Technical Report

June 2025

Velocity: IoT Performance Tracker

Ilyass Souhail

SAMK

Data Engineering

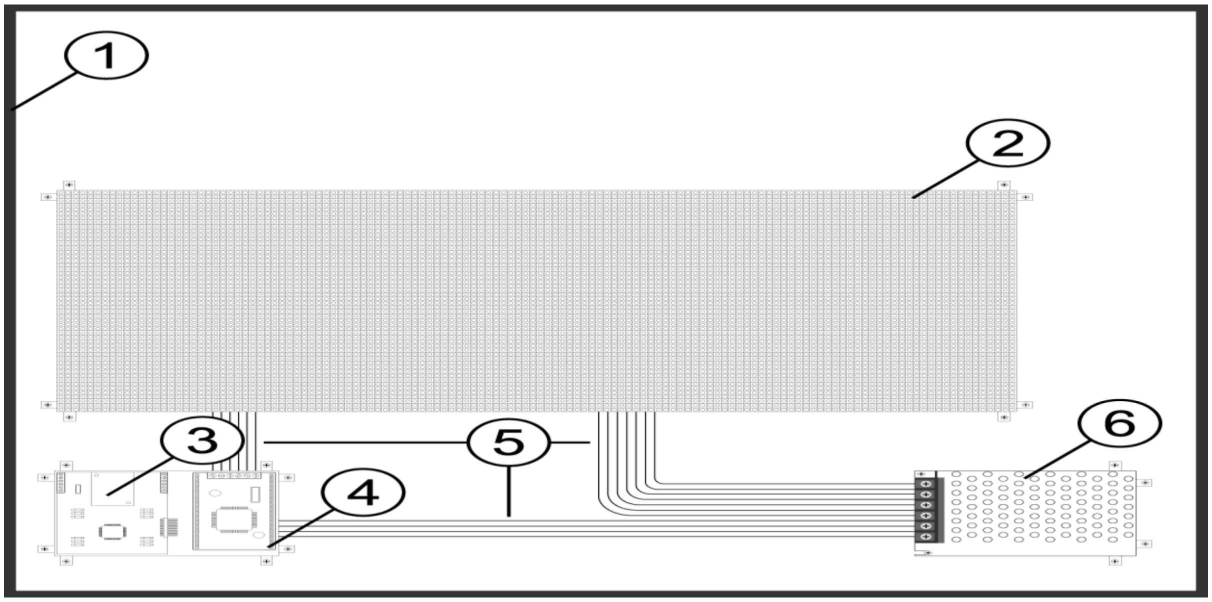
1. Contents

2. Hardware Review:

The interactive outdoor speedometer display is intended to measure and display walkers jogging speeds with a high degree of accuracy (two decimals). It is shielded by a sturdy housing with passive ventilation, has a microprocessor for power, and employs a P5 LED matrix for clear visual output. After eliminating unimportant targets, an FMCW radar sensor uses Doppler shift to determine speed and detect motion. The display offers both standard and custom settings and is made to be easily mounted in urban areas.

The microcontroller is a component inside the metal casing, it's a small computer in a chip that reads sensors in the case of speedometer reads FMCW radar sensor, Furthermore, It's the chip runs programs to process data. Basically, controls LED display to show speed.

Components inside the metal casing



Number 4 component is what microcontroller, can be an Arduino Nano, ESP32, or STM32, depending on requirements for processing power, connectivity, and power consumption.

3. Retrieving data Methods:

Following research into the best method for retrieving speed data, it is necessary to discuss three approaches that can achieve the desired outcome. Each approach has advantages and disadvantages based on several factors.

3.1. SD Card logging:

This method saves speed data directly onto an SD card connected to the microcontroller. It stores the information locally so it can be accessed later by removing the card. This way, data is safely recorded even without internet or wireless connection.

It is user-friendly, works without internet or Wi-Fi, and ensures data safety even when power is out, but requires physically removing the SD card for access and lacks live or remote access.

3.2. Wi-Fi enabled board:

This method uses a microcontroller with built-in Wi-Fi to send speed data directly to the internet or a server. It allows real-time access to the data from anywhere without needing to remove any storage device.

Pros include instant data access, no physical retrieval, live monitoring, and alerts, but require stable internet connection, complex setup, and use more power than offline methods.

3.3. Camera + OCR:

This approach uses a camera to capture images or video of the speed displayed on the device. Then, software reads the digits using Optical Character Recognition (OCR) to convert the visuals into data. It doesn’t require any changes to the speedometer hardware.

It doesn't require modification, can be set up quickly with existing cameras, and is useful when hardware access is limited, but accuracy can be affected by glare, motion blur, or lighting, and requires additional software.

4. Evolution of Data retrieval methods:

The selection of the optimal data retrieval method for the speedometer display requires a thorough evaluation of practical factors like implementation ease, data accessibility, reliability, power consumption, and accuracy, to identify their strengths and limitations.

4.1. Comparison:

Evidently to know which way is the best and suitable for retrieving data from speedometer, the following table shows the differences of implementing the 3 ways by several aspects.

Table 1: Methods Comparison

Aspect	SD Card Logging	Wi-Fi Enabled Board	Camera + OCR
Ease of Setup	Easy	Medium	Easy
Cost	Low	Medium	Low (if you have a camera)
Data Access	Offline, manual retrieval	Real-time, remote access	Offline, after processing images
Reliability	Very reliable (no internet needed)	Depends on Wi-Fi and server	Depends on image quality
Power Usage	Low	Higher (Wi-Fi uses power)	Medium (camera + processing)
Accuracy	High	High	Medium (OCR errors possible)
Scalability	Moderate (local data only)	High (multiple devices easy)	Low (manual setup per device)
Hardware Impact	Requires adding SD module	May require replacing board	No hardware changes needed

4.2. Verdict

Generally, the SD Card logging is more stable and less expensive either in computing or cost also in energy usage. Nevertheless, the data won't be scalable, no streaming data yet no need for real time or service provider.

There is also a wide issue concerning identifying joggers, therefore using wireless board is efficient in the matter only if the users wear tags, the SD Card can still serve as backup.

5. IoT Ecosystem:

This section highlights the operational efficiency and individualised performance feedback of an interactive speedometer display system by outlining its theoretical architecture, components, data flow, and user interface accessibility.

