IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511105

Design and Implementation of an IoT-Based Student Bus Tracking System Using ESP32, MQTT and FastAPI

Mrs. Jayashuba J[1], Dhatri Muppuri[2], Raavi Pranitha[3], Ragipati Supriya[4]

Project Guide, Department of Computer Science and Engineering, K. S. School of Engineering and Management (K.S.S.E.M.) Bengaluru, India[1]

Department of Computer Science and Engineering, K. S. School of Engineering and Management (K.S.S.E.M.) Bengaluru, India[2-4]

Abstract: Students who depend on college buses often lack timely information about where the bus is and when it reaches their stop. This paper presents an IoT-based student bus-tracking system that com-bines an ESP32 GPS unit, MQTT communication, a FastAPI backend and a browser-based map. The backend validates incoming telemetry, maintains each bus's movement state, and stores positions and alerts in Supabase. Live positions and alerts are pushed over WebSockets to open browser clients, while Firebase Cloud Messaging (FCM) delivers background notifications to the devices of users subscribed to a bus; Supabase authentication handles sign-in. The alert engine generates event-based alerts for stop proximity, route deviation, trip milestones and loss of tracking data, and a GPS simulator exercises this logic. The reported evaluation comprises 24 automated unit tests, simulator runs, and a walking trial with the ESP32/GPS unit. The simulator runs ex-posed a missed-arrival case in which a bus crossed a stop between two consecutive location updates, and a segment-based proximity check was introduced to correct it. The paper describes the architecture, vali-dation rules and deployment considerations of a practical student bus-tracking prototype.

Index Terms: bus tracking, ESP32, GPS, MQTT, FastAPI, Web-Socket, Firebase Cloud Messaging, Internet of Things

I. INTRODUCTION

Colleges that operate their own buses usually follow fixed routes, yet students rarely have live information about the ve-hicle. An application that shows the bus position and alerts a rider when the bus starts moving or nears a stop addresses this need directly, and the vehicle-side unit can be assembled from commodity parts.
Building such a system involves several design decisions that are seldom reported together: how the vehicle unit reports its position, how the data reach the rider, how invalid or missing data are handled, and how alerts are derived from raw coordi-nates. This paper documents one set of answers built around an ESP32, MQTT and FastAPI.

A Requirements

The system was built to the following requirements.
R1 The vehicle unit reports position, altitude and speed about every three seconds while a valid, recent GPS fix exists.
R2 Positions reach riders as live map updates.
R3 Positions and alerts are stored, and stored positions can be shown as a route.
R4 Users sign in through Supabase authentication and sub-scribe to buses.
R5 Riders receive alerts for movement start, stop approach and arrival, route deviation and recovery, and trip start and completion, including when the application is in the back-ground.
R6 Invalid, stale or missing tracking data are detected and re-ported rather than displayed as valid positions.

B Contributions

The contributions are: (i) an end-to-end data path from a GPS unit through MQTT and a FastAPI backend to a live map and push notifications, with a simulator that feeds the same engine;
(ii) a documented set of validation rules and alert thresholds with their implementation; (iii) an automated test suite for that logic; and (iv) an analysis of the design considerations of the transport layer. College bus tracking is an established appli-cation, so the contribution lies in a documented, reproducible implementation rather than in a new technique.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511105

II. RELATED WORK

Harini Vasaki *et al.* describe an IoT-based college bus tracking and monitoring system in which a GPS device on the bus sends its position to a server and users view it in a mobile applica-tion [1]. Yeruva *et al.* present EduBusGuru, a mobile applica-tion for college bus tracking and management [2]. Sandhiya *et al.* report a real-time college bus tracking and notification sys-tem that alerts a student when a bus is within a set distance [3]. Jayashubha *et al.* survey IoT-based bus tracking and arrival-time prediction [4]. These works were consulted through their ab-stracts and bibliographic records, so Table 1 lists only the at-tributes stated there.

Table 1. Closely Related College-Bus Studies (From Abstracts)

Work	Location source	Stated features
[1]	GPS device on bus, server upload	Mobile app showing bus location
[2]	Not stated	Mobile app for tracking and management
[3]	GPS-based	Proximity alert to students
[4]	Survey of ESP32/GPS-based designs	Arrival-time prediction approaches

On the communication side, MQTT is a publish/subscribe protocol with three delivery levels: QoS 0 (at most once), QoS 1 (at least once) and QoS 2 (exactly once) [5]. WebSocket pro-vides a persistent full-duplex channel over a single TCP con-nection [6]. The present design uses MQTT between the vehi-cle unit and the backend, and WebSocket between the backend and the client.

III. SYSTEM ARCHITECTURE

Fig. 1 shows the architecture. The vehicle unit, an ESP32 with a GPS receiver, publishes JSON telemetry over a phone-hotspot Wi-Fi link to an MQTT broker [7]. A FastAPI [8] backend subscribes to the topic with the Paho Python client [9]. Each message is validated and passed to a tracking engine, which keeps one state and one alert engine per bus, stores positions and alerts in Supabase [10], broadcasts them to WebSocket clients, and sends push notifications through Firebase Cloud Messag-ing (FCM) [11]. A GPS simulator that emits NMEA GGA sen-tences, parsed with pynmea2 [12], feeds the same engine, so the map and the alert logic behave identically for both sources. Su-pabase also provides user authentication. Two channels reach the rider: WebSocket carries live positions and alerts to browser clients that have the page open, and FCM carries push notifi-cations to the registered devices of subscribed users, including when the page is closed or the screen is off.

All buses share one topic and are distinguished by the bus_id payload field. The engine tracks up to 50 distinct iden-tifiers, and the simulator supplies additional buses.

IV. IMPLEMENTATION

A Vehicle Unit

The unit is an ESP32 connected to a GPS receiver on UART1, with RX on GPIO16, TX on GPIO17 and a baud rate of 9600. The firmware uses WiFi. h, HardvareSerial. h, TinyGP-SPlus [13], PubSubClient [14] and ArduinoJson [15]. It joins a phone hotspot, connects to broker. hivemq. com: 1883 with client identifier ESP32_GPS_BUS_1, and publishes to trackr/positions. About every three seconds it publishes, but only if the GPS location is valid and its age is below 5000 ms. The payload has five fields: bus_id, latitude, longitude, altitude (m) and speed (km/h). PubSubClient publishes at QoS 0. The firmware re-establishes the Wi-Fi and MQTT connections every five seconds if either drops. The serial-monitor output shows the parsed GPS values and each publication.

B Backend

1) Ingestion

The Paho client connects with a random backend client identi-fier, because a duplicate of the device's identifier would make the broker disconnect the device. It subscribes at QoS 0 in its connect callback so that the subscription is restored after each reconnect, and hands each message to the asyncio event loop. The payload is validated by the rules in Table 2; a rejected mes-sage is counted and logged and does not reach the map or the database.

2) Tracking Engine

For each accepted position the engine updates the bus state, evaluates that bus's alert engine, broadcasts a location

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511105

message to WebSocket clients, stores the position, and then stores and broadcasts any resulting alerts. Broadcast precedes storage so that a slow database write does not delay the map. A watch-dog marks a bus stale when no accepted message has arrived for 15 s, which is five nominal publish intervals. The simulator advances along an OSRM road route (straight segments if the routing service is unreachable) and emits one GGA sentence every three seconds; it enters the engine through the same func-tion as live telemetry.

3) Movement State

The engine derives a movement state from GPS speed with hysteresis: a speed below 3 km/h is treated as STOPPED and a speed above 5 km/h as MOVING, while values in between leave the state unchanged to suppress jitter in the GPS speed. A transition from STOPPED to MOVING produces a movement event.

4) Persistence and API

Positions (bus identifier, latitude, longitude, speed, altitude, source and receive time) and alerts are stored in Supabase. The REST interface provides route and simulation controls, stored-route retrieval, alert retrieval with role and bus filters, device-token registration for FCM, and a tracking-status endpoint that reports the MQTT connection state, the number of rejected pay-loads and the per-bus age and state.

C Authentication, Subscriptions and Notifications

Supabase one-time-password sign-in and sign-up are imple-mented in the client and in backend routes, together with a bearer-token dependency that validates a token with Supabase.

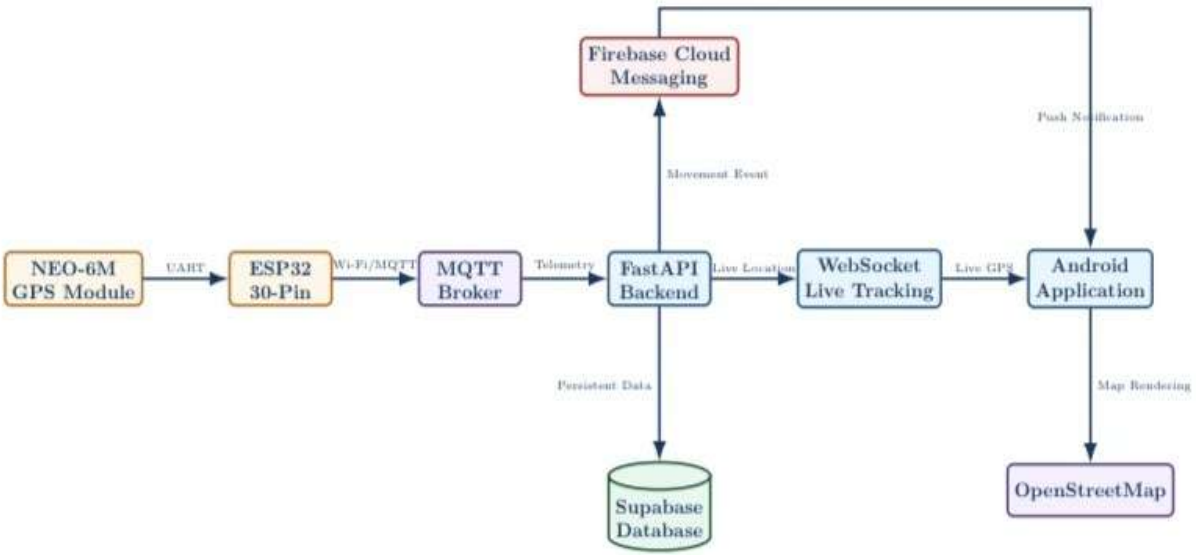


Figure 1. System architecture of the student bus tracking system.

This dependency is not attached to the tracking routes (Sec-tion VII-D). A signed-in user subscribes to one or more buses, and the client registers an FCM token with the backend.

Live updates and notifications use two different mechanisms. The WebSocket channel serves browser clients that are con-nected: it carries every position and every alert to all connected clients without per-rider filtering, and it stops when the page is closed. FCM serves background and device notifications: a movement or approach event for a bus is sent through FCM to the registered devices of the users subscribed to that bus, so sub-scriptions determine who receives push notifications but do not restrict what the WebSocket carries. Targeting alerts to an indi-vidual rider's own stop is not part of the present implementation and is listed as future work.

Each alert also carries a target role (driver, administrator, rider or all). The alert-retrieval endpoint and the client use this label, together with the bus, to filter which alerts are shown. This is display filtering and not access control, because the end-points do not verify the caller's role.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511105

D Browser Client

The client shows the route, stops and one marker per bus on an OpenStreetMap-based map. A bus is dimmed and labelled when it is stale, and a "no signal" indicator is shown in the sta-tus bar. Rider-facing alerts appear as in-app messages and in an alert list, recent stored alerts are loaded at start-up, and the Web-Socket reconnects with exponential back-off. Stored positions are retrieved from the backend and drawn as a route polyline for historical viewing. Because the client registers an FCM token, alerts are also delivered as system notifications when the page is in the background or the screen is off.

Table 2. Validation Rules and Alert Thresholds

Rule	Where	Condition
GPS validity and freshness	Firmware	Publish only if location is valid and age is less than 5000 ms
Payload	Backend	JSON object ≤ 1 KB; `bus_id` string ≤ 32 characters (optional allow-list); latitude and longitude finite, in range, and not (0,0); altitude numeric; speed 0–300 km/h
Stale telemetry	Backend	No accepted message for 15 s: mark stale, signal-loss alert; cleared by the next valid message
Signal quality	Engine	Fix quality 0: signal loss; fewer than 4 satellites: warning at most once per 60 s; unparsable frame: corrupt-stream alert
Invalid position	Engine	Non-finite or out-of-range coordinates skipped
Movement	Engine	Speed less than 3 km/h: STOPPED; greater than 5 km/h: MOVING
Stop approach	Engine	Distance ≤ 200 m, once per stop per run
Stop arrival	Engine	Distance ≤ 50 m, once per stop per run
Off-route	Engine	Greater than 50 m for 3 consecutive fixes; cleared at ≤ 30 m
Trip start / end	Engine	Route progress ≥ 0.005 / ≥ 0.995, once each

V. VALIDATION RULES AND ALERT LOGIC

Table 2 lists the validation rules and alert thresholds and states where each is enforced.

Algorithm 1 Handling one MQTT message

```
procedure ONMESSAGE(m)
    if validation of m fails then
        count and log; return
    update bus state and movement state
    compute route progress
    signal-quality check; clear stale latch
    if signal not lost then
        off-route check
        stop approach and arrival (Eq. 3)
        trip start and completion checks
    broadcast position; store position
    store alerts; broadcast alerts; send push
```

A Distance and Stop Proximity

Distance to a stop is computed with the haversine formula [16], using an Earth radius $R = 6371000$ m:

$$= \sin^2\frac{\Delta\phi}{} + \cos\phi_1 \cos\phi_2 \sin^2\frac{\Delta\lambda}{}, \qquad (1)$$

$$d = 2R \operatorname{atan2}\left(\sqrt{a}, \sqrt{1-a}\right). \qquad (2)$$

Simulator output from an earlier version of the engine, which compared only the current fix with each stop, contained eleven approach alerts but only one arrival alert. When the distance travelled between two fixes exceeds the 50 m arrival radius, the bus can pass a stop without a fix falling inside it. The engine therefore also tests the segment between the previous and the current fix. With s_i the shortest distance from stop i to that seg-ment, computed in a local equirectangular projection, the effec-tive distance is

 This work is licensed under a Creative Commons Attribution 4.0 International License



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026
DOI: 10.17148/IJARCCE.2026.1511105

$$d^{\text{eff}} = \min(d_i, s_i). \qquad (3)$$

The segment test is applied only when consecutive fixes are at most 1000 m apart, so that a single erroneous jump does not trigger alerts along its path. An approach alert is emitted when $d_i^{\text{eff}} \le 200$ m and an arrival alert when $d_i^{\text{eff}} \le 50$ m, each at most once per stop until the engine is reset. The two checks are independent, so a stop crossed between fixes yields both alerts. Algorithm 1 summarises the handling of one message.

B Off-Route Detection

The distance to the route polyline is computed in the same local projection, so east–west offsets are scaled by $\cos\phi$. An ear-lier implementation used a flat 111 km per degree in both axes, which overestimates east–west offsets by about 2.6% at 13°N. The engine keeps a counter of consecutive fixes farther than 50 m from the route; at three, the state latches to off-route and the alert is labelled for the driver, administrator and rider roles. The state clears only at ≤30 m, which gives hysteresis, and one good fix resets the counter before the latch.

C Signal Quality, Staleness and Trip Milestones

Signal-quality alerts use the NMEA fix quality and satellite count when they are available, which is the case for the sim-ulator. MQTT telemetry carries neither, so for live telemetry the vehicle unit's own validity check stands in for the fix qual-ity. Staleness and a fix of quality zero share one latch, which suppresses spatial checks until a valid position arrives and then emits a restoration alert.

Table 3. Implemented Components and How They Were Exercised

Component	Evidence
GPS acquisition, MQTT publish	Serial-monitor output, walking trial
MQTT ingestion, validation	MQTT ingestion, validation
Unit tests; walking trial	Unit tests; walking trial
Alert and movement logic	Alert and movement logic
Sign-in and subscriptions	Sign-in and subscriptions
Functional run of the client	Functional run of the client
Live map and in-app alerts (WebSocket)	Live map and in-app alerts (WebSocket)
Push notifications (FCM)	Push notifications (FCM)
Functional run of the client	Functional run of the client

Trip milestones use a normalised route progress $p \in [0, 1]$. For the simulator, p is the fraction of the simulated path already travelled. For live telemetry, the fix is projected onto the route polyline and

$$p = \frac{L(x)}{L_{\text{route}}} \qquad (4)$$

where $L(x)$ is the distance along the polyline from its start to the projection of the fix and L_{route} is the total polyline length. A trip-start alert is emitted the first time $p \ge 0.005$, that is, after about half a percent of the route, and a trip-completion alert the first time $p \ge 0.995$; each is emitted once per run. Because the projection is unambiguous only if the route start and end are in different places, the method assumes a route that does not loop.

VI. VERIFICATION

A Evaluation Overview

The evaluation has three parts that should be read separately: automated unit tests of the validation and alert logic, simulator runs that exercise the complete backend and map with generated positions, and a hardware trial in which the ESP32/GPS unit was carried on foot. Table 3 summarises how each component was exercised. WebSocket delivery of live positions and in-app alerts was exercised in the walking trial and the simulator runs, whereas FCM background notifications were exercised in functional runs of the client.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511105

B Automated Unit Tests

A standard-library test suite of 24 tests passes. It covers the haversine and local-projection distances, route progress, pay-load validation (accepted payloads and fourteen kinds of re-jected payload), the once-per-stop alerts, segment crossing, the jump guard, off-route debouncing and hysteresis, trip mile-stones, signal loss, restoration and cooldown, and the tracking engine with an in-memory store. The engine tests check that a valid payload reaches the broadcast hub and the store, that re-jections are counted, that stale detection is raised exactly once and cleared by the next message, that the bus limit is enforced, and that the hub drops its oldest messages when a client is slow.

C Simulator Runs

Three simulator runs on one day with the earlier version of the engine produced 36 alert records: three trip starts, eleven ap-proach alerts, one arrival alert and two completions for riders. The single arrival alert exposed the stop-crossing case described above and motivated the segment test of Eq. (3). These runs show the behaviour of the earlier engine on simulated positions; they are not measurements of the current code.

D Hardware Walking Trial

The ESP32/GPS unit was carried on foot to exercise the live path from GPS acquisition through MQTT, the backend and storage to the map and the stored route. This trial demonstrates that the data path operates with real GPS data at pedestrian speed. It was not a trip on a college bus route, so behaviour at vehicle speeds and over a full route remains to be evaluated.

E Scope of the Evaluation

The paper evaluates functional behaviour: that each stage of the data path operates and that the alert logic behaves as specified. It does not report latency, update rate, message-delivery ratio or GPS accuracy. Such measurements would require timestamps and sequence numbers in the payload and are left for future work.

VII. DESIGN CONSIDERATIONS

A Public Broker

The vehicle unit publishes to the public HiveMQ broker over port 1883 without transport encryption, and the broker does not authenticate publishers [7]. This keeps the prototype easy to re-produce, but anyone who knows the topic can read the teleme-try or publish messages to it. Payload validation rejects mal-formed data but not well-formed forgeries, and the optional bus allow-list and the 1000 m jump guard only narrow the exposure. User sign-in through Supabase protects the application, not the MQTT channel. A private broker with TLS and per-device cre-dentials is the appropriate choice for deployment, and the sys-tem is therefore presented as a prototype.

B Client Identifier and QoS 0

A broker must disconnect an existing client when a new con-nection uses the same client identifier [5]. On a shared broker, another client using ESP32_GPS_BUS_1 would displace the ve-hicle unit, so a deployment should use unique identifiers. Pub-SubClient publishes at QoS 0 [14], which keeps the unit simple but means that lost messages are not retransmitted. The three-second publish interval limits the effect of a single lost position on the map.

C Uplink and Timestamps

The vehicle unit uses a phone hotspot as its uplink. The payload carries no timestamp, so the backend stamps the receive time and staleness is judged by silence alone. Adding a timestamp and a sequence number to the payload would allow latency and loss to be measured directly.

D Authorization and Roles

The bearer-token dependency that validates a Supabase token is implemented but is not attached to the tracking routes. The WebSocket, route, simulation and notification endpoints there-fore do not require a token, and no role checks exist. Any client that can reach the service can open the WebSocket, replace the route, control the simulation, and read or mark alerts. The driver, administrator and rider labels on alerts are used only for filtering what is displayed and must not be read as enforced per-missