saved image with a button.

* + 1. **FR6 - Automated Notifications**

Automated notification functionality ensures that operators are promptly informed when potentially dangerous objects are detected. The system sends real-time alerts to subscribed users via Telegram, increasing responsiveness and safety.

* **Hardware Analysis and Choice**

The automated notification system does not require any additional hardware on the RC Car. The existing setup, comprising the ESP32-CAM for video streaming and NodeMCU for sensor management, remains unchanged. All notification logic is implemented on the backend server, which processes image frames and object detection results. The only requirement is a network connection between the backend and the internet to allow for communication with the Telegram Bot API.

* **Communication Technology Analysis (Software Analysis and Choice)**

The system provides automatic notification functionality by integrating with the Telegram messaging platform with a publisher and subscriber approach. When the backend’s object detection module (YOLOv3) identifies a "dangerous" object (such as a "person)", in the video stream, a dedicated notification logic is triggered. Detected objects and their metadata are evaluated for relevance and confidence; if a detection meets the configured criteria (e.g., label is "person" and confidence > 60%), the event is immediately broadcast to all registered Telegram users.

This is accomplished by serializing both the annotated image (with detection overlays) and contextual data (confidence, GPS coordinates) and sending them via the Telegram Bot API. The backend supports a notification cooldown mechanism to avoid spamming users with repeated alerts for the same object class within a short period. The list of recipients is managed dynamically: users can register and unregister their chat IDs using simple Telegram commands.

Notifications can be triggered both automatically by real-time detection events and manually by users via the web dashboard (for instance, to share saved images from the gallery). All notification logic, including image formatting, cooldown enforcement, and error handling, is managed on the backend, ensuring reliability and scalability.

This approach was chosen for its immediacy, ease of integration, and broad user accessibility. Alternatives such as SMS or email were considered, but would either increase latency, cost, or require complex setup. By leveraging Telegram’s robust API and cross-platform support, the system ensures instant, reliable delivery of critical alerts, supporting rapid operator response in surveillance scenarios.

* **User Interface Analysis**

The Telegram notification system is designed to be straightforward and accessible for any number of users. To subscribe to alerts, a user simply needs to find the bot on Telegram and send the /start command. When this command is received, the backend registers the user’s unique chat ID. From that moment, whenever the backend detects a dangerous object (e.g., a person) in the video stream, it sends a notification, including a message, GPS location, and optionally an annotated photo, to all registered users.

The multi-user functionality is fully transparent: each user manages their subscription independently. If a user no longer wishes to receive notifications, they can send the /stop command at any time, and the backend will automatically remove their chat ID from the subscribers’ list. This approach respects privacy and ensures that notifications are only sent to those who have explicitly opted in.

To prevent notification spamming, especially in the case of repeated detections in consecutive frames, the backend implements a cooldown mechanism. For each object class (such as “person”), a minimum interval is enforced between notifications, so users receive only relevant alerts without being overwhelmed.

The web dashboard provides clear instructions for subscribing to the bot and displays the notification system’s status in real time. Users are guided through the process: they simply click a link to the Telegram bot, activate notifications with /start, and can deactivate them with /stop.

* + 1. **FR7: GPS Module Integration**

Precise location tracking is enabled by integrating a GPS module into the system. The RC Car’s real-time coordinates are continuously updated and made available through the interface, allowing the operator to track and map the vehicle’s movements accurately.

* **Hardware Analysis and Choice**

To provide real-time geolocation, the system integrates a GPS module directly with the NodeMCU board. The choice to connect the GPS to the NodeMCU, rather than the ESP32-CAM, is primarily driven by pin availability and system scalability. The ESP32-CAM's pins are almost entirely dedicated to camera and motor driver functionality, leaving insufficient resources for additional modules. By contrast, the NodeMCU offers flexible GPIO access and is already responsible for aggregating environmental sensor data (such as temperature and humidity).

The GPS module is connected via a dedicated UART interface to the NodeMCU. This allows continuous acquisition of latitude and longitude coordinates without interfering with the main video and control operations. The NodeMCU periodically reads GPS data, combines it with environmental readings, and transmits the aggregated data via serial UART to the ESP32-CAM. This modular approach improves system maintainability and allows future expansion with minimal hardware changes.

* **Communication Technology Analysis (Software Analysis and Choice)**

The system provides real-time GPS position tracking to enable precise localization of the vehicle during operation. This is achieved by integrating a GPS module connected to a NodeMCU microcontroller, which functions as a sensor bridge. The NodeMCU continuously receives data from the GPS hardware, decodes and maintains the most up-to-date latitude and longitude information in memory, and periodically (every 2 seconds by default) transmits the latest coordinates, together with environmental sensor values, to the ESP32-CAM over a UART serial connection.

On the ESP32-CAM, the firmware parses the incoming serial data stream, extracts the GPS coordinates, and packages them into structured JSON messages. These are immediately forwarded to the backend server over the persistent WebSocket channel. The backend updates the global GPS state and broadcasts each new position to all connected web dashboard clients in real time, ensuring that the vehicle’s current location is always visible to the operator. The frontend dashboard displays the latest GPS coordinates alongside other telemetry and can associate location data with every saved image and detection event.

Communication with GPS is treated like communication with other sensors, so the technologies adopted and the principles and motivations behind them are the same.

* **User Interface Analysis**

The web dashboard displays the current GPS location of the RC Car in real time, updating automatically as fresh data arrives from the backend. The interface presents latitude and longitude values clearly alongside other sensor readings. When saving an image frame or receiving an alert (e.g., via Telegram), the corresponding GPS coordinates are always included, allowing the user to trace the origin of each event precisely. This integrated approach provides operators with full situational awareness, supporting both live navigation and post-event analysis.

1. **Design**
   1. **Architecture Definition**

The architecture of the IoT RC Car for surveillance and data retrieval has been refined to maximize real-time performance, modularity, and user accessibility, fully leveraging a WebSocket-based paradigm for all core operations. The system is organized into three tightly integrated layers that together provide robust, extensible, and secure operation in a wide variety of contexts.

* + 1. **Embedded Subsystem (Onboard RC Car)**

The foundation of the system is the RC Car itself, which incorporates two cooperating microcontrollers: the ESP32-CAM and the NodeMCU ESP8266. This dual-microcontroller approach is adopted to clearly separate the real-time, high-bandwidth tasks, such as video capture and motor control, from the periodic, lower-priority acquisition of environmental and positional data.

* *ESP32-CAM*: Handles all time-critical operations onboard the vehicle. It captures video frames from the camera, encodes them as JPEG images, and streams them via WebSocket to the backend server over WiFi. This microcontroller also manages direct control of the DC motors (for movement and steering) and the flash LED, ensuring immediate response to user commands. All network connectivity is managed directly on the ESP32-CAM, including automatic reconnection and error handling. Additionally, it receives environmental and GPS data from the NodeMCU via a dedicated UART serial connection, integrating these readings into the data packets sent to the backend.
* *NodeMCU ESP8266:* Serving as the vehicle’s “sensor hub,” the NodeMCU is dedicated to reading environmental sensors (such as temperature and humidity) and acquiring current GPS coordinates. This data is aggregated into a structured string and transmitted to the ESP32-CAM via UART at regular intervals. In our implementation, sensor and GPS data are sent every 2 seconds, which represents a good balance between the need for timely updates and the desire to avoid overloading the communication channel or consuming excessive power. This interval can be adjusted easily if different application requirements emerge; for most surveillance and monitoring scenarios, an update every 1-5 seconds is generally sufficient to provide meaningful feedback.