5.1. Microcontroller:

Since the need of integrating both Wi-Fi and Bluetooth Low energy (BLE), the ESP32 is the most suitable microcontroller for data retrieval due to its sufficient power for sensor processing and network tasks and necessary for outdoor, battery-operated devices.

The ESP32 is a good choice because it is cheap and has a strong development ecosystem. It offers powerful features at a low price, with documentation, libraries, and a large online community.

5.2. Data server/Cloud Platform:

With a generous free tier for development and smaller projects, Firebase is an affordable cloud platform with strong real-time capabilities. It is perfect for IoT integration because of its Realtime Database and Cloud Firestore, which allow for real-time speed updates. Unlike AWS IoT, Firebase is a strong contender for sending data from devices like the ESP32 because of its ease of development and IoT integration.

5.3. UI Platform:

Compared to writing separate apps for each platform, Flutter's ability to write code once and deploy to both iOS and Android platforms saves development time and money, making it perfect for creating native mobile user interfaces.

The Dart programming language, a cross-platform, adaptable widget-based system, hot reload, outstanding performance, and pixel-perfect control for a consistent user interface across various devices are all features of Google's Flutter, an open-source UI software development kit.

5.4. Identification sensor:

BLE tags are tiny, low-power gadgets that send out distinct signals. Every jogger wears a tag that transmits a distinct ID. These signals are scanned and detected by the ESP32's integrated BLE capability, which enables the system to connect speed data to the jogger. These tags are appropriate for prolonged use because they use little power.

6. Data Acquisition and Cloud Transmission:

This chapter examines the core functions of the system, such as data gathering and cloud integration, as well as the mechanisms used by the edge device to identify joggers and measure speed accurately.

6.1. FMCW to data:

Through physical connections, such as the ESP32's GPIO pins, the ESP32 microcontroller receives speed data from the FMCW radar sensor. These pins serve as communication lines, and a particular protocol is used to send the speed data. SPI (Serial Peripheral Interface) and UART (Universal Asynchronous Receiver-Transmitter) are common protocols.

Table 2: SPI pins

SPI	MOSI	MISO	CLK	CS
VSPI	GPIO 23	GPIO 19	GPIO 18	GPIO 5
HSPI	GPIO 13	GPIO 12	GPIO 14	GPIO 15

Table 3: UART pins

UART	TX	RX
UART0	GPIO 1	GPIO 3
UART1	GPIO 10	GPIO 9
UART2	GPIO 17	GPIO 16

The ESP32 firmware continuously watches the data stream from the radar and listens for incoming data on the assigned GPIO pins shown in tables above. To guarantee a steady and accurate display, the firmware can handle transient readings, filter out noise, average multiple readings, and translate new speed values into a useful numerical format.

6.2. BLE Identification:

The system broadcasts distinct identification signals using BLE tags to pinpoint individual joggers. A tiny, low-power BLE tag that broadcasts individual IDs is worn by every jogger. The ESP32 microcontroller actively listens for these advertising packets and extracts unique IDs thanks to its integrated BLE module. The ESP32 then compares the speed determined by the FMCW radar with the detected jogger ID. The firmware of the ESP32 determines which BLE tag is most pertinent, guaranteeing precise attribution to the jogger.

6.3. Cloud Transmission:

The ESP32 microcontroller is a device that uses a BLE tag to identify joggers and gathers speed data from an FMCW radar sensor. This data is then sent to the Firebase cloud platform for storage and access. The firmware compiles the data into a structured payload, typically JSON, which is widely supported for cloud services and device communication. The device has a built-in Wi-Fi module with necessary network credentials. The ESP32 encrypts sensitive data using HTTPS during transit. A Firebase Cloud Function processes the data and stores it in Firestore, ensuring real-time data transfer.

6.4. Data Correlation and Timestamping:

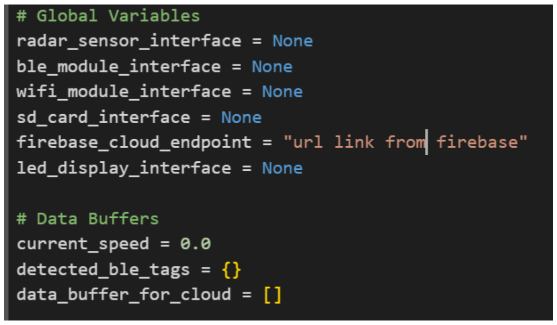
This step is like noting when something happened and then matching a picture to its caption. To determine which jogger the speed it just measured from the radar belongs to, the ESP32 first examines the BLE tags in the area. It selects what appears to be the most likely tag, possibly the closest one. The ESP32 simultaneously appends a precise time and date stamp to this aggregated data. It now has a comprehensive record, such as "Jogger A ran at 8 km/h at 10:00 AM," rather than just a speed number and the jogger's ID separately. Accurate performance tracking and sending relevant data to the cloud depend on this combined and time-stamped piece of data.

6.5. Conceptual Firmware Logic:

This section elaborates on the intelligent core of the ESP32 microcontroller: its firmware. The ESP32's ability to coordinate the intricate processes of sensor data collection, processing, jogger identification, data correlation, and secure cloud transmission is explained in the high-level operational design. To understand the system's autonomous functionality and its capacity to effectively handle various data streams at the edge, one must have a solid understanding of this conceptual logic.

