**IoT Project**

**RC Car for Surveillance or Data Retrieval**

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

This section analyzes each Functional Requirement for the Radio-Controlled Car and discusses the rationale behind the implemented solutions.

* + 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):**

Communication between the RC Car and the operator is established over WiFi, leveraging the ESP32-CAM’s built-in 802.11b/g/n capabilities. WiFi was selected for its high bandwidth and low latency, both crucial for responsive remote control and live video streaming. Compared to alternatives like Bluetooth or cellular networks, WiFi offers a straightforward, cost-effective solution well-suited to environments like labs, schools, and homes. While cellular connectivity could extend the system’s reach to outdoor or infrastructure-less scenarios, it would require additional hardware and ongoing data plans, which were beyond the scope of this prototype.

On the software side, the system uses a RESTful HTTPS protocol for command and control. The web interface sends movement commands as HTTPS POST requests to a Flask backend running on a secure server or PC. The ESP32-CAM, in turn, polls the backend using HTTPS GET requests to retrieve and execute the latest command. The explicit use of HTTPS (rather than HTTP) ensures that all communication, including user login credentials, control commands, and sensor data, is encrypted during transit. This is critical for protecting against eavesdropping, man-in-the-middle attacks, and unauthorized access, especially since the web dashboard includes user authentication features.

This architecture was chosen for its simplicity, compatibility, and strong security properties. HTTPS is universally supported on both browsers and embedded devices, and it integrates naturally with Flask and the ESP32’s networking libraries. It also makes debugging and monitoring traffic straightforward, while providing a robust foundation for future expansion (such as secure WebSocket integration or API authentication tokens).

While protocols like MQTT or WebSocket Secure (WSS) could offer more efficient or continuous real-time messaging, adopting them would increase system complexity and require dedicated infrastructure (for example, an MQTT broker or a persistent socket server). For the current project goals, reliable and secure remote control, simplicity of deployment, and ease of maintenance, the combination of HTTPS-based REST APIs and periodic polling offers the best tradeoff between security, performance, and developer effort.

* **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**
* **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)**

For real-time image retrieval, the system utilizes WiFi for data transmission between the ESP32-CAM and the Flask backend server. JPEG frames are periodically captured on the ESP32-CAM and sent to the backend via HTTPS POST requests to a dedicated /upload endpoint. The backend receives, buffers, and processes these frames in real time. Using HTTPS ensures all images and metadata are transmitted securely, protecting them from eavesdropping or tampering. The Flask server then serves the video stream to the web interface via a dedicated /video\_feed endpoint using an MJPEG (multipart JPEG) response, which is supported by modern browsers and requires minimal client-side processing. This architecture offers a good balance between simplicity, cross-platform compatibility, and low latency, meeting the requirements for responsive surveillance and remote operation.

* **User Interface Analysis**

The user interface displays the live video stream in real-time directly within the dashboard. 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**
* **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)**

Sensor data is transmitted from the NodeMCU to the ESP32-CAM via UART serial at 9600 baud, ensuring low-latency and robust communication even in noisy environments. The ESP32-CAM parses these values and includes them as custom HTTPS headers when uploading each camera frame to the Flask backend. This piggybacking approach minimizes network overhead and ensures that every image/frame is associated with the most recent sensor readings, all while maintaining secure, encrypted communication between the device and the server.

The Flask backend extracts these sensor values from the HTTPS headers of incoming image POST requests and updates the latest environmental readings in its internal state. The web interface can then retrieve up-to-date sensor data at any time by querying dedicated REST endpoints, receiving the most recent values in JSON format. This architecture provides a simple, scalable, and real-time-capable solution for monitoring sensor values and can be easily extended with additional sensors by updating the NodeMCU firmware and backend parsing logic.

* **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 changes in the environment.
* This approach enables non-technical users to benefit from advanced sensor capabilities with no need for specialized tools or software, and supports future expansion (e.g., sensor history, alerts) with minimal changes to the interface or API.
  + 1. **FR4 - Data Saving**
* **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)**

When the user issues a command to save an image frame via the web interface, the backend Flask application retrieves the most recent frame received from the ESP32-CAM. The saving process is initiated by an HTTPS POST request from the dashboard to the dedicated /save\_image endpoint. The backend then applies, if it’s present, any active video effects (such as negative or object detection overlays) to the frame before saving it. The image, along with its metadata (timestamp, GPS position, sensor readings, applied effects), is stored in a MongoDB database using GridFS, which allows for efficient handling of binary files and scalable metadata management.

This approach ensures that the saved images are always synchronized with the current state of the system and preserves all relevant context for later analysis. All communications involved in the saving process use HTTPS, guaranteeing data integrity and confidentiality during transfer between client and server. The backend architecture is modular, making it easy to extend with additional image processing or export features if needed.

* **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**
* **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)**

Image frames are continuously sent from the ESP32-CAM to the Flask backend via HTTPS POST requests. The object recognition process is triggered in two scenarios:

* *Real-time detection*: When the detection mode is enabled from the web interface, each incoming frame is queued for processing by the backend. YOLOv3 is applied server-side to detect and classify objects in the scene, and the results are overlaid on the live video stream before being sent to the operator.
* *On-demand analysis:* When a frame is saved (FR4), the backend can also perform object recognition on the stored image for later review and statistics.

All communication between the ESP32-CAM, backend, and web dashboard uses secure HTTPS channels to ensure data integrity and privacy. The backend uses efficient queues and threading to avoid blocking the live video stream during detection, and results are linked to each frame’s metadata for traceability. Object detection results (labels, bounding boxes, confidence) are stored in the database alongside each analyzed image.

* **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**
* **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)**

To provide automated notifications via Telegram, it is essential that the backend Flask server is reachable from the public internet so that Telegram’s servers can deliver webhook events and receive updates in real time. In most development scenarios, the Flask backend is hosted on a private or local network, which is not directly accessible from outside. To overcome this, the project uses ngrok, a practical tunneling service that creates a secure, temporary HTTPS endpoint linked to the local Flask server.

When ngrok is started, it generates a unique public URL, which is then set as the webhook endpoint for the Telegram bot. This allows Telegram’s infrastructure to communicate directly with the backend, forwarding commands like /start or /stop from users, and enabling the backend to send notifications and images to all subscribers. This solution is particularly effective for development, testing, and demonstration, as it does not require the configuration of firewalls, port forwarding, or a static IP address. All communication through ngrok is encrypted, guaranteeing data confidentiality and integrity for both commands and notifications.

* **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 own 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**
* **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 NodeMCU runs firmware that retrieves GPS coordinates in real time and sends them to the ESP32-CAM as part of a combined sensor data string (e.g., "lat, lon, temp, hum"). The ESP32-CAM, upon receiving this data, parses the values and attaches them as custom HTTP headers (e.g., X-GPS) when uploading each video frame to the Flask backend.

On the backend, the Flask application extracts the GPS data from the incoming headers with every frame upload, ensuring that location information is always synchronized with the video stream and sensor readings. The latest GPS coordinates are stored in memory and made available to the web interface via dedicated endpoints. This design ensures that every saved image and every detection event can be correlated with an exact physical location, greatly enhancing the system’s utility for surveillance and data analysis.

* **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. **Analysis of Non-Functional Requirement:**
     1. **NFR1 - System Responsiveness**

The responsiveness of the control interface is a key factor in ensuring safe and effective remote operation. The system is designed so that user commands (steering, speed, effect toggles) are transmitted from the web interface to the Flask backend and then polled by the ESP32-CAM with minimal latency. Network communication is optimized via lightweight HTTPS requests and the use of queues in the backend, ensuring that the time from the user’s command to the car’s physical reaction remains within a fraction of a second under normal WLAN conditions. The video stream, processed in real time, is displayed with low latency on the dashboard to provide immediate visual feedback.

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

The RC Car is engineered to operate for at least one hour of continuous use on a single charge. This is achieved through careful hardware selection (efficient DC motors, optimized ESP32 and NodeMCU duty cycles) and code-level optimizations (minimized polling intervals, sleep modes where possible). The system avoids unnecessary sensor polling and disables power-hungry peripherals (e.g., the camera flash) when not needed. Energy usage is further monitored and can be logged for future improvements.

* + 1. **NFR3 - Scalability**

Scalability is built into the system both at the hardware and software levels. The NodeMCU’s flexible GPIO layout allows for the addition of new sensors (e.g., extra environmental modules, advanced GPS, IMU) without rewiring the core system. On the backend, the database schema and API endpoints are designed to accommodate new types of data and functionalities. The modular codebase (with clear separation between video, control, sensor management, and notifications) ensures that new features can be integrated with minimal impact on existing performance.

* + 1. **NFR4 - Durability**

The mechanical and electronic components of the RC Car are selected to withstand outdoor use, including minor shocks and vibrations. All PCBs and wiring are secured in protective enclosures, and connectors are chosen for their resistance to loosening. The firmware includes safety timeouts to avoid motor burnout and can detect abnormal sensor values, triggering failsafe routines if necessary.

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

The web dashboard and notification system are designed for users with no technical background. The interface provides clear controls, real-time status feedback, and tooltips. All critical actions (such as subscribing to notifications or saving frames) require at most one or two clicks. The system includes guidance for connecting to the Telegram bot and visualizes all relevant data (video, sensor values, GPS) intuitively.

* + 1. **NFR6 - Affordability**

All components are selected with cost-effectiveness in mind, using widely available modules such as the ESP32-CAM and NodeMCU. The use of open-source software (Flask, OpenCV, YOLOv3, MongoDB) eliminates licensing costs. The system can be assembled and maintained with a modest budget, making it suitable for educational, hobbyist, or pilot industrial applications.

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

The architecture of the IoT RC Car for surveillance and data retrieval has been carefully designed to balance real-time performance, modularity, and ease of use. The system is organized into three primary layers that interact seamlessly to deliver all required functionalities.

* + 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*: This microcontroller is dedicated to time-critical operations. It captures video frames from the onboard camera, encodes them as JPEG images, and transmits them to the backend server over WiFi. In addition, it is responsible for directly controlling the DC motors for movement and steering and the flash LED, ensuring immediate response to user commands. The ESP32-CAM also manages the WiFi connection, handles network errors, and performs polling for movement or effect commands sent by the user via the backend. It also receives data from the NodeMCU via a dedicated UART serial connection to enhance vehicle and environmental monitoring.
* *NodeMCU:* 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 if different application requirements emerge; for most surveillance and monitoring scenarios, an update every 1–5 seconds is generally sufficient to provide meaningful feedback for both vehicle control and environmental awareness.

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 battery pack, ensuring stable operation and at least one hour of autonomy even under intensive use.

* + 1. **Communication Layer**

The communication layer acts as the nervous system of the architecture, ensuring seamless data flow between the car and the backend. The ESP32-CAM connects to a WiFi network and uses HTTP/HTTPS requests to send video frames (with embedded sensor and GPS data in custom headers) to the Flask backend. At the same time, it periodically polls the backend for new movement or effect commands, which are then executed with minimal latency. The use of industry-standard protocols such as HTTPS and UART not only guarantees compatibility and robustness but also greatly simplifies troubleshooting and future expansions. This architecture also supports the use of tunneling tools like ngrok for remote access, making the system suitable for demonstrations and real-world deployments without complex network configuration.

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

The backend is built around a Python Flask application, which acts as a central hub for data ingestion, processing, storage, and user interaction. When the backend receives a new video frame, it immediately parses the accompanying sensor and GPS data, updating the system state in real time. Depending on user settings, it can apply visual effects to the video stream or invoke machine learning algorithms (such as YOLOv3) for automatic object detection. All data—images, sensor readings, GPS locations, and detection results—are persistently stored in a MongoDB database using GridFS, ensuring that nothing is lost and enabling detailed post-event analysis.

The backend also exposes a secure web interface, accessible via any modern browser, that allows users to control the car, view live video and sensor data, manage saved images, and configure notification preferences. For added security and flexibility, the system integrates session-based authentication and supports multiple users. A Telegram bot is tightly integrated into this layer, delivering instant notifications (with annotated images and precise coordinates) to any number of subscribers whenever dangerous objects are detected.

* + 1. **Real-World and Digital Association**

Every action performed on the RC Car, be it a movement command, a sensor reading, or a detection event, has an immediate digital counterpart in the backend and user interface. This creates a true "digital twin" of the vehicle’s operation, where physical events are mirrored and recorded in the digital world. The database maintains a complete history of all relevant data, linking each image or notification to its exact real-world context.

**Competitors analysis???**