The hardware wiring reflects this division of responsibility: all environmental sensors and the GPS module are connected to the NodeMCU, while the ESP32-CAM handles the camera, motors, and flash. For power supply, the NodeMCU is connected to a power bank, which also supplies the ESP32-CAM through the breadboard circuitry. The motors, due to their higher current demand, are powered by a dedicated 9V battery pack, ensuring stable operation and at least one hour of autonomy even under intensive use.

* + 1. **Communication Layer**

The communication layer serves as the nervous system of the RC Car IoT architecture, ensuring reliable and low-latency data exchange between all subsystems. Its design is based on a clear distinction between onboard communication (within the RC Car) and external communication (between the car and the backend server), each optimized for its specific context.

**Onboard Communication (UART Serial):**

Inside the RC Car, data transfer between the ESP8266 NodeMCU and the ESP32-CAM is managed via a dedicated point-to-point UART serial channel. This physical connection is chosen for its simplicity, robustness, and ability to provide predictable low-latency communication, even in noisy environments. The NodeMCU, acting as the sensor and GPS aggregator, transmits a structured string containing all relevant readings (temperature, humidity, and GPS coordinates) to the ESP32-CAM at fixed intervals (every 2 seconds in the reference implementation). The ESP32-CAM parses these packets and makes the information available for transmission to the backend, guaranteeing that the latest environmental and positional data are always up to date and synchronized with the video stream.

**External Communication (WiFi + WebSocket/HTTP):**

All data exchange between the RC Car and the backend Flask server takes place over WiFi, using standard TCP/IP protocols following a client-server model, with the ESP32-CAM acting as the client and the Flask application as the server. The ESP32-CAM connects to a specified WiFi network and communicates with the backend exclusively via HTTPS requests:

* *Video & Telemetry Upload:* Each captured frame and the latest sensor/GPS data are bundled and sent over the active WebSocket connection to the backend, enabling true real-time streaming and immediate data processing.
* *Command Polling***:** Control commands (movement, effects, etc.) are transmitted from the backend to the ESP32-CAM via the same WebSocket channel, ensuring instantaneous, low-latency response and eliminating the polling delays inherent in HTTP-based approaches.
* *REST API (auxiliary):* REST endpoints are retained for operations not requiring real-time performance (such as gallery management, metadata updates, and manual Telegram notifications). All critical, latency-sensitive communications are handled via WebSocket.

This architectural choice was driven by the need for lower latency and a robust, scalable communication model, particularly for video and control tasks. WebSocket ensures efficient, always-on, and truly bidirectional communication, which is crucial for teleoperation and live monitoring. The system is compatible with secure tunneling solutions (e.g., ngrok) to enable remote access without complex firewall configuration. While HTTP may be used for REST endpoints, all time-critical data now flows through the secure WebSocket channel..

* + 1. **Backend & User Interface Subsystem**

The backend subsystem is built as a centralized client-server platform using Flask, designed to handle data ingestion, processing, persistent storage, and user interaction in a seamless, secure manner. Upon receiving each new video frame and sensor packet from the ESP32-CAM via WebSocket, the backend immediately parses and updates the system’s state. This ensures the web dashboard reflects the most current environmental readings and GPS location alongside the live video feed, maintaining full situational awareness for the operator.

Key features and architectural elements include:

* *Real-time Effects and Detection*: Based on user commands, the backend can apply visual effects (such as negative filters) or run advanced machine learning algorithms (YOLOv3 via OpenCV) for object detection. These operations are performed server-side, offloading all heavy computation from the ESP32-CAM.
* *Persistent Storage with MongoDB/GridFS*: All incoming data, such as images, sensor readings, GPS coordinates, and detection results, is stored in MongoDB, leveraging GridFS for efficient handling of large binary files and associated metadata. This enables robust, scalable storage and advanced querying (including geospatial and statistical analysis).
* *Web Dashboard:* The user interface is a modern, responsive web dashboard accessible from any browser, communicating with the backend via WebSocket for real-time video, status, and control. REST APIs serve only for gallery and asynchronous tasks.
* *Secure Single-User Authentication*: All critical functions are protected by login. The system enforces a single-user model: each vehicle is delivered with unique default credentials (which should be changed at first use), and only one user can control the system at a time, preventing command conflicts and unauthorized access. Passwords are securely hashed, and session management follows best practices.
* *Telegram Bot Integration:* A tightly integrated Telegram bot enables real-time push notifications. When the object detection module identifies a dangerous object (e.g., a person), the system instantly sends an annotated image and GPS position to all registered Telegram subscribers, ensuring alerts reach users even when they are not actively monitoring the web dashboard.
* *Extensibility*: The modular design of both backend and frontend allows easy integration of new sensors, detection models, or notification channels with minimal changes to the core architecture.

This architecture delivers a highly robust, extensible, and user-friendly surveillance platform, capable of adapting to a wide range of deployment scenarios, from home security to industrial monitoring to scientific exploration, while guaranteeing real-time performance, data durability, and secure, exclusive operation.

Immagine che contiene testo, schermata, diagramma, Icona del computer

Il contenuto generato dall'IA potrebbe non essere corretto.

* + 1. **Database**

The database system in the RC car surveillance platform is designed to be transparent, efficient, and secure. It allows for flexible expansion and robust management of all critical data.

**General Structure:**

We rely on MongoDB as our main database, leveraging its document-oriented flexibility and native support for a variety of data types. For storing large binary files, such as captured images and their thumbnails, we use GridFS. This official MongoDB specification seamlessly manages large files by splitting them into chunks and providing robust metadata management. This approach allows us to avoid the pitfalls of traditional filesystem storage, such as filename collisions, incomplete metadata, or concurrency issues under heavy load.

Data in our system falls broadly into three categories:

* + Image data and metadata (for each captured frame);
  + User credentials and authentication data;
  + Session/activity data (for traceability and diagnostics);

Each of these is handled by a dedicated collection or storage mechanism within MongoDB, with strong links between them to guarantee data integrity and facilitate future expansion.

**Table Structure:**

Below is a comprehensive overview of the MongoDB collections and GridFS-related tables:

|  |  |
| --- | --- |
| Collection | Content |
| `surveillance\_images` | Metadata for each saved image (JSON/BSON document) |
| `images.files` (GridFS) | Metadata for each full-resolution image |
| `images.chunks` (GridFS) | Binary chunks of full-resolution images |
| `thumbnails.files` (GridFS) | Metadata for each thumbnail image |
| `thumbnails.chunks` (GridFS) | Binary chunks of thumbnail images |
| `users` | User credentials and metadata (hashed passwords, etc.) |
| system\_config | Global configuration (e.g., dangerous object class labels) |

Each file stored in GridFS (either in `images.files` or `thumbnails.files`) is split into 255KB chunks, which are stored as binary documents in the corresponding `.chunks` collection (`images.chunks` or `thumbnails.chunks`). The `.files` collections contain metadata such as filename, upload date, content type, and references to associated metadata. This chunked storage allows for efficient retrieval, streaming, and management of large files, even if the overall number or size of images grows substantially.