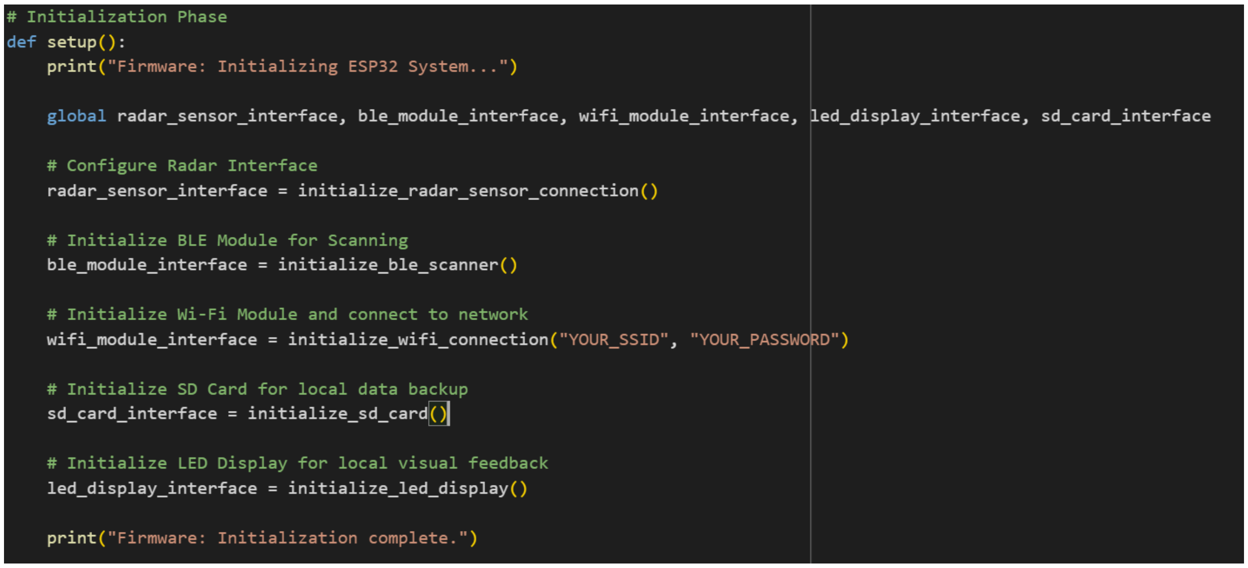
That been said this part includes an explaining of a conceptual pseudocode in python language instead of C or C++.

Figure 1: the setup code



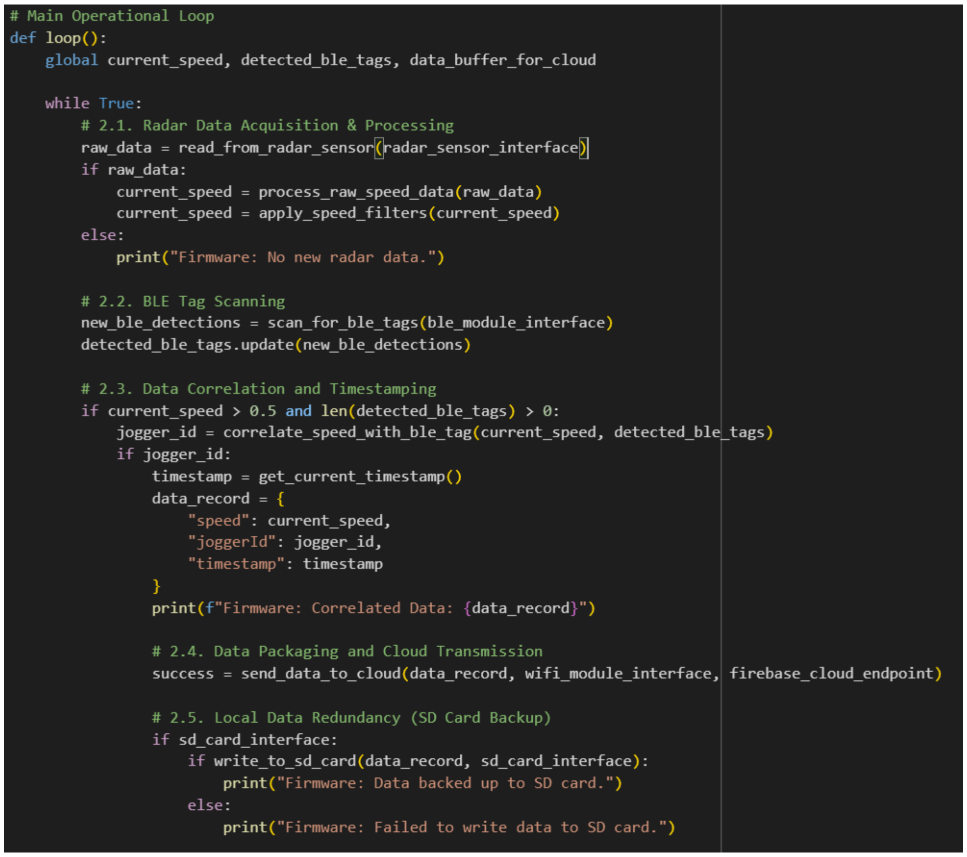
```
# Global Variables
radar_sensor_interface = None
ble_module_interface = None
wifi_module_interface = None
sd_card_interface = None
firebase_cloud_endpoint = "url link from firebase"
led_display_interface = None

# Data Buffers
current_speed = 0.0
detected_ble_tags = {}
data_buffer_for_cloud = []
```

Figure 2: setup () function



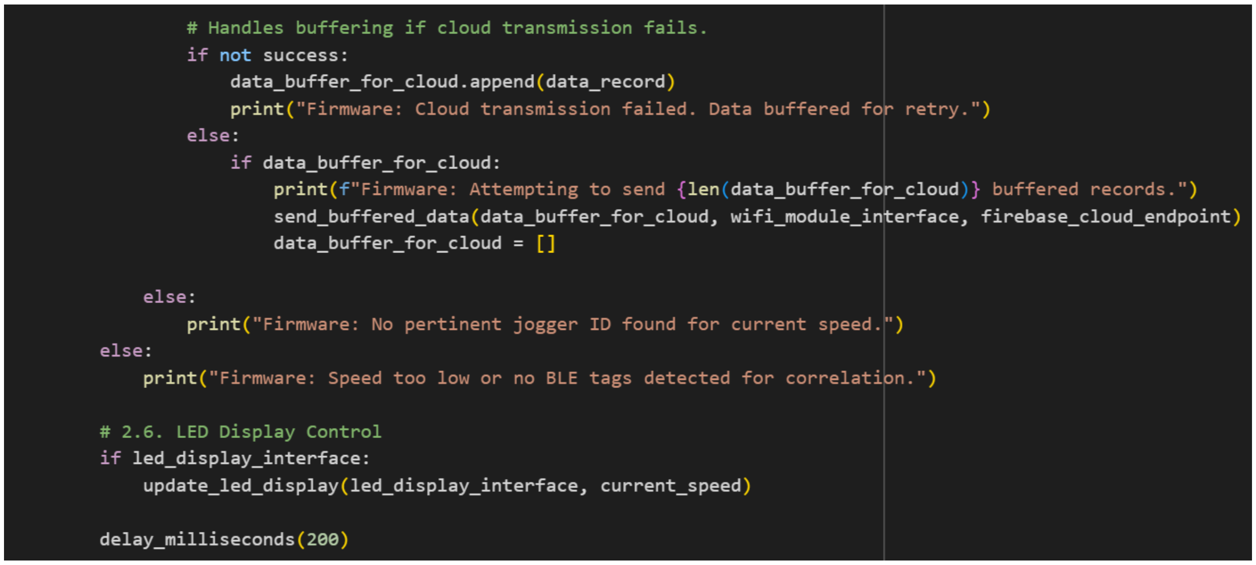
```
# Initialization Phase
def setup():
    print("Firmware: Initializing ESP32 System...")

    global radar_sensor_interface, ble_module_interface, wifi_module_interface, led_display_interface, sd_card_interface

    # Configure Radar Interface
    radar_sensor_interface = initialize_radar_sensor_connection()

    # Initialize BLE Module for Scanning
    ble_module_interface = initialize_ble_scanner()

    # Initialize Wi-Fi Module and connect to network
    wifi_module_interface = initialize_wifi_connection("YOUR_SSID", "YOUR_PASSWORD")

    # Initialize SD Card for local data backup
    sd_card_interface = initialize_sd_card()

    # Initialize LED Display for local visual feedback
    led_display_interface = initialize_led_display()

    print("Firmware: Initialization complete.")
```

