# Vehicle-to-Cloud Report

**Prepared by:**

**Saif Tamer**

**Ali Adel**

**August 20, 2025**

## Table of Contents

# 1. Executive Summary

## 1.1 Goals & approach

The main goal of the tasks is to enable vehicle-to-cloud communication for real-time monitoring of key vehicle parameters. The approach combines embedded systems, cloud integration, and web-based visualization:

Establish communication between the STM32F4 microcontroller and the ESP32 module, where the ESP32 is configured to handle Wi-Fi connectivity and data transmission.

Design a basic cloud system to receive, store, and make vehicle data accessible through standard scripts written in Python and C++.

Develop a dashboard web application using PHP to provide the driver with real-time status and vehicle performance information in an accessible format.

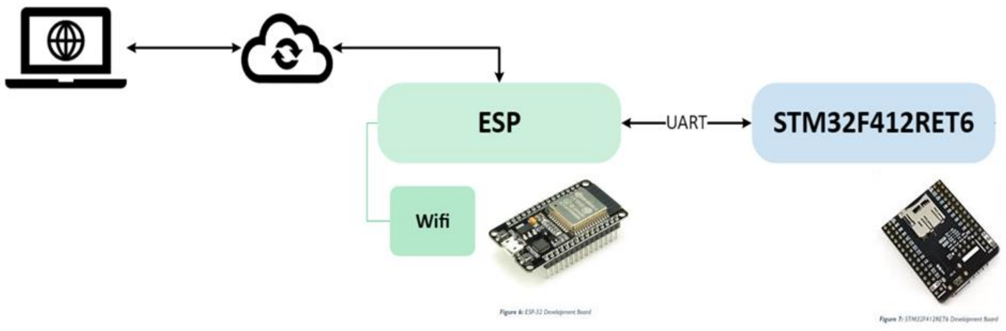

ESP
UART
STM32F412RET6
Wifi
Figure 6: ESP-32 Development Board
Figure 7: STM32F412RET6 Development Board

*Figure 1: System Overview*

## 1. 2 Main results

Successful communication setup: The ESP32 reliably transmitted sensor data to the cloud, ensuring stable connectivity and structured data transfer.

Cloud system deployment: A simplified cloud environment was implemented, capable of storing incoming data and providing retrieval mechanisms for further analysis.

Functional dashboard: A PHP-based web dashboard was created, displaying real-time parameters to support driver awareness and vehicle monitoring.

Initial performance insights: System testing revealed effective data transfer with manageable latency

## 1. 3 Codes

All source codes, including the ESP32 modules, STM32 communication, and the station-side Python scripts, are available on GitHub and have also been uploaded alongside this document on the team's shared drive.

Github link:[https://github.com/aliadelmahdi/Racing-Team-Tasks]

# 2. Tasks Overview

## 2. 1 ESP Code Structure

The ESP32 software is modularized into multiple source and header files to ensure scalability, readability, and ease of maintenance:

- HTTPRequest.cpp / HTTPRequest.h

  Implements HTTP request handling, enabling the ESP32 to send and receive data from cloud services.

- JSONParser.cpp / JSONParser.h

  Responsible for parsing and processing JSON-formatted data exchanged with the cloud server.

- main.ino

  The central esp sketch that initializes the system, manages main loop operations, and coordinates between modules.

- RandomLinkGenerator.cpp / RandomLinkGenerator.h

  Provides randomized link generation functions, used for testing the cloud without UART.

- sendftp.cpp / sendftp.h

  Handles file/data transfer via FTP, used as an alternative communication channel.

- uart.cpp / uart.h

  Manages UART communication with the STM32F4 microcontroller. It defines the buffer structure, parsing routines, and synchronization of incoming/outgoing data.

- WiFiManager.cpp / WiFiManager.h

  Configures and manages Wi-Fi connectivity of the ESP32, ensuring stable network communication.

This modular separation allows independent development and debugging of communication, networking, and data-handling features.

## 2. 2 Microcontrollers Overview

- **ESP32**

The ESP32 is a dual-core Wi-Fi and Bluetooth-enabled microcontroller. In this system, it serves as the networking and cloud gateway, responsible for:

- Establishing Wi-Fi connection
- Handling HTTP/FTP communication with the cloud
- Managing data parsing and formatting
- Sending updated telemetry values from the STM32 to the server