**Image and Metadata Storage:**

Whenever a new image is captured by the ESP32-CAM, either on user request or automatically, both the binary JPEG file and a compressed thumbnail are sent to the backend server. The server first stores the full-resolution image in GridFS, generating a unique `filename` that includes a timestamp to avoid any possible collision. At the same time, a thumbnail (usually 150x150 pixels) is generated using OpenCV and also stored in a dedicated GridFS collection for fast gallery rendering.

The metadata for each image is then assembled in a separate MongoDB document in the `surveillance\_images` collection. Every field name used is explicit and consistent across the system.

Below is a list of all the main fields and their names as found in the saved documents:

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Description |
| `\_id` | ObjectId | MongoDB document unique identifier |
| `filename` | string | Unique image filename (e.g., `captured\_20250709\_190701\_123.jpg`) |
| `gridfs\_image\_id` | ObjectId | Reference to the GridFS file for the full image |
| `gridfs\_thumbnail\_id` | ObjectId | Reference to the GridFS file for the thumbnail |
| `size` | int | Image file size in bytes |
| `created\_at` | datetime | Image creation timestamp |
| `updated\_at` | datetime | Last update timestamp |
| `location` | GeoJSON Point | Optional: `{"type": "Point", "coordinates": [lon, lat]}` |
| `gps\_raw` | object | Raw GPS info: `{ "lat": float, "lon": float, "gps\_string": str }` |
| `environmental` | object | Environmental readings: `{ "temperature": float, "humidity": float, "timestamp": datetime }` |
| `detection` | object | Detection results |
| `effects` | object | Visual effects flags: `{ "negative": bool, "object\_detection": bool }` |
| `tags` | array of str | Tags associated with the image |
| `description` | string | Optional description |
| `analysis` | any | Optional future analytics/AI field |
| `esp32\_session\_id` | string | String identifier for the session/capture batch |
| `system\_version` | string | System version string |
| `saved\_by` | string | The username of user who saved the image |

The detection field structure (`detection`) is the following:

- `active` (bool): Whether detection was enabled for this frame;

- `objects\_count` (int): Number of detected objects;

- `boxes` (array): List of detected objects, each with:

- `x`, `y`, `w`, `h` (int): Bounding box coordinates and size;

- `label` (string): Detected class label (e.g., "person");

- `confidence` (float): Detection confidence (0.0–1.0);

- `class\_id` (int): COCO class index;

- `processing\_time` (float): Time taken for detection (seconds)

- `detection\_timestamp` (datetime): Timestamp of detection process

And a realistic example of the JSON format is like:

```

{

"\_id": "ObjectId(...)",

"filename": "captured\_20250709\_190701\_123.jpg",

"gridfs\_image\_id": "ObjectId(...)",

"gridfs\_thumbnail\_id": "ObjectId(...)",

"size": 63211,

"created\_at": "2025-07-09T19:07:01Z",

"updated\_at": "2025-07-09T19:07:01Z",

"location": { "type": "Point", "coordinates": [12.4924, 41.8902] },

"gps\_raw": {

"lat": 41.8902,

"lon": 12.4924,

"gps\_string": "41.8902,12.4924"

},

"environmental": {

"temperature": 27.6,

"humidity": 48.2,

"timestamp": "2025-07-09T19:07:01Z"

},

"detection": {

"active": true,

"objects\_count": 1,

"boxes": [

{

"x": 120, "y": 90, "w": 200, "h": 320,

"label": "person",

"confidence": 0.88,

"class\_id": 0

}

],

"processing\_time": 0.41,

"detection\_timestamp": "2025-07-09T19:07:01Z"

},

"effects": {

"negative": false,

"object\_detection": true

},

"tags": ["alert", "person"],

"description": "Person detected near entrance",

"analysis": null,

"esp32\_session\_id": "esp32\_session\_1657388821",

"system\_version": "2.1\_gridfs",

"saved\_by": "admin"

}

```

This explicit structure allows for advanced queries and efficient serving in the web dashboard, where thumbnails are shown in the gallery and full images are fetched only on demand.

**User Credentials and Authentication:**

Security and traceability are equally important for a surveillance system. User credentials are stored in a dedicated `users` collection, and we take special care to never store plain-text passwords. Instead, each password is processed using bcrypt, one of the most robust password hashing algorithms available, with a unique salt per user. This ensures that even in the event of a data breach, passwords cannot be recovered by attackers using precomputed tables or brute-force methods.

A user document in the `users` collection has the following fields:

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Purpose |
| `\_id` | ObjectId | MongoDB document unique identifier |
| `username` | string | Unique username |
| `password\_hash` | bytes | Hashed password (bcrypt) |
| `created\_at` | datetime | Account creation timestamp |
| `last\_login` | datetime | Last successful login |
| `is\_active` | bool | Whether the account is enabled |

Whenever a user logs in or changes their credentials, these actions are timestamped and written atomically for full auditability. All authentication and user management logic is handled in the backend (`auth.py`), ensuring that the application never exposes sensitive data even internally.

In this case, an example of the possible JSON format is:

```

{

"\_id": "ObjectId(...)",

"username": "admin",

"password\_hash": "$2b$12$abcdef...",

"created\_at": "2025-07-09T19:07:01Z",

"last\_login": "2025-07-09T19:12:11Z",

"is\_active": true

}  
```

**System Config:**

Stores global configuration settings, in our implementation, includes the list of user-defined dangerous classes for object detection notifications. Each document in `system\_config` has an `\_id` field identifying the config type (e.g., `"notification\_config"`), plus a `dangerous\_classes` field listing the current labels to monitor.

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Description |
| `\_id` | string | Fixed value `"notification\_config"` for this config |
| `dangerous\_classes` | array | List of COCO labels (strings) to be considered dangerous |

An example of the related JSON can be:

```

{

"\_id": "notification\_config",

"dangerous\_classes": ["person", "car", "dog"]

}

```

* 1. **Algorithm Design**

This section presents the main algorithms and logical flows that realize the system’s functional requirements (FRs). Not every FR is mapped to a single algorithm: some requirements are satisfied by shared flows, while others involve tightly coupled processes.

* **User Authentication & Session Management:**

The authentication and session management algorithm guarantees that only one authorized user can access and control the system at any given time, in line with the project's operational safety requirements. When a user first accesses the web application, they are presented with a login page where they must enter their credentials. These credentials are checked against the database (MongoDB), and if they match, a secure session is established. This session allows the user to interact with all protected features of the platform, including video streaming, vehicle control, gallery access, and notification management, without needing to re-authenticate for each action.

Throughout the session, activity from the user (such as dashboard interactions or command sends) automatically renews the session timeout, which ensures that users are not logged out during active use. If there is a prolonged period of inactivity or if the user chooses to explicitly log out, the session is destroyed and all access is immediately revoked, redirecting the user back to the login page. All protected endpoints and APIs check for an active session, so any attempt to access the system without proper authentication is denied.