Figure 3: loop function 1



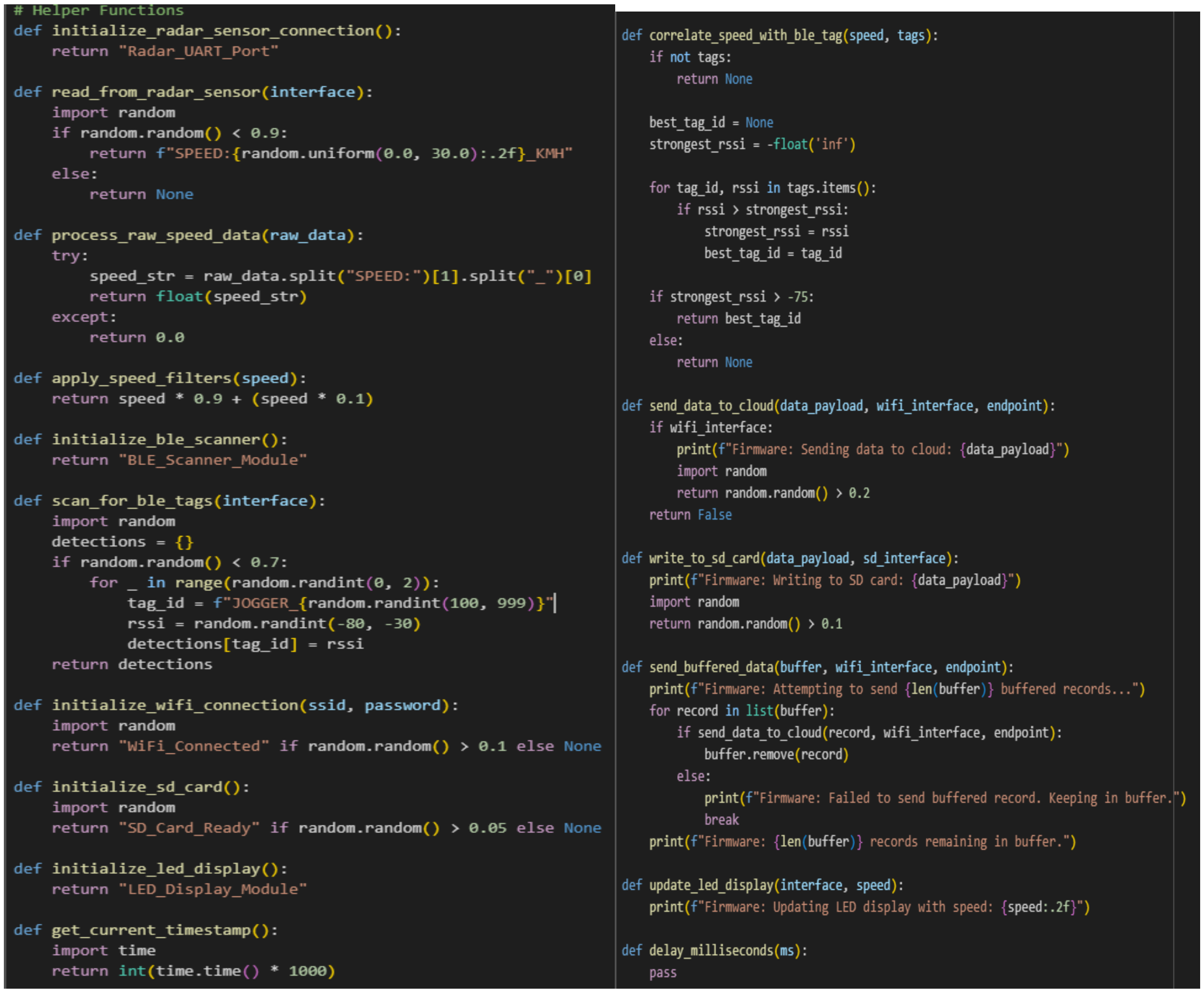
```
# Main Operational Loop
def loop():
    global current_speed, detected_ble_tags, data_buffer_for_cloud

    while True:
        # 2.1. Radar Data Acquisition & Processing
        raw_data = read_from_radar_sensor(radar_sensor_interface)
        if raw_data:
            current_speed = process_raw_speed_data(raw_data)
            current_speed = apply_speed_filters(current_speed)
        else:
            print("Firmware: No new radar data.")

        # 2.2. BLE Tag Scanning
        new_ble_detections = scan_for_ble_tags(ble_module_interface)
        detected_ble_tags.update(new_ble_detections)

        # 2.3. Data Correlation and Timestamping
        if current_speed > 0.5 and len(detected_ble_tags) > 0:
            jogger_id = correlate_speed_with_ble_tag(current_speed, detected_ble_tags)
            if jogger_id:
                timestamp = get_current_timestamp()
                data_record = {
                    "speed": current_speed,
                    "joggerId": jogger_id,
                    "timestamp": timestamp
                }
                print(f"Firmware: Correlated Data: {data_record}")

                # 2.4. Data Packaging and Cloud Transmission
                success = send_data_to_cloud(data_record, wifi_module_interface, firebase_cloud_endpoint)

                # 2.5. Local Data Redundancy (SD Card Backup)
                if sd_card_interface:
                    if write_to_sd_card(data_record, sd_card_interface):
                        print("Firmware: Data backed up to SD card.")
                    else:
                        print("Firmware: Failed to write data to SD card.")
```