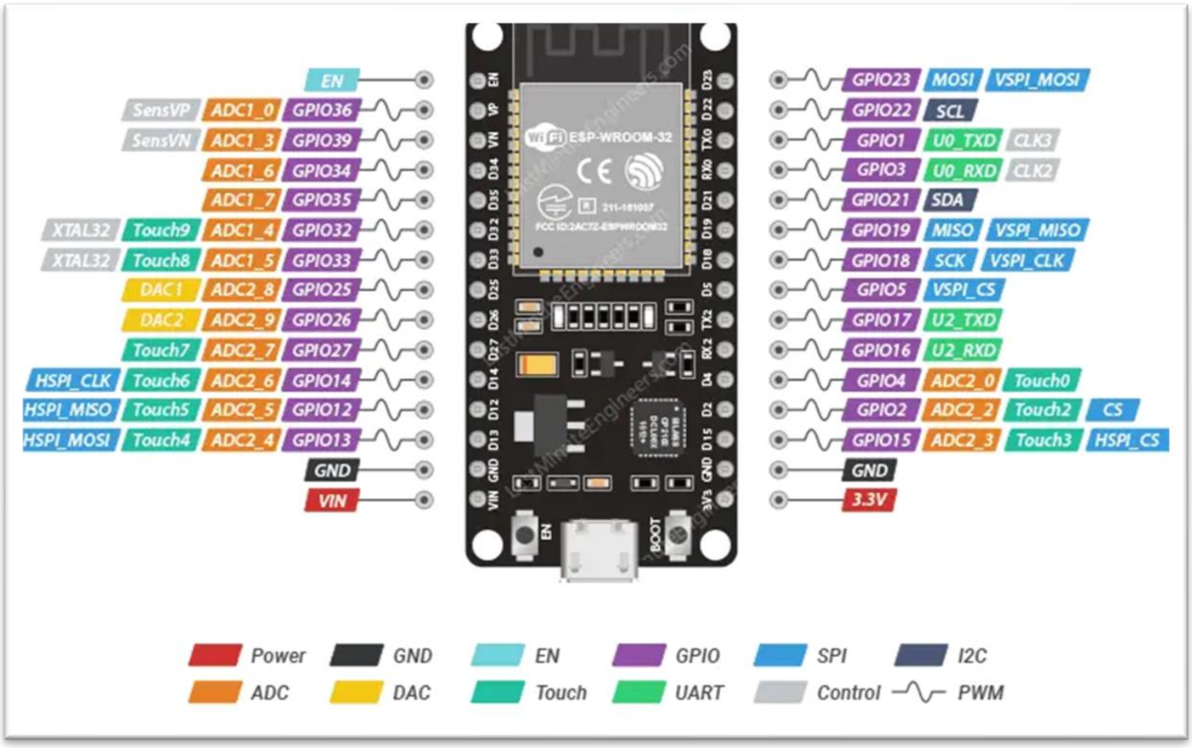


*Figure 2: ESP32 Pinout Reference*

- **STM32F4**

  The STM32F4 microcontroller acts as the data acquisition and control unit, interfacing with various sensors and modules in the system (e.g., temperature, speed, GPS). Its main role is to collect raw telemetry data, format it, and transmit it to the ESP32 over UART

## 2. 3 UART Communication Between ESP32 and STM32

- **Data Structure**

The ESP32 and STM32 exchange data through a UART interface at a baud rate of 9600 bps. The transmitted buffer follows the structured format:

```
<S>ID_1:32.5,ID_2:32.5,ID_3:28.0,ID_4:45.0,ID_5:30.0,ID_6:30.0<E>
```

- <S> : Start delimiter
- <E> : End delimiter
- ID_x : Parameter identifiers
- value : Numerical reading associated with the parameter

This format is flexible, allowing any number of IDs to be transmitted in a single frame.

ID Mapping

The following table shows the mapping between ID tags and their corresponding system parameters:

| ID | Parameter | Description |
|---|---|---|
| ID_1 | left_inverter_max_temp | Left inverter maximum temperature |
| ID_2 | right_inverter_max_temp | Right inverter maximum temperature |
| ID_3 | ambient_temperature | Ambient environmental temperature |
| ID_4 | car_speed_gauge | Vehicle speed reading |
| ID_5 | lat | GPS latitude |
| ID_6 | lon | GPS longitude |
| ID_7 | yaw_rate | Vehicle yaw rate |
| ID_8 | baro | Barometric pressure |
| ID_9 | heading_angle | Heading angle |
| ID_10 | heading_dir | Heading direction |
| ID_11 | left_wheel_spd | Left wheel speed |
| ID_12 | right_wheel_spd | Right wheel speed |
| ID_13 | left_mosfet_1 | Left inverter MOSFET #1 |
| ID_14 | left_mosfet_2 | Left inverter MOSFET #2 |
| ID_15 | left_mosfet_3 | Left inverter MOSFET #3 |
| ID_16 | left_motor_temp | Left motor temperature |
| ID_17 | right_mosfet_1 | Right inverter MOSFET #1 |
| ID_18 | right_mosfet_2 | Right inverter MOSFET #2 |
| ID_19 | right_mosfet_3 | Right inverter MOSFET #3 |
| ID_20 | right_motor_temp | Right motor temperature |