A fundamental aspect of this algorithm is the management of credentials. Upon first deployment, each device comes with a unique default username and password, which are intended solely for initial access. The user is prompted to change these credentials after logging in for the first time, and can update their username or password at any later time through a dedicated "Change Credentials" interface. To change credentials, the user must provide their current username and password for verification. If the entered information is valid and the new credentials meet the system’s requirements (e.g., unique username, strong password), the change is applied immediately in the database, and the session is updated accordingly if the username was changed. If there is any error, such as entering an incorrect current password, reusing an existing username, or providing a weak password, the system notifies the user and asks for correction.

This approach ensures that the system remains private and secure, with all data and control features strictly limited to the authenticated operator. The use of session management, strong password hashing, and protected communication channels (HTTPS and secure WebSocket) collectively provides a robust defense against unauthorized access, man-in-the-middle attacks, and accidental misuse. While only one user at a time can control the system, the notification subsystem (via Telegram) allows multiple trusted recipients to subscribe to alerts, without granting them control access.

The algorithm can be represented with the following flowchart:

Immagine che contiene testo, schermata, diagramma, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

* **Remote Command Handling:**

The remote command handling subsystem enables the authenticated user to control the RC vehicle in real time from the web dashboard. When the user interacts with the movement controls (directional buttons or keyboard arrows), adjusts the speed slider, toggles the headlights, or activates any available peripheral, the web interface immediately constructs a command message and sends it via a persistent WebSocket connection to the Flask backend. For movements, the most convenient solution to use is that as long as the button is pressed, the command continues to be sent; when the button is released, it is like sending a stop command.

Upon receiving this message, the backend first checks that the user session is valid and authenticated. If the session is active, the backend parses the command and validates it against the set of supported commands. These include vehicle movement ("avanti", "indietro", "sinistra", "destra", "stop"), speed adjustment (with values between 80 and 255), and flash control ("n" for ON, "m" for OFF). The backend maintains the most recent command and the current speed value as part of its internal state. When a speed command is received, it updates the speed value, which will be applied to subsequent movement commands. If any received command is malformed or outside the allowed set, it is rejected, and the user interface is notified.

Once validated, the backend immediately forwards the command to the ESP32-CAM using its WebSocket connection. The ESP32 firmware receives the command and performs the corresponding physical action, such as setting the motor directions, adjusting PWM speed, or toggling the flash LED. For movement, the ESP32 supports moving forward, backward, turning left or right, and stopping. Speed is controlled via PWM and can be updated dynamically. The flash can be turned on or off at any time. The firmware includes a safety mechanism: if no command is received within a defined timeout (typically 250 ms), the motors are automatically stopped to prevent uncontrolled movement in the event of network issues or client crashes. After the command is processed, the backend sends a feedback message to the web interface, confirming the action or reporting any error. The dashboard updates its status display accordingly, so the operator always has immediate visual feedback about the current state of the vehicle and the success of each operation. If the WebSocket connection is interrupted or the session becomes invalid, no commands are accepted, and the user is prompted to log in again.

This approach ensures that every action from the operator is executed with minimal latency, with all possible operations, movement, speed, and flash supported, and provides robust safety and feedback mechanisms in case of network failure or user error.

The algorithm is mapped by the following flowchart:

Immagine che contiene testo, schermata, Carattere, grafica

Il contenuto generato dall'IA potrebbe non essere corretto.

* **Real-Time Video Acquisition:**

The real-time video streaming algorithm is designed to deliver a low-latency, high-reliability video feed from the vehicle’s onboard ESP32-CAM camera to the operator’s web dashboard. This streaming is essential for remote driving, surveillance, and rapid decision-making in challenging environments.

As soon as the ESP32-CAM is powered on and connected to WiFi, it begins capturing images using its camera module. Each frame is JPEG-encoded for high compression and speed, then immediately sent over a persistent WebSocket connection directly to the Flask backend server. This direct, binary transfer minimizes overhead and ensures that each frame is delivered as quickly as possible. The WebSocket protocol was chosen specifically for its ability to support real-time, bidirectional data streams with much lower latency than traditional HTTP polling. Upon receiving a frame, the Flask backend decodes the JPEG data and, if any visual effects (such as negative or object detection overlays) are currently enabled, applies them on the fly using OpenCV. The processed frame is then re-encoded as JPEG and broadcast to all connected, authenticated web clients via their own WebSocket connections. This design allows for multiple simultaneous viewers and consistent, synchronized video feeds.

The algorithm is robust against variable network conditions. If the network slows down or clients drop frames, only the most recent frame is retained and sent, avoiding backlog and excessive memory usage. Video effects and object detection are handled asynchronously in dedicated threads to prevent blocking the main video pipeline and ensure the UI remains responsive. All frame processing is optimized for performance, with frame skipping and adjustable JPEG quality to balance bandwidth and image clarity.

If the ESP32-CAM or the network connection is lost, the backend notifies all clients immediately so the operator can take corrective action. When the connection is re-established, streaming resumes automatically.

This architectural choice, centered on WebSocket streaming and real-time image processing, ensures the operator always has an up-to-date and reliable view of the environment, which is critical for both manual control and AI-assisted surveillance.

The algorithm is represented in the following flowchart:

Immagine che contiene testo, schermata, Carattere, design

Il contenuto generato dall'IA potrebbe non essere corretto.

* **Sensor Data Aggregation & Sync:**

The Sensor Data Aggregation and Synchronization algorithm is responsible for collecting all environmental and operational data from the vehicle’s onboard sensors, such as temperature, humidity, and GPS position, and reliably delivering it to the backend and web dashboard.

When the system powers up, the NodeMCU microcontroller initializes its connected sensors. The GPS module continuously outputs NMEA sentences with latitude and longitude, while the DHT11 sensor provides temperature and humidity readings at fixed intervals (every 2 seconds). The NodeMCU firmware reads sensor values, parses and validates the data, and periodically sends a compact, comma-separated string over UART (serial) to the ESP32-CAM. This string typically includes latitude, longitude, temperature, and humidity.

On the ESP32-CAM side, the firmware reads these serial messages from the NodeMCU. As soon as a valid and complete set of sensor data is received, the ESP32-CAM immediately packages the values into a JSON structure and transmits them over the persistent WebSocket connection to the Flask backend. This approach ensures minimal delay between sensor reading and data delivery, and maintains synchronization between the sensor state and other vehicle operations (such as camera frames and movement commands).

The Flask backend receives sensor data updates via WebSocket, parses the JSON, and updates its internal state. Each new reading is sent to the connected and authenticated web client in real time, so that the dashboard UI can display up-to-date environmental conditions and GPS location. The backend stores each reading alongside video frames and image metadata, ensuring that every captured frame can be associated with the precise sensor and location data at the moment of capture.

The algorithm is robust against missing or malformed data. If a particular sensor reading is unavailable or invalid, the value is marked as "unknown" or omitted from the update, and the system continues operating with the remaining available data. The NodeMCU and ESP32-CAM periodically monitor the freshness of sensor data; if no update is received within a given interval, the backend notifies the user and flags the sensor as temporarily offline.