Figure 4: loop function 2



```
                # Handles buffering if cloud transmission fails.
                if not success:
                    data_buffer_for_cloud.append(data_record)
                    print("Firmware: Cloud transmission failed. Data buffered for retry.")
                else:
                    if data_buffer_for_cloud:
                        print(f"Firmware: Attempting to send {len(data_buffer_for_cloud)} buffered records.")
                        send_buffered_data(data_buffer_for_cloud, wifi_module_interface, firebase_cloud_endpoint)
                        data_buffer_for_cloud = []

            else:
                print("Firmware: No pertinent jogger ID found for current speed.")
        else:
            print("Firmware: Speed too low or no BLE tags detected for correlation.")

        # 2.6. LED Display Control
        if led_display_interface:
            update_led_display(led_display_interface, current_speed)

        delay_milliseconds(200)
```

Figure 5: Helper functions



```
# Helper Functions
def initialize_radar_sensor_connection():
    return "Radar_UART_Port"

def read_from_radar_sensor(interface):
    import random
    if random.random() < 0.9:
        return f"SPEED:{random.uniform(0.0, 30.0):.2f}_KMH"
    else:
        return None

def process_raw_speed_data(raw_data):
    try:
        speed_str = raw_data.split("SPEED:")[1].split("_")[0]
        return float(speed_str)
    except:
        return 0.0

def apply_speed_filters(speed):
    return speed * 0.9 + (speed * 0.1)

def initialize_ble_scanner():
    return "BLE_Scanner_Module"

def scan_for_ble_tags(interface):
    import random
    detections = {}
    if random.random() < 0.7:
        for _ in range(random.randint(0, 2)):
            tag_id = f"JOGGER_{random.randint(100, 999)}"
            rssi = random.randint(-80, -30)
            detections[tag_id] = rssi
    return detections

def initialize_wifi_connection(ssid, password):
    import random
    return "WiFi_Connected" if random.random() > 0.1 else None

def initialize_sd_card():
    import random
    return "SD_Card_Ready" if random.random() > 0.05 else None

def initialize_led_display():
    return "LED_Display_Module"

def get_current_timestamp():
    import time
    return int(time.time() * 1000)
```

```
def correlate_speed_with_ble_tag(speed, tags):
    if not tags:
        return None

    best_tag_id = None
    strongest_rssi = -float('inf')

    for tag_id, rssi in tags.items():
        if rssi > strongest_rssi:
            strongest_rssi = rssi
            best_tag_id = tag_id

    if strongest_rssi > -75:
        return best_tag_id
    else:
        return None

def send_data_to_cloud(data_payload, wifi_interface, endpoint):
    if wifi_interface:
        print(f"Firmware: Sending data to cloud: {data_payload}")
        import random
        return random.random() > 0.2
    return False

def write_to_sd_card(data_payload, sd_interface):
    print(f"Firmware: Writing to SD card: {data_payload}")
    import random
    return random.random() > 0.1

def send_buffered_data(buffer, wifi_interface, endpoint):
    print(f"Firmware: Attempting to send {len(buffer)} buffered records...")
    for record in list(buffer):
        if send_data_to_cloud(record, wifi_interface, endpoint):
            buffer.remove(record)
        else:
            print(f"Firmware: Failed to send buffered record. Keeping in buffer.")
            break
    print(f"Firmware: {len(buffer)} records remaining in buffer.")

def update_led_display(interface, speed):
    print(f"Firmware: Updating LED display with speed: {speed:.2f}")

def delay_milliseconds(ms):
    pass
```

Before the ESP32 starts its job of monitoring joggers and sending data, the setup step makes sure that all the "tools" and "connections" are set up correctly and ready to go.

Setting up the ESP32's hardware means getting the internal parts and peripherals ready, giving them resources, and doing one-time tasks when the device starts up.

The figure 3 displays the ESP32's primary operational loop, which is always running. Radar data collection and processing, BLE tag scanning, data correlation and timestamping, data packaging and cloud transmission, and local data redundancy are all included. Utilising the radar sensor's raw data, the system looks for Bluetooth Low Energy tags, compares the measured speed to the most pertinent jogger ID, and then transmits the information to the Firebase cloud.

The firmware's reasoning for controlling data flow after it has been collected and processed, determining what to do with it (send now, buffer for later, or discard if irrelevant), and giving the user constant feedback is displayed across the screen.

Initialising the radar sensor connection, reading and processing raw speed data, applying speed filters, initialising the Bluetooth Low Energy (BLE) module, initialising the Wi-Fi connection, initialising the SD card, initialising the LED display, and obtaining the current timestamp are among the tasks performed by the ESP32. These features make it easy to comprehend how the ESP32 interacts with storage, communication modules, and sensors.

Correlate_speed_with_ble_tag, **send_data_to_cloud**, **write_to_sd_card**, **send_buffered_data**, **update_led_display**, and **delay_milliseconds** are among the features of the ESP32 firmware.

In addition to providing a short break in the main loop, these features assist in connecting measured speed to joggers, sending data to the Firebase cloud, writing data to an SD card for local backup, managing retrying data transmission, and updating the physical LED display.

6.6. Local Backup:

Although the Firebase cloud is the main destination for data transmission, SD card logging is considered for local backup and data redundancy. This would act as a backup, guaranteeing that data is securely captured even in situations where internet connectivity is momentarily interrupted or unavailable. The firmware for the ESP32 could be made to write speed and jogger ID data to the local SD card and the cloud at the same time, offering a reliable backup system that could be retrieved later if necessary.

7. Cloud Data Model

For the mobile application to query and display the ESP32's raw data stream, this chapter converts it into a structured, easily accessible, and secure database.

7.1. Data Storage:

Firebase Firestore is a NoSQL document database that organizes data into flexible documents and collections, providing real-time updates and automatic scaling for mobile app use. Its flexible structure allows for easy organization changes.