| | | |
|---|---|---|
| ID_21 | total_voltage | System voltage |
| ID_22 | total_current | System current |
| ID_23 | power_consumed | Power consumed |
| ID_24 | energy_consumed | Energy consumed |
| ID_25 | soc | State of charge (battery) |
| ID_26 | turnright | Right turn indicator |
| ID_27 | turnleft | Left turn indicator |
| ID_28 | lights_v1 | Vehicle high beam lighting status |

- ## Initialization and Synchronization

All IDs are initialized with default values in the ESP32. These default values persist until the STM32 transmits the real sensor values. Therefore, it is crucial for the STM32 to send all parameter values at system startup, ensuring that the cloud reflects actual system conditions instead of placeholder initialization values.

- ## UART Testing

UART functionality was validated through:

- GPS module integration: real latitude and longitude data were parsed and verified.
- Parameter consistency checks: verifying that the buffer format and values received on the ESP32 matched the STM32 transmission.

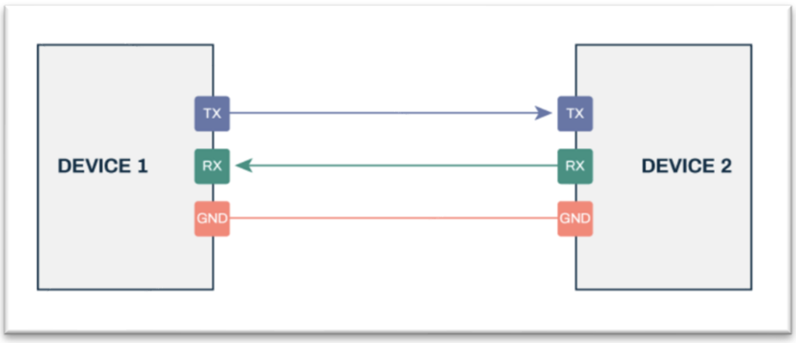


*Figure 3: Serial Communications Protocol (UART)*

## 2. 4 Wi-Fi Configuration on ESP32

The ESP32 has two main modes which define how the ESP will interact with other devices on a wireless network. These modes are:

- **Station Mode (STA)**

In this mode, the ESP32 acts like a client device. It connects to an existing Wi-fi Access point (AP) and obtains an ip address from the router in order to be able to communicate with other devices connected to the same network or access point.

Steps to connect to the ESP32 in STA mode:

- Configure the SSID (network name) and Password in the ESP32 firmware.
- Use WiFi.begin(ssid, password); in the code.
- The ESP32 will attempt to connect to the router and establish a link to the internet/cloud server.

This is the mode used in our system case as the ESP32 needs internet access to be able to upload the data from the STM to the Cloud.

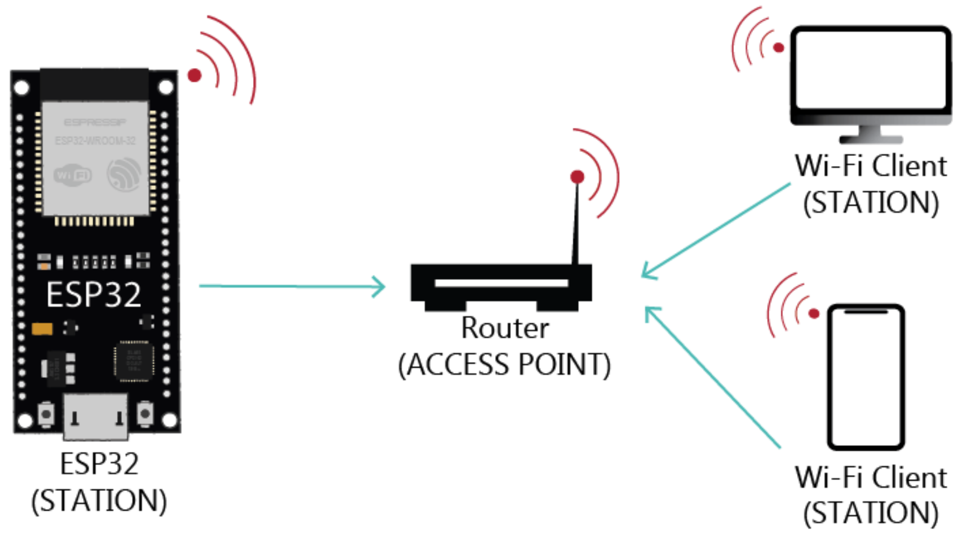


*Figure 4: Station Mode*

## • Access Point Mode (AP)

In this mode, the ESP32 itself becomes a Wi-Fi hotspot, other devices (smartphones, laptops, or even another ESP32) can connect directly to it without needing an external router. This mode is most likely used in IoT systems for local configurations or when no router is available.

Steps to connect to the ESP32 in AP mode:

- Set the ESP32 to AP mode with WiFi.softAP(ssid, password);.
- Devices can then scan for the SSID broadcasted by ESP32 and connect using the password.
- Useful for point-to-point communication but does not provide internet access unless bridged.

We are not going to use this configuration method as it doesn't provide internet connectivity unless an additional bridge is implemented while out system requires internet access to upload telemetry data to the cloud. The mode is useful for local data exchange but will be less practical for data streaming to an online server.

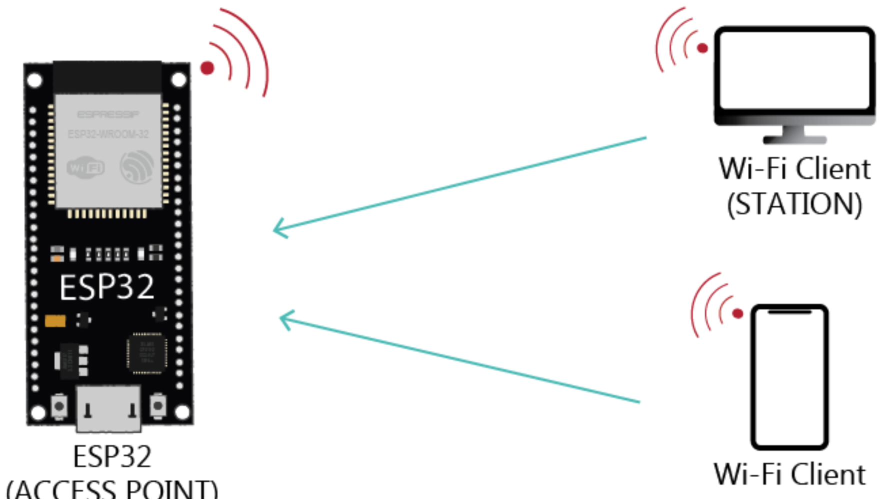


*Figure 5: Access Point Mode*

## 2. 5 Vehicle-to-Cloud System

- **Communication protocols and requirements (FTP, HTTP)**

**HTTP (Hypertext Transfer Protocol):**

- HTTP is the standard protocol used for web communication, where clients like browsers or IoT devices send requests and servers respond with data.
- It is widely used for APIs, dashboards, and small data exchanges.
- **Strengths:** Easy integration with web services, supports secure communication with HTTPS, scalable for request–response operations.
- **Limitations for our case:** Free web servers often impose strict limits on request rates and data size. To continuously upload large amounts of telemetry data at short intervals (sub-second updates), HTTP would require a paid hosting plan.

**FTP (File Transfer Protocol):**

- FTP is designed for transferring files between client and server.
- In our system, the ESP32 collects telemetry data, converts it into a JSON file (data.json), and uploads it to the server using FTP.
- The monitoring station or dashboard can then retrieve the latest file at regular intervals.
- **Strengths:** Simple to implement, supported by free hosting services, efficient for storing structured data files.
- **Limitations:** Less secure than HTTPS, not as flexible for real-time APIs.

## Why We Need These Protocols in Our System

In a **vehicle-to-cloud system**, the ESP32 must push sensor data from the car (using STM32 → ESP32) to an online server where it can be accessed by the monitoring dashboard and station software.

- Without a protocol like HTTP or FTP, there is no way to move telemetry from the car's embedded hardware to a remotely accessible storage location.
- HTTP could provide flexible API-like interactions, but would require paid server plans due to the high frequency of uploads.
- FTP allows us to repeatedly overwrite a single JSON file on the server (e.g. data.json) at short intervals without hitting usage limits on free hosting.

Therefore, **FTP** is chosen as the practical solution for our project since it ensures continuous data streaming to the cloud without additional costs.

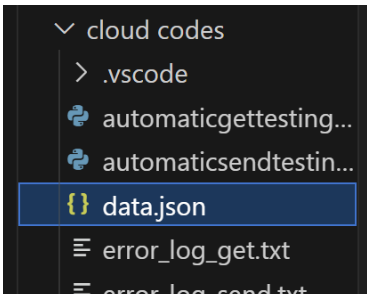


Figure 6: Cloud files

- **Server type and required specifications**

The server plays a critical role in storing and making telemetry data available for retrieval. For our system, we use a free FTP hosting service (ftpupload.net), which offers basic file storage and retrieval.