This synchronized, event-driven data pipeline allows operators to monitor the vehicle’s position and environmental context at all times, provides accurate metadata for saved images and detection events, and lays the groundwork for future extensions with additional sensor types.

**Immagine che contiene testo, schermata, Carattere, design

Il contenuto generato dall'IA potrebbe non essere corretto.**

* **Data Persistence:**

The data persistence module is designed to ensure that all information produced by the RC car system is reliably and efficiently stored, with a structure that supports real-time access, advanced metadata queries, and secure user management. At the core of the storage architecture is a MongoDB database, which offers a flexible document model and is well-suited for handling both structured metadata and large binary files thanks to its native GridFS integration.

The main database, named `iot\_surveillance`, contains several types of data, each with a clearly defined structure. Every time an image is acquired, either by user action or automatically during an object detection event, the backend saves not only the raw image but also a rich set of metadata providing full context for later analysis. The image itself, encoded as a JPEG, is stored in GridFS, which splits large files into chunks and avoids issues with filename collisions or filesystem limits. Alongside the image, a thumbnail is generated using OpenCV and saved separately in GridFS to provide efficient previews in the web dashboard’s gallery.

For each saved image, a dedicated metadata document is created in the `surveillance\_images` collection. This document includes a unique filename, references to the corresponding GridFS file IDs (for both the image and the thumbnail), and a comprehensive set of contextual data: the exact timestamp of acquisition, GPS coordinates (both as raw values and as a GeoJSON point, which enables geospatial queries), and environmental sensor readings (such as temperature and humidity) captured at the moment the image was taken. If the image was generated as a result of object detection, the metadata also records the detection results, including the number and type of objects found, their bounding boxes, the time required for processing, and any visual effects that were applied. Additional fields track user-supplied information such as tags, descriptions, and the username of the person who saved the image, as well as system-specific metadata like ESP32 session IDs and software version.

User authentication data is managed in a separate `users` collection. Each user record contains a unique username, a securely hashed password (using bcrypt), timestamps for account creation and the last successful login, and a flag indicating whether the account is active. The system is initialized with a default admin user, and any credential changes or new users are immediately reflected in this collection. Passwords are never stored in plain text; only salted hashes are saved, ensuring robust security. Each authentication event updates the relevant fields, allowing for auditing and tracking of access over time.

The backend exposes all stored data via dedicated API endpoints. When the user browses the image gallery from the dashboard, the backend retrieves metadata from the `surveillance\_images` collection and streams the relevant image or thumbnail directly from GridFS. This supports rapid browsing even with large datasets and enables advanced filtering by date, detection results, tags, or location. When an image is deleted, both the main file and its thumbnail are atomically removed from GridFS, and the associated metadata document is deleted as well, preventing orphaned data. The system also supports updating metadata, such as adding tags or descriptions after the fact.

All operations are logged for traceability, and the database structure is designed to be easily extendable: new metadata fields, user roles, or data types can be added as the project evolves without breaking compatibility. The combination of MongoDB’s flexible schema, GridFS’s robust handling of large files, and careful metadata design allows the system to scale smoothly and provide a reliable foundation for both current and future needs.

The main algorithms that are used by the database for the various operations performed are represented below.

Immagine che contiene testo, diagramma, schermata, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

**Immagine che contiene testo, schermata, diagramma, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene testo, schermata, diagramma, linea

Il contenuto generato dall'IA potrebbe non essere corretto.**

* **Object Recognition:**

The object recognition subsystem allows the RC car to automatically detect and classify objects using computer vision, both in real time on the video stream and on demand for images stored in the gallery.

When object detection is enabled from the dashboard, the backend processes one out of every three incoming frames from the ESP32-CAM (skipping two out of three to optimize performance). Each selected frame is passed to the YOLOv3 model via OpenCV, which returns bounding boxes, labels, and confidence scores for detected objects. If any objects are found, the frame is annotated and the detection results are broadcast in real time to all connected web clients. If a dangerous object (e.g., "person") is detected with high confidence, a notification is immediately sent via Telegram with the annotated image and relevant context (such as GPS). However, in the live recognition mode, the system does not save every processed frame to the database. Images are only saved when explicitly requested by the operator (e.g., by clicking the save button), at which point the current detection results and metadata are attached to the stored record. This avoids overwhelming the database with unnecessary frames.

An important feature is the ability to run object recognition on any image saved in the gallery. When the user selects the "classify" option on a saved image, the backend loads the original image from GridFS, decodes it, and runs YOLOv3 detection. If at least one object is detected, a new annotated version of the image is saved to the database along with the detection metadata. If no objects are found, the system simply reports the result to the user without saving a duplicate image. Regardless of the detection result, if any dangerous object is recognized, the system sends the annotated image to Telegram as an alert, even if the image is not saved. This guarantees that relevant detections are both actionable and traceable, but avoids polluting the gallery with uninteresting copies.

All detection results, whether from real-time or gallery classification, include precise metadata such as class labels, bounding box coordinates, confidence values, processing time, and timestamps. These are stored alongside the image in MongoDB only if the detection is non-empty. This dual-mode approach allows the system to provide both instant situational awareness and post-hoc analysis, while keeping data storage efficient and focused on relevant events.

This detection algorithm can be represented by the following flowchart:

Immagine che contiene testo, diagramma, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

* **Automated Notification:**

The automated notification subsystem ensures that operators are promptly and intelligently informed whenever the RC car detects specific, user-defined objects of interest. This feature is a core part of the surveillance workflow, tightly coupled to the object recognition module, and is built to maximize both security and situational awareness in real time.

When the backend’s detection pipeline identifies an object classified as “dangerous”, the notification logic is triggered. Users can freely choose which object classes, from those available (such as “person”, “car”, “dog”, etc.), are considered dangerous or relevant to their mission, adapting the notification system to use cases ranging from human intrusion alerts to wildlife monitoring or asset protection. The list of monitored objects can be updated on demand, and these preferences directly drive the real-time notification workflow.

Once a dangerous object is detected with a sufficiently high confidence score, the notification subsystem initiates an alert via a Telegram Bot, which is integrated directly with the backend application. The bot delivers real-time messages to all registered users’ mobile devices, including clear information: the detected object label, its confidence score, a contextual alert message, GPS coordinates of the vehicle at the time of detection, and a snapshot image annotated with bounding boxes. This ensures the operator receives actionable information wherever they are, not just when actively watching the dashboard.

To prevent notification spam, especially relevant when a dangerous object remains in view for multiple frames, the system enforces a configurable cooldown period for each object type (typically 30 seconds by default). When an object triggers an alert, additional detections of the same class are temporarily suppressed, so the operator is not overwhelmed by repeated notifications, but will still be notified of new, significant events or changes in the scene.

The notification module is also designed with robust user management and privacy controls. Only authenticated users can modify notification preferences or trigger manual alerts; Telegram notifications are available only to users who have explicitly registered via the bot (using a /start command), and can be unsubscribed at any time. All notification events are logged for auditability and troubleshooting.

This tightly integrated, customizable, and secure notification mechanism transforms the RC car from a passive sensor platform into an active, always-on sentry, capable of delivering critical, context-rich alerts in real time, tailored to the operator’s mission and environment.

Immagine che contiene testo, diagramma, linea, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

* 1. **Any approach used for sending data**

Designing the data transmission architecture for the RC Car IoT system required a series of iterative, evidence-based decisions, each shaped by the practicalities of embedded hardware, the need for real-time responsiveness, and the diversity of data flowing through the system, including video, sensor readings, remote control commands, and user notifications. The final solution combines both state-of-the-art and pragmatic choices, with each data path optimized for robustness, security, and maintainability.

* **Real-Time Video, Sensor Data, and Commands: WebSocket-First Architecture**

In the current system, WebSocket is the backbone for all real-time, bidirectional communication between the ESP32-CAM and the Flask backend. This includes video frame transmission, sensor data (such as GPS and environmental readings), and control commands for vehicle movement. The move to WebSocket was motivated by its ability to maintain a persistent, full-duplex channel over a single TCP connection, which dramatically reduces latency compared to the old HTTP polling or HTTP POST model. This enables the backend to push updates to the web dashboard instantly and allows the ESP32-CAM to stream video frames and sensor payloads efficiently and with lower overhead.

During development, secure variants of these protocols, namely HTTPS and WebSocket Secure (WSS), were evaluated. However, HTTPS introduced too much latency for real-time streaming use cases on the ESP32-CAM, making the video feed noticeably less responsive. As for WSS, its adoption was hindered by the requirement for a valid SSL/TLS certificate, which was not available during the prototyping and testing phases. Implementing self-signed certificates for WSS was also problematic, as it complicates browser and client trust chains and increases setup complexity, especially for quick local deployments. For these reasons, the system currently operates over standard, unencrypted WebSocket and HTTP channels within a trusted local network environment, prioritizing low latency and ease of deployment. In a production scenario, adding HTTPS/WSS with a valid certificate would be highly recommended to ensure data security.

Alternatives such as RTSP and HTTP polling were evaluated early in development. RTSP, while standard for professional cameras, was rejected due to the lack of native support and the complexity of integrating sensor data per frame. HTTP-based polling and POSTs, although robust, resulted in significant latency and were inefficient on embedded hardware. WebSocket, after careful tuning and the introduction of frame skipping (processing only one out of every three frames for object detection), demonstrated superior stability and responsiveness, even with the ESP32-CAM’s limited RAM. Robust reconnection logic ensures that any dropped connection is automatically re-established, minimizing downtime and operator intervention.

* **Microcontroller-to-Microcontroller: UART for Sensor Aggregation**

Sensor data from the NodeMCU (responsible for environmental and GPS readings) is transmitted to the ESP32-CAM using a UART serial connection. This approach was chosen after comparing I2C, SPI, and wireless methods (like ESP-NOW). UART offers a direct, point-to-point, interference-resistant link with minimal protocol overhead, making it ideal for simple, reliable sensor bridging. The ESP32-CAM receives this data, packages it into its WebSocket messages, and relays it to the backend and web dashboard with each video frame or on a fixed schedule.

* **Control Commands: Real-Time and Fallback**

Movement commands from the dashboard to the vehicle are sent over WebSocket in normal operation, ensuring immediate response and feedback. However, to maximize robustness, the system retains a minimal HTTPS REST API endpoint as an emergency fallback: if WebSocket connectivity degrades, the ESP32-CAM can poll for commands via HTTP GET. This hybrid approach guarantees remote operability even in adverse network conditions, with the persistent, event-driven WebSocket as the primary path and stateless HTTP polling as a last-resort backup.

* **User Interaction and Dashboard: REST + WebSocket**

The web dashboard leverages both RESTful HTTP endpoints (for image gallery management, authentication, and CRUD operations) and WebSocket (for live video, sensor, and event updates). All sensitive operations are protected by session-based authentication using Flask’s secure session cookies and bcrypt-hashed credentials, ensuring that only authorized users can initiate actions or access private data. Real-time video is streamed as a sequence of JPEG frames over WebSocket and rendered in the browser canvas, eliminating the need for MJPEG or legacy HTTP streaming endpoints.

* **Persistent Storage: MongoDB + GridFS**

All image and metadata storage is handled by the backend, which communicates with MongoDB via the PyMongo driver using the native MongoDB wire protocol (over TCP, usually port 27017). Images are stored in GridFS, which fragments large files into chunks and links them to metadata documents containing timestamps, GPS, sensor values, detection results, and user annotations. The backend is the only component with direct access to the database, ensuring data integrity and security; clients interact exclusively with the backend via authenticated API endpoints.

* **Automated Notifications: Telegram Bot API**

Automated notifications for dangerous or user-defined objects are dispatched using the Telegram Bot API. This channel was selected for its instant delivery, support for rich media (including annotated images), and user-friendly subscription model. Multiple operators can register or unregister for alerts simply by messaging the bot. The backend, upon detection of a relevant event, sends the alert message and image directly to all active subscribers using HTTPS requests to the Telegram servers. Cooldown logic is enforced to prevent notification spam.

* **Security Considerations**

All web-facing endpoints and data flows are protected by session-based authentication. Credentials are securely hashed with bcrypt, and all user actions are auditable. The direct MongoDB connection is accessible only to the backend, further isolating the database from untrusted clients. While JWT or OAuth could be adopted for more complex, multi-user or distributed scenarios, the current architecture’s simplicity and local focus prioritize robust, operator-friendly access control.

We summarize the communications in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| Data Flow | Protocol | Sender → Receiver | Security/Notes |
| Video, sensors, commands | WebSocket (TCP) | ESP32-CAM ↔ Flask | Persistent, bidirectional, low latency |
| Sensor bridge | UART (serial) | NodeMCU → ESP32-CAM | Simple, point-to-point, robust |
| Fallback movement ctrl | HTTP GET | ESP32-CAM → Flask | Emergency use only |
| User commands/gallery | HTTP (REST) | Browser ↔ Flask | Authenticated, session-based |
| Image storage | MongoDB Wire | Flask ↔ MongoDB | Backend only, never exposed to clients |
| Telegram notifications | HTTP (Telegram) | Flask → Telegram API | Rich media, opt-in, cooldown, instant |

This multi-layered, protocol-optimized architecture allows the RC Car system to achieve the goals of real-time control, robust data delivery, security, and extensibility expressed by the NFRs, while remaining practical for embedded hardware and local deployments.

* 1. **Design of HW wiring.**

The hardware wiring design for the RC Car surveillance system was developed with the primary objectives of reliability, modularity, and expandability. This design supports the system’s functional requirements and ensures robust communication, efficient power distribution, and ease of maintenance for future upgrades or troubleshooting.

At the core of the hardware subsystem, two microcontrollers operate in close synergy: the ESP32-CAM, which manages video, motor control, and network communication, and the NodeMCU ESP8266, which acts as a dedicated sensor hub. The division of responsibilities between these boards is reflected directly in the physical wiring layout.

* **Sensor and Peripheral Connections:**

All environmental sensors, such as the DHT11 for temperature and humidity, and the GPS module for acquiring real-time location, are connected exclusively to the NodeMCU. For example, the DHT11 sensor is wired to the NodeMCU’s digital pin D2, while the GPS module is connected to D1 and communicates via a software serial interface. This arrangement ensures that sensor readings are decoupled from the time-critical video streaming tasks, reducing interference and allowing each microcontroller to operate optimally within its domain. The NodeMCU reads all sensor values at regular intervals (typically every 2 seconds) and assembles them into a single structured string. This unified data packet is then sent over a dedicated UART serial connection to the ESP32-CAM. The UART wiring follows the standard convention, with the TX pin on the NodeMCU connected to the RX pin on the ESP32-CAM and vice versa, and a shared ground between the two boards to guarantee signal integrity and prevent communication errors.

|  |  |  |  |
| --- | --- | --- | --- |
| Peripheral / Sensor | Board | Pin | Description / Note |
| UART RX (from NodeMCU TX) | ESP32-CAM | U0RXD | Serial data input from NodeMCU (sensor data) |
| UART TX (to NodeMCU RX) | ESP32-CAM | U0TXD | Serial data output to NodeMCU (if needed) |
| UART TX (to ESP32-CAM RX) | NodeMCU | TX | Serial data output (aggregated sensor values) |
| UART RX (from ESP32-CAM TX) | NodeMCU | RX | Serial data input (optional, not used) |
| DHT11 Temp/Humidity Sensor | NodeMCU | D2 | Reads temperature and humidity |
| GPS Module TX | NodeMCU | D1 | Serial input (receives GPS data) |
| GPS Module RX | NodeMCU | (Not connected) | Not used (GPS is only sending data to NodeMCU) |
| GND (Common Ground) | - | GND | Shared ground reference for all modules |

* **Motor Wiring and Flash:**

The wiring of the DC motors in the RC Car is designed for robust and predictable behavior, while conforming to the hardware limitations of the ESP32-CAM and the need to avoid conflicts with the camera's data lines. In this project, speed control for both motors is handled through a single ESP32-CAM output pin, while direction control remains fully independent for each motor.

Specifically, both enable pins (ENA and ENB) of the L298N H-Bridge motor driver are physically connected and wired to GPIO12 on the ESP32-CAM. As a result, both the left and right motors always receive the same Pulse Width Modulation (PWM) signal for speed control, meaning they always operate at the same speed. This approach greatly simplifies the wiring and firmware logic and avoids the need for additional PWM channels on the ESP32-CAM, which are limited and already partially used by the camera module.

Direction control for each motor remains independent, achieved as follows:

* The right motor's direction is controlled by IN1 (connected to GPIO13) and IN2 (connected to GPIO15).
* The left motor's direction is controlled by IN3 (connected to GPIO14) and IN4 (connected to GPIO2).

By setting these direction pins HIGH or LOW in various combinations, the firmware can command the car to move forward, reverse, or turn by making the wheels rotate in opposite directions, while both motors always share the same speed.

For illumination, a flash LED is connected to GPIO4 of the ESP32-CAM. This LED can be remotely activated or deactivated through the web interface, providing necessary lighting for the onboard camera in low-light conditions.

During system initialization, all motor direction and flash pins are set to LOW, and the enable pin is set to zero PWM, ensuring that the motors remain stationary and the car does not move unexpectedly during boot or reset. Only after the system is fully initialized and user commands are received do the motors become active.

This explicit wiring configuration, with both ENA and ENB on GPIO12, independent direction pins, and the flash on GPIO4, ensures that the system is reliable, easy to maintain, and accurately reflects both the hardware and software implementation.

The following table summarizes the key wiring connections between the ESP32-CAM and the L298N motor driver:

|  |  |  |
| --- | --- | --- |
| Function | ESP32-CAM GPIO | L298N Pin |
| Motor speed (ENA+ENB) | GPIO12 | ENA, ENB |
| Right motor direction | GPIO13, GPIO15 | IN1, IN2 |
| Left motor direction | GPIO14, GPIO2 | IN3, IN4 |
| Flash LED | GPIO4 | - |

* **Power Distribution Strategy:**

A well-designed power distribution scheme is fundamental for the reliability and longevity of any embedded system, and is especially critical in a mobile, sensor-rich platform like this RC Car. In this project, careful consideration was given to both the electrical requirements of each component and the need to isolate sensitive electronics from sources of noise and instability.

The NodeMCU is powered by a standard USB power bank, which ensures a stable, regulated voltage and provides sufficient current for prolonged operation. The ESP32-CAM is powered from the 5V output VV of the NodeMCU via breadboard. All environmental sensors, such as the temperature/humidity sensor (DHT11) and the GPS module, are in turn powered via the 3.3V output pin of the NodeMCU itself, with the supply routed through a breadboard to facilitate easy wiring and prototyping. This arrangement allows all low-power digital components to share a single, reliable supply chain, while the USB power bank guarantees at least one hour of continuous use as required by the project’s autonomy goal. The use of a USB power bank also makes it easy to recharge or swap out the main supply, further enhancing the system’s practicality and portability.

In contrast, the DC motors, which draw substantially higher currents and can generate electrical noise, are powered by a separate 9V battery pack from the L298N, which is directly powered by the battery linked to the 12V input. By keeping the motor supply independent from the logic supply, potential issues such as voltage drops and transient disturbances caused by sudden changes in motor load are prevented from propagating to the microcontrollers. This separation is essential to avoid unexpected resets, data corruption, or erratic behavior of the ESP32-CAM and NodeMCU, especially during maneuvers that require rapid acceleration or changes in direction.

In a potential production scenario, it would be possible to power the entire system, including both the logic circuits (ESP32-CAM, NodeMCU, and all sensors) and the DC motors, with a single 12V supply. Thus, eliminates the need for a separate USB power bank and reduces the overall complexity of the power system. This integrated approach would offer a more compact and streamlined solution, particularly suitable for mass production or deployments where space is at a premium. However, due to the unavailability of a dedicated 12V supply during prototyping and testing, the split power strategy, with independent sources for logic and motors, was adopted in the current implementation.

Despite the physical separation of the power sources, all ground (GND) lines from the USB power bank, the 9V motor battery, the ESP32-CAM, the NodeMCU, and all sensors are connected to establish a single common reference point. This common ground is crucial for several reasons: it ensures accurate sensor readings and it guarantees robust UART serial communication between the NodeMCU and the ESP32-CAM, which fundamentally depends on a shared ground potential for correct logic level interpretation.

This dual-domain power strategy, with strict ground unification, provides both the electrical isolation needed for stability and the reference integrity required for reliable sensing and communication, forming the backbone of the RC Car’s dependable autonomous operation.

|  |  |  |  |
| --- | --- | --- | --- |
| Component | Power Source | Voltage | Notes / Routing |
| ESP32-CAM | USB Power Bank | 5V | Supplied from NodeMCU VV output, routed through breadboard |
| NodeMCU | USB Power Bank | 5V | Powered via USB cable |
| Environmental Sensors | NodeMCU (3.3V pin) | 3.3V | Supplied from NodeMCU 3.3V output, routed through breadboard |
| GPS Module | NodeMCU (3.3V pin) | 3.3V | Supplied from NodeMCU 3.3V output, routed through breadboard |
| DC Motors (via L298N) | 9V Battery Pack | 9V | Motors have dedicated supply, physically separated from logic power |
| L298N Logic (Vcc) | 9V Battery Pack | 9V | Logic side of H-Bridge powered with the same battery pack of motors |
| Common Ground | All sources | - | All GND lines connected together for reliable reference |

The wiring layout is deliberately organized to facilitate both initial assembly and future maintenance. Signal wires, especially those for UART and sensor data, are kept as short as possible to minimize electromagnetic interference (EMI) and signal degradation. The use of a breadboard or prototyping PCB allows for clean routing of power and signal lines, grouping related components (e.g., sensors, motors, camera) to simplify troubleshooting. During prototyping and assembly, all connections are clearly labeled and documented, both in the physical build and in the accompanying schematic/wiring diagram. This documentation is essential for ensuring reproducibility and ease of handover to other team members or future developers.

A fundamental goal of the wiring design is to support future scalability. By grouping sensor connections on the NodeMCU, the system can be easily expanded to accommodate new types of sensors (such as gas detectors or ultrasonic rangefinders) with minimal changes to the existing wiring.

Immagine che contiene testo, schermata, manometro

Il contenuto generato dall'IA potrebbe non essere corretto.

* 1. **Associations of real objects – Digital Twins**

In this project, the notion of a Digital Twin is not treated as an abstract technological concept but as a practical, essential tool that enables effective remote operation, supervision, and analysis of the RC Car system. The digital counterpart of the RC Car is embodied in the entire software stack: the Flask backend, the database, and above all, the web dashboard that operators use to interact with the vehicle.

Every aspect of the car's physical state, such as its location, environmental conditions, video feed, sensor status, and system status, is captured and constantly synchronized with the web application. When the RC Car moves, the GPS coordinates on the dashboard update in real time; when a sensor measures a temperature or humidity value, this reading is shown instantly to the user; if the wireless connection drops or a sensor malfunctions, the dashboard reflects this with clear status messages or alerts. The live video stream, transmitted from the ESP32-CAM, completes the experience, allowing the operator to “see through the eyes” of the vehicle and make informed decisions as if they were physically present.

Control commands also flow in the opposite direction: when an operator interacts with the dashboard (steering, toggling the flash, or saving an image frame), these commands are transmitted to the physical car, executed by the onboard firmware, and their effects are immediately reflected in the dashboard. This creates a seamless feedback loop between the real system and its digital representation, ensuring that the two are always aligned.

The web application is designed not only to present the current state, but also to aggregate historical data, such as saved images, detected objects, alert logs, and sensor trends. This enables both immediate awareness and later analysis, such as reviewing where the vehicle has been or what it has detected. The backend also handles notifications, for example via Telegram, to alert the operator if critical objects are recognized by the onboard AI.

During development and testing, this “digital twin” approach proved invaluable: it allowed for rapid identification of failures or misalignments between hardware and software, remote troubleshooting, and efficient demonstration of system capabilities to non-technical users. The architecture is inherently extensible: new sensors or actuators on the car can be integrated with minimal changes to the digital side, ensuring that the digital twin always faithfully mirrors the evolving capabilities of the physical platform.

Ultimately, in this RC Car system, the digital twin is not an optional extra but the primary interface for all meaningful interaction, control, and monitoring. The entire user experience, from driving the vehicle to analyzing sensor data and reviewing surveillance events, happens through this live, responsive, and comprehensive digital counterpart, which is tightly coupled with the real car at all stages of operation.

* 1. **Mock-up of any user interface**

1. **Limitations and Future Expansions**

The current RC car surveillance platform is a robust proof-of-concept, but like any prototype, it comes with a set of practical and architectural limitations that also point the way to future evolutions. Below, we list the main constraints encountered so far and the most promising directions for further development, always keeping in mind that several current limitations can become opportunities for expansion in the next iterations of the project.

**Power Supply:**

The system relies on a dual power supply, one for the ESP32-CAM and one for the motors. This is not due to a design preference, but simply to limited component availability (not having a reliable, compact step-down converter at build time). Integrating all power management into a single, unified battery system would simplify wiring, reduce weight, and increase reliability (12V should be sufficient). In the future, a control for the percentage of battery available can also be developed, possibly plotting the value on the user dashboard.

**ESP32-CAM Image Quality:**

The ESP32-CAM remains a compelling choice for prototyping thanks to its low cost, integrated WiFi, and extremely compact footprint, making it ideal for battery-powered or size-constrained robotics. However, the limitations of its OV2640 sensor and the board’s available RAM quickly become apparent in practical use. While datasheets promise high resolutions, in reality, the module is only stable at VGA (640x480) or below; attempts to use higher resolutions lead to frequent buffer overflows, dropped frames, or outright crashes, especially during continuous streaming or image capture. These constraints mean the platform is not suitable for tasks such as high-accuracy object recognition or any application where visual detail is important. In the future, it will be necessary to evaluate alternative camera solutions that maintain a small size and low power consumption but offer a significant boost in image quality and reliability. This could include Raspberry Pi Camera modules (v2, v3, or HQ) or compact USB UVC cameras paired with a Raspberry Pi Zero 2 W, Jetson Nano, or similar single-board computers The most important requirements will be: physical compactness (to fit the car’s chassis), moderate power draw (to preserve autonomy), and software support for robust streaming and real-time inference.

**Unencrypted Communications:**

Currently, all network communications, including both HTTP requests and WebSocket (WS) connections, are unencrypted. This means credentials, video streams, and sensor data can be intercepted by anyone on the same network if the system is used outside a trusted LAN. During development, we attempted to use WSS (WebSocket Secure), but without a valid, trusted certificate (i.e., one signed by a recognized Certificate Authority), browsers and most client libraries simply refuse to establish the connection, unlike HTTPS, where you can sometimes bypass warnings with self-signed certificates. This strict requirement is the main reason why the current prototype operates in cleartext: it avoids the complexity of certificate management and compatibility issues during testing. However, for any real-world or public deployment, migrating to HTTPS/WSS with proper certificate management is not just recommended but essential to protect user data and system integrity.

**Web App and Gallery Scalability:**

At present, the web dashboard fulfills its basic role as a gallery and management interface for saved images, but it is not designed for large-scale or long-term use. All images are listed in simple chronological order, and the only navigation is scrolling; there are no tools for filtering by time, detected object, tag, or GPS location. As the number of saved frames grows, finding specific events or images quickly becomes impractical. In terms of future development, a priority will be to implement advanced filtering and search functionality, allowing users to narrow results by date, tag, detected object, or even environmental parameters. This flexibility would make the gallery not just a passive archive but an intelligent and interactive data exploration tool.

**Sensor and AI Expansion:**

While the system is designed with modularity in mind, only a basic set of sensors is currently integrated: temperature and humidity (DHT11), and GPS for geolocation, all managed via the NodeMCU. Object detection is limited to YOLOv3. There is significant potential for future expansion: adding more environmental sensors (such as gas sensors like MQ-2 or MQ-135, air quality sensors like CCS811 or BME680, or even a microphone for audio event detection) would broaden the monitoring capabilities. On the AI side, integrating more advanced computer vision features, such as anomaly detection, face recognition, or custom-trained models for specific use cases, is a natural next step. The architecture is already prepared for these additions, requiring updates primarily to the NodeMCU firmware and the backend’s data handling logic.