Firebase Firestore is a cloud-based filing cabinet that organizes data into individual files, like a filing cabinet. When a ESP32 measures a speed, it creates a **data_record** with speed, ID, and timestamp, which is sent to Firebase. Firebase automatically stores the new file in the "Speed Readings" drawer. The Flutter mobile app is connected to this cloud filing cabinet, and Firestore notifies the app of new data, ensuring real-time updates are automatically received.

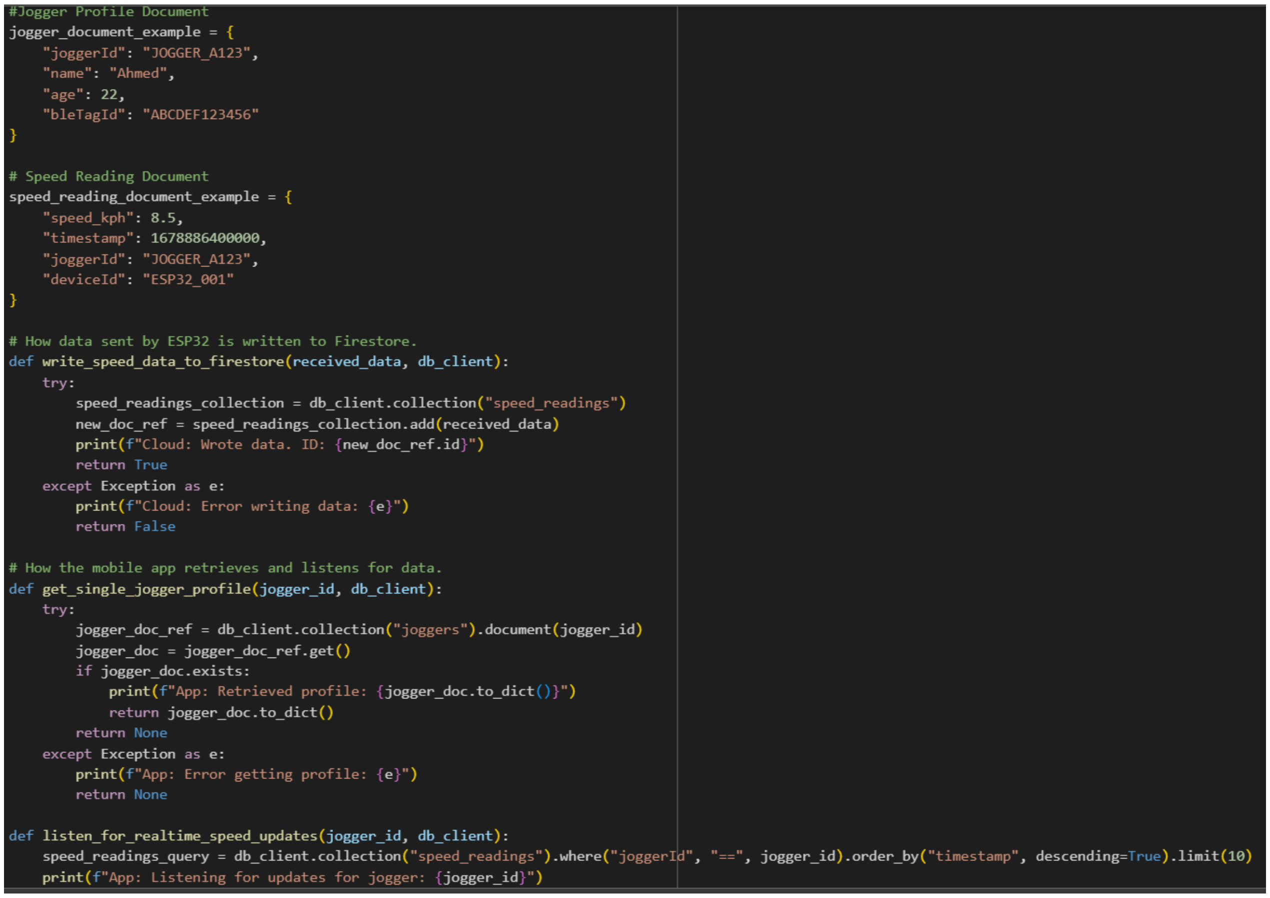
7.2. Data Model Design and Schema:

This section discusses the design of the Firebase Firestore data storage solution, outlining its structure, collections, and documents for efficient storage, retrieval, and relationship management of speed and jogger identification data.

The design consists of collections, which are top-level containers in Firestore, housing documents of similar types. Documents are individual records within a collection, represented as flexible JSON-like objects. Each document encapsulates all relevant attributes for a single entity, such as a jogger profile or a speed measurement event. Fields are key-value pairs within each document that hold data pieces. Relationships are how different documents are conceptually linked to each other.

The section outlines the conceptual data structure for Firestore, focusing on two main types of documents: profiles for joggers and **speed_readings**.

Figure 6: Cloud Data Model



```python
#Jogger Profile Document
jogger_document_example = {
    "joggerId": "JOGGER_A123",
    "name": "Ahmed",
    "age": 22,
    "bleTagId": "ABCDEF123456"
}

# Speed Reading Document
speed_reading_document_example = {
    "speed_kph": 8.5,
    "timestamp": 1678886400000,
    "joggerId": "JOGGER_A123",
    "deviceId": "ESP32_001"
}

# How data sent by ESP32 is written to Firestore.
def write_speed_data_to_firestore(received_data, db_client):
    try:
        speed_readings_collection = db_client.collection("speed_readings")
        new_doc_ref = speed_readings_collection.add(received_data)
        print(f"Cloud: Wrote data. ID: {new_doc_ref.id}")
        return True
    except Exception as e:
        print(f"Cloud: Error writing data: {e}")
        return False

# How the mobile app retrieves and listens for data.
def get_single_jogger_profile(jogger_id, db_client):
    try:
        jogger_doc_ref = db_client.collection("joggers").document(jogger_id)
        jogger_doc = jogger_doc_ref.get()
        if jogger_doc.exists:
            print(f"App: Retrieved profile: {jogger_doc.to_dict()}")
            return jogger_doc.to_dict()
        return None
    except Exception as e:
        print(f"App: Error getting profile: {e}")
        return None

def listen_for_realtime_speed_updates(jogger_id, db_client):
    speed_readings_query = db_client.collection("speed_readings").where("joggerId", "==", jogger_id).order_by("timestamp", descending=True).limit(10)
    print(f"App: Listening for updates for jogger: {jogger_id}")
```

The **write_speed_data_to_firestore** function ensures data is saved in the **speed_readings** collection. The **get_single_jogger_profile** function allows the mobile app to retrieve specific jogger details, while the **listen_for_realtime_speed_updates** function automatically receives updates.

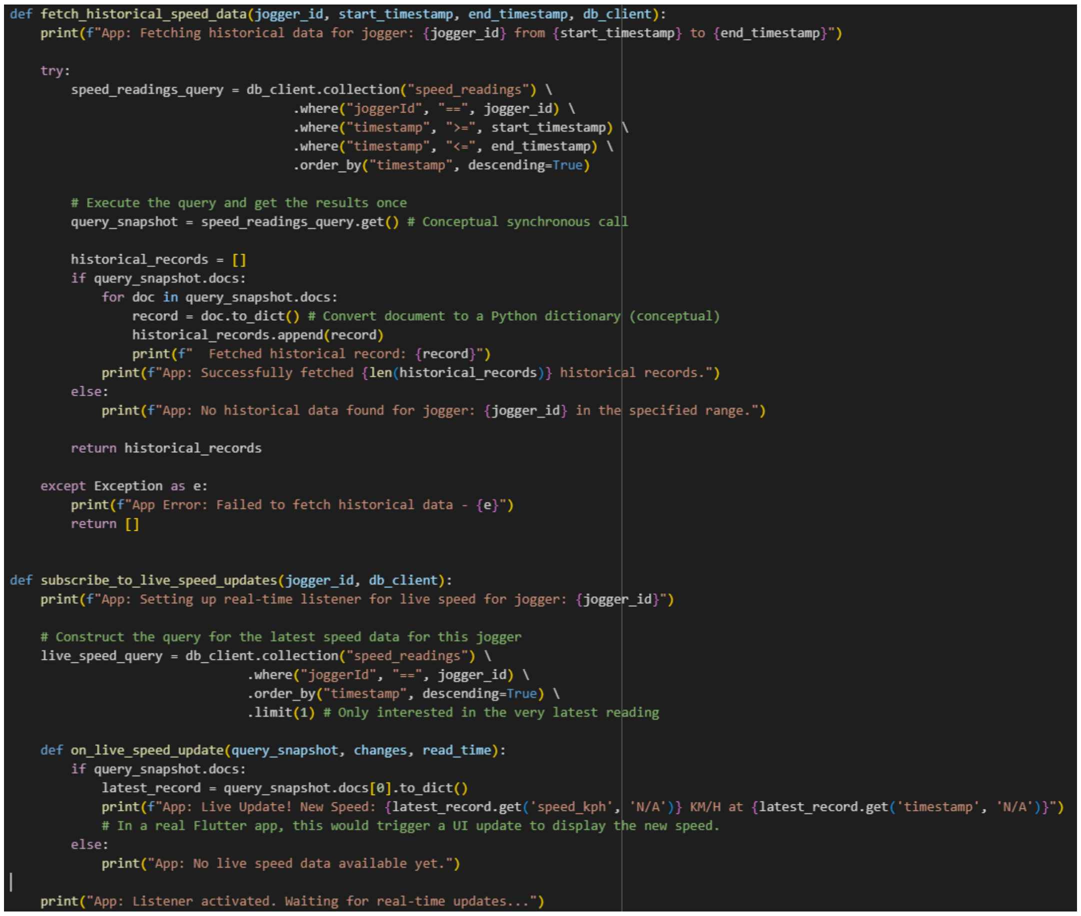
7.3. Data Access:

Using Firestore's real-time update and historical record capabilities, the mobile application effectively retrieves stored speed and jogger data.

The mobile application uses Firebase Firestore to access historical data and real-time updates. It initiates one-time queries for historical performance data, filtering the dataset using criteria like **joggerId** and timestamp ranges.

Firestore returns a snapshot of matching documents, which the application processes to populate historical views, charts, and summaries. This ensures instantaneous UI updates, providing a fluid and dynamic performance monitoring experience.

Figure 7:Data access



```python
def fetch_historical_speed_data(jogger_id, start_timestamp, end_timestamp, db_client):
    print(f"App: Fetching historical data for jogger: {jogger_id} from {start_timestamp} to {end_timestamp}")

    try:
        speed_readings_query = db_client.collection("speed_readings") \
                               .where("joggerId", "==", jogger_id) \
                               .where("timestamp", ">=", start_timestamp) \
                               .where("timestamp", "<=", end_timestamp) \
                               .order_by("timestamp", descending=True)

        # Execute the query and get the results once
        query_snapshot = speed_readings_query.get() # Conceptual synchronous call

        historical_records = []
        if query_snapshot.docs:
            for doc in query_snapshot.docs:
                record = doc.to_dict() # Convert document to a Python dictionary (conceptual)
                historical_records.append(record)
                print(f"  Fetched historical record: {record}")
            print(f"App: Successfully fetched {len(historical_records)} historical records.")
        else:
            print(f"App: No historical data found for jogger: {jogger_id} in the specified range.")

        return historical_records

    except Exception as e:
        print(f"App Error: Failed to fetch historical data - {e}")
        return []


def subscribe_to_live_speed_updates(jogger_id, db_client):
    print(f"App: Setting up real-time listener for live speed for jogger: {jogger_id}")

    # Construct the query for the latest speed data for this jogger
    live_speed_query = db_client.collection("speed_readings") \
                           .where("joggerId", "==", jogger_id) \
                           .order_by("timestamp", descending=True) \
                           .limit(1) # Only interested in the very latest reading

    def on_live_speed_update(query_snapshot, changes, read_time):
        if query_snapshot.docs:
            latest_record = query_snapshot.docs[0].to_dict()
            print(f"App: Live Update! New Speed: {latest_record.get('speed_kph', 'N/A')} KM/H at {latest_record.get('timestamp', 'N/A')}")
            # In a real Flutter app, this would trigger a UI update to display the new speed.
        else:
            print("App: No live speed data available yet.")

    print("App: Listener activated. Waiting for real-time updates...")
```

The pseudocode demonstrates how a mobile application can access data from Firebase Firestore. It demonstrates two main methods: fetching historical speed data and subscribe to live speed updates. The former involves fetching a one-time snapshot of past jogging data, while the latter involves setting up a listener to automatically update the live display with new speed records.

7.4. Security

Firebase Firestore database security is crucial for privacy and integrity of collected speed data. Firebase Security Rules control access to data through server-side access control statements. Key principles include authentication, authorization, data ingestion, user data access, and preventing unauthorized modification/deletion. Authenticated users or services must prove their identity to Firebase, while mobile app users must use Firebase Authentication for sign-up and login. Rules restrict data ingestion to authorized Firebase Cloud Functions, ensuring only authorized users can access their personal profile data. These rules ensure data integrity and prevent unauthorized access.

8. User Interface and Data Visualisation:

The User Interface (UI) is a crucial part of an interactive speedometer system, transforming raw data into useful information. It provides a smooth and interesting experience. The chapter covers the design of the application's screens, data visualization from Firestore, and general design tenets.

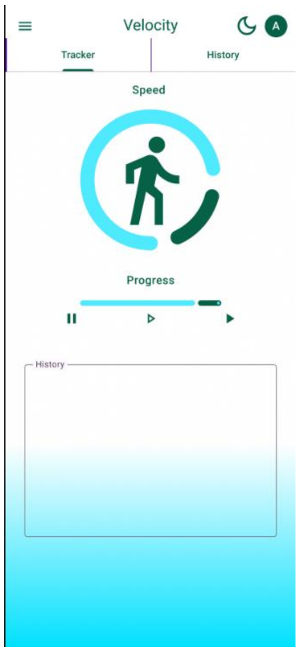
8.1. Core Principals of UI Design:

The mobile application will be designed to be intuitive, clear, and visually appealing. It will be self-explanatory, allowing users to easily find information and perform actions without extensive instructions. The data will be presented in a clean, uncluttered format, with careful consideration of typography, colour contrast, and element spacing. The UI layout will adapt fluidly across various screen sizes and orientations, ensuring a consistent viewing experience. The application will provide immediate visual feedback for user interactions and real-time data updates, fostering user engagement and confidence in the system's responsiveness.

8.2. UI Design Prototype:

The layout, interactive features, and information presentation of the main user interface of the "Velocity" mobile application are all shown in this section's conceptual prototype using Figma. Real-time tracking and personal historical data are instantly accessible thanks to the design's consolidation of key functions onto a single screen.

Figure 8: Velocity Menu



The application name "Velocity," the theme mode "Dark & Light," the profile icon, and the hamburger menu are all located in the bar.

Tabs: the main page is on the left, and the second page, which will be discussed shortly, is on the right.

First, the speed in relation to the maximum is visualised; second, the jogger's progress is tracked, including how many steps he took and his goal; and finally, the user's history is presented, including how much was sensed, his maximum, minimum and average speed and steps.

This design uses cool colours (green accents and cyan/teal) against a light background to create a minimalist and contemporary look. The app title, an auxiliary menu, and user/utility icons (profile access, theme toggle) are all located in the top navigation bar. The main "Tracker" and "History" tabs allow you to quickly switch between a placeholder for historical data and real-time speed visualisation (with conceptual session controls). By providing easy access to both live tracking and personal history from a single point of contact, this combined strategy optimises for mobile usage.

Auxiliary sections of the "Velocity" application are located beyond this main consolidated screen and are usually accessed through the top-left menu. Global System Analytics is one such section that aims to give a comprehensive overview of all user data. Presented through a variety of visualisations to highlight community-level trends, this includes important insights like the total number of unique joggers, the system-wide average speeds, and the highest speed ever recorded. To ensure data accuracy and privacy within the system, a Profile Management section provides users with a dedicated area to review and update their personal information, including details associated with their unique BLE identification tag.

8.3. User Experience (UX) and Accessibility:

The "Velocity" app is made to guarantee strong accessibility and the best possible User Experience (UX). A user-centred approach informs the design, which prioritises real-time metrics, ensures intuitive interaction, and reduces cognitive load. Users can easily begin, pause, or review sessions thanks to the self-explanatory navigation elements and controls. For effective use both during and after a run, important features like the live speed display and fast access to history are positioned prominently. To provide a unified and predictable user experience, a consistent design language will be used throughout the application, including consistent typography, iconography, and interaction patterns.

Basic accessibility guidelines, such as appropriate touch target size, legal typography, sufficient colour contrast, and unambiguous visual cues, will guide the design of the "Velocity" app's user interface. To ensure readability for users with different visual acuities, all text and important user interface elements will retain enough colour contrast against their backgrounds. The application's design makes sure that as many people as possible can use it.

9. Conclusion:

To give joggers real-time performance tracking and thorough data visualisation, this report has thoroughly described the conceptual design and theoretical implementation of an IoT-Enabled Interactive Speedometer System. The suggested architecture successfully combines an ESP32 microcontroller for edge processing, BLE tags for jogger identification, and an FMCW radar for accurate data acquisition.

With strong security regulations guaranteeing data integrity and privacy, Firebase Firestore functions as a scalable and real-time cloud data model. In addition, local SD card logging offers an essential layer of data redundancy, protecting against possible data loss in the event of network outages.

"Velocity," a user-focused mobile application, provides an easy-to-use interface for both historical analysis and real-time tracking. Although this theoretical framework shows great promise for effective data flow and user engagement, it inevitably faces issues that are typical of IoT deployments, including maintaining reliable wireless connectivity in a variety of settings, maximising battery life for continuous operation, and resolving possible real-world interference for precise BLE correlation. These elements reflect intrinsic constraints that need to be carefully considered for a workable implementation.

The system provides several opportunities for future improvement and advancement. They include adding social sharing or gamification features to the mobile application to increase user engagement, integrating with external mapping services for route tracking and distance calculation, and deploying advanced machine learning algorithms on the ESP32 for more complex motion filtering and prediction. To increase connectivity range and efficiency in various outdoor environments, more research into low-power wide-area network (LPWAN) technologies may be investigated.

Overall, a speedometer by itself wouldn't be as useful. The "Velocity" app made the innovation more interesting and fun by letting users interact with it and showing useful information about joggers and analytics. It can also lead to other ideas and inspirations, especially in the fitness world.