Server Requirements:

- **Type:** FTP server capable of handling frequent file overwrites.
- **Directory Structure**: Must allow storing telemetry under a defined folder.
- **File Management:** Must support replacing the data.json file multiple times per second without crashing or denying access.

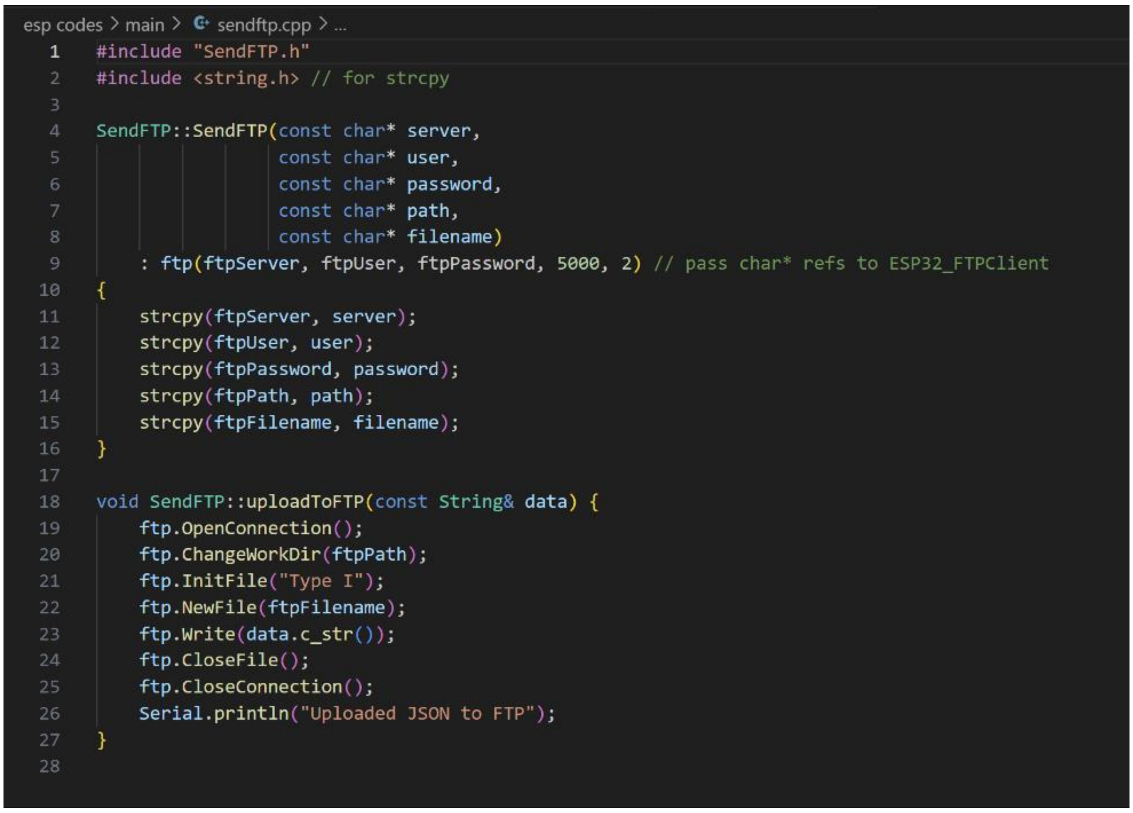
- **Bandwidth:** Should allow stable uploads/downloads for telemetry updates every 0.5 seconds or faster.
- **Reliability:** Server uptime must be sufficient for continuous monitoring during vehicle operation.

### Data transmission from ESP32 to server and retrieval (via PC and ESP32)

The data flow in the vehicle-to-cloud system is divided into two main processes: uploading telemetry data from the car to the server and retrieving the data at the monitoring station.

- **Transmission from ESP32 to Server**

- The STM32 microcontroller continuously collects telemetry values (e.g., temperatures, vehicle speed, GPS, voltage, current).
- These values are sent to the ESP32 over UART in a structured data frame.
- The ESP32 parses the UART buffer and converts the values into a JSON file (**data.json**).
- Using the FTP protocol, the ESP32 uploads data.json to the server directory (**/htdocs/new/**).
- Each new upload overwrites the previous file, ensuring that the server always stores the most recent vehicle state.



esp codes > main > sendftp.cpp > ...

```cpp
#include "SendFTP.h"
#include <string.h> // for strcpy

SendFTP::SendFTP(const char* server,
                 const char* user,
                 const char* password,
                 const char* path,
                 const char* filename)
    : ftp(ftpServer, ftpUser, ftpPassword, 5000, 2) // pass char* refs to ESP32_FTPClient
{
    strcpy(ftpServer, server);
    strcpy(ftpUser, user);
    strcpy(ftpPassword, password);
    strcpy(ftpPath, path);
    strcpy(ftpFilename, filename);
}

void SendFTP::uploadToFTP(const String& data) {
    ftp.OpenConnection();
    ftp.ChangeWorkDir(ftpPath);
    ftp.InitFile("Type I");
    ftp.NewFile(ftpFilename);
    ftp.Write(data.c_str());
    ftp.CloseFile();
    ftp.CloseConnection();
    Serial.println("Uploaded JSON to FTP");
}
```

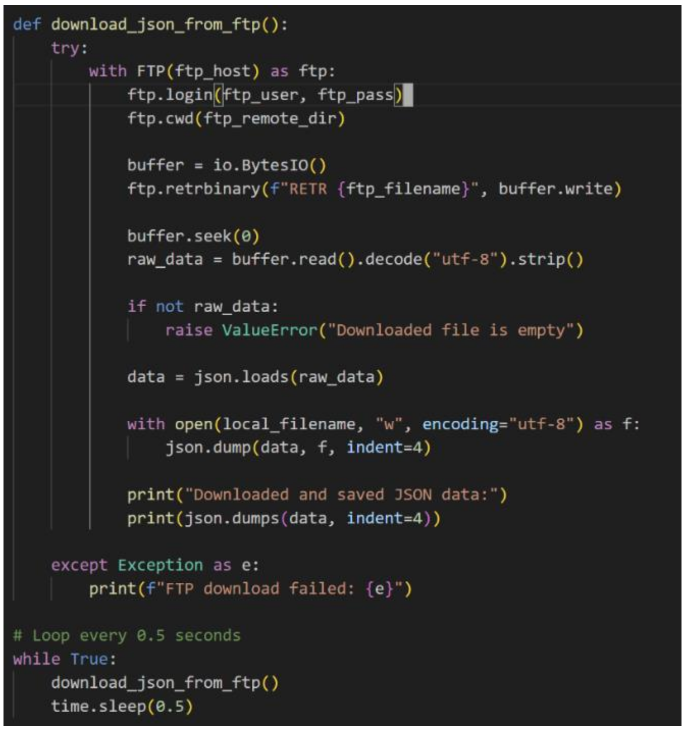
*Figure 7: SendFTP Car Side*

- This process is repeated periodically at intervals ≥0.5 seconds.

## • Retrieval on the PC

On the monitoring station, a Python script (**getftp.py**) is responsible for continuously retrieving the telemetry file from the FTP server.

- The script connects to the FTP host (ftpupload.net) using valid credentials and navigates to the remote directory (**/htdocs/new**).
- It downloads the latest version of data.json at regular intervals (every 0.5 seconds in the current implementation).
- Each retrieval cycle overwrites the local copy of data.json, ensuring that the PC always has access to the most up-to-date telemetry values coming from the vehicle.
- The file content is parsed from raw **JSON** into a Python dictionary, which can be logged, validated, or pre-processed before use.



```python
def download_json_from_ftp():
    try:
        with FTP(ftp_host) as ftp:
            ftp.login(ftp_user, ftp_pass)
            ftp.cwd(ftp_remote_dir)

            buffer = io.BytesIO()
            ftp.retrbinary(f"RETR {ftp_filename}", buffer.write)

            buffer.seek(0)
            raw_data = buffer.read().decode("utf-8").strip()

            if not raw_data:
                raise ValueError("Downloaded file is empty")

            data = json.loads(raw_data)

            with open(local_filename, "w", encoding="utf-8") as f:
                json.dump(data, f, indent=4)

            print("Downloaded and saved JSON data:")
            print(json.dumps(data, indent=4))

    except Exception as e:
        print(f"FTP download failed: {e}")

# Loop every 0.5 seconds
while True:
    download_json_from_ftp()
    time.sleep(0.5)
```

*Figure 8: getftp.py*

- Once downloaded and saved locally, the JSON data is made available for the dashboard web application, which parses and displays parameters such as: temperatures, speed, GPS location, and power system metrics in real time.

This automated retrieval process guarantees continuous synchronization between the car-side ESP32 uploads and the station-side monitoring app, enabling real-time visualization of vehicle performance.

- **Additional Files:**

- **Sendftp_station.py** - A small utility that **manually updates driver.json** on the same FTP directory (/htdocs/new) to reflect the **number of laps** reported by a team member outside the car. It builds a one-element JSON array (example: {"number_of_labs": 7} and uploads it every **0.5 s**, so the dashboard's "Number of labs" widget stays in sync.

- **Sendftp.py** - A station-side tool for **end-to-end testing**. It synthesizes realistic telemetry (temperatures, speed, GPS, power, SoC, etc.) and **uploads data.json** to /htdocs/new via FTP on a short interval. This verifies that the **frontend updates correctly** without needing the car online.

## 2. 6 Car Dashboard Web Application

- **Programming languages used**

The car dashboard web application was developed using a combination of web-based technologies:

- **PHP** – Handles server-side logic, file access, and rendering of telemetry data from data.json.
- **HTML & CSS** – Provide the structural layout and styling of the dashboard interface.
- **JavaScript** – Enables interactivity and real-time updates of parameters displayed on the dashboard.

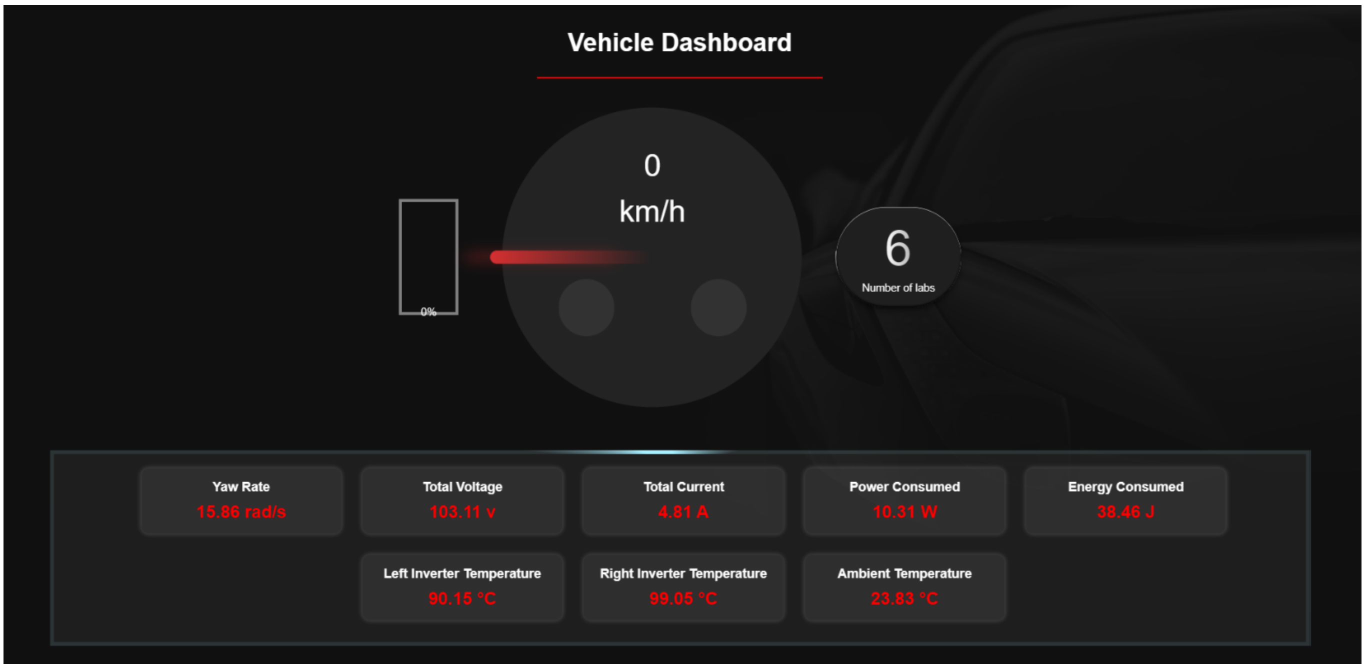
This lightweight stack ensures that the dashboard can be hosted on the same server that stores telemetry data, requiring no additional frameworks or paid hosting solutions.

- **File structure of the web application code**

- **index.php** - Main entry point; loads and displays the dashboard interface by parsing the latest data.json file.
- **style.css** - Core stylesheet responsible for layout design, colors, and formatting of dashboard components.
- **data.json** - The file downloaded from the FTP server that stores the values sent from the ESP.

- **driver.json** – The file downloaded from the FTP server that stores the value of the number of laps which is updated manually by a team member outside the vehicle.

  - **Parameters displayed on the dashboard**



| Yaw Rate | Total Voltage | Total Current | Power Consumed | Energy Consumed |
|---|---|---|---|---|
| 15.86 rad/s | 103.11 v | 4.81 A | 10.31 W | 38.46 J |

| Left Inverter Temperature | Right Inverter Temperature | Ambient Temperature |
|---|---|---|
| 90.15 °C | 99.05 °C | 23.83 °C |

*Figure 9: Vehicle Dashboard*

| | | | |
|---|---|---|---|
| *Speed (km/h)* | *Total Voltage (V)* | *Total Current (A)* | *Power Consumed (W)* |
| *Energy Consumed (J)* | *Left Inverter Temperature (°C)* | *Right Inverter Temperature (°C)* | *Ambient Temperature (°C)* |
| *Yaw Rate (rad/s)* | *Battery Percentage (%)* | *Number of laps Completed (Obtained from driver.json file)* | |

## Additional Parameters displayed in the Software Application:

- Date
- Lateral rate
- Longitudinal rate
- Baro
- Heading angle
- Heading direction
- Left wheel speed
- Right wheel speed
- Left mosfet 1
- Left mosfet 2
- Left mosfet 3
- Left motor temperature
- Right mosfet 1
- Right mosfet 2
- Right mosfet 3
- Right motor temperature

# 3. Running the System

The system consists of two main parts: the Car Side (where the ESP32 and STM32 operate together to collect and transmit data) and the Station Side (where the received data is retrieved from the cloud and made available for monitoring).

## 3. 1 Car Side

On the car side, the ESP32 is responsible for communication with the cloud, while the STM32F4 collects telemetry values from sensors and subsystems.

- **Wi-Fi Connectivity**
  - The ESP32 must connect to a Wi-Fi network in order to upload telemetry data.
  - The Wi-Fi SSID (username) and password must be hard-coded in the ESP32 firmware before deployment.
  - A stable internet speed of at least 30 Mbps is required to ensure reliable and continuous cloud communication.
- **Communication Rate**
  - The communication rate between the ESP32 and STM32 must be set to greater than 0.5 seconds per update for reliable operation with the free cloud server.
  - While the ESP32 can technically receive data at any rate, sending too many updates to the server at a fast pace may exceed the limits of the free hosting plan.
  - Ongoing work is focused on reducing this delay to 0.2 seconds for improved responsiveness.

- **Data Upload Procedure**

Once the ESP32 is connected to Wi-Fi:

1. The STM32 transmits telemetry values over UART.
2. The ESP32 receives the data, parses it, and structures it into a JSON format.
3. The formatted JSON file (data.json) is automatically uploaded to the server via FTP.

- **Configurable FTP Parameters**

The FTP server details are defined in the ESP32 firmware inside JSONParser.cpp, specifically in the uploadToFTP function. The following object initializes the FTP connection:

```
SendFTP ftpSender(
"ftpupload.net",
"if0_38887836",
"uZnrqJ55D3nN",
"/htdocs/new",
"data.json"
);
```

Explanation of parameters:

- "ftpupload.net" → FTP server domain used to host the uploaded files.
- "if0_38887836" → FTP username used for authentication.
- "uZnrqJ55D3nN" → FTP password paired with the username for secure login.
- "/htdocs/new" → Remote directory path on the server where the file will be uploaded.
- "data.json" → Filename of the JSON file that contains all updated telemetry values.

By modifying these parameters in the ESP32 code, the system can be easily redirected to another FTP server.

## 3. 2 Station Side

On the station side, a Python script is used to fetch data from the cloud and make it available locally for monitoring and further processing.

- **Python Script: getftp.py**
  - This script connects to the configured FTP server, downloads the latest telemetry file, and stores it as data.json on the local machine.
  - The data.json file contains all sensor and system variables received from the cloud, reflecting the most recent state of the vehicle.
- **Configurable Delay**
  - The script includes a download delay parameter that controls how often new data is retrieved from the cloud.

- It is recommended to set the delay to **greater than 0.5 seconds** when using the free server to avoid overloading the service.
- A shorter delay (e.g., ~0.2 seconds) may be feasible on higher-performance servers, but on free hosting, it risks data congestion and failed download.

## 4. References & Appendix

1. ESP32 Technical Reference Manual – Espressif Systems [https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf]
2. STM32F4 Series Reference Manual – STMicroelectronics [https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf]
3. UART Protocol Overview – GeeksforGeeks [https://www.geeksforgeeks.org/computer-networks/universal-asynchronous-receiver-transmitter-uart-protocol/]
4. File Transfer Protocol (FTP) – IETF RFC 959 [https://www.rfc-editor.org/rfc/rfc959]
5. Python Documentation: ftplib – FTP Client Library [https://docs.python.org/3/library/ftplib.html]
6. Team's Repository on GitHub [https://github.com/aliadelmahdi/Racing-Team-Tasks]
7. Server Account [https://dash.infinityfree.com/accounts]
8. Dashboard [https://racingteam.rf.gd/]
9. Online JSON Data [https://racingteam.rf.gd/new/data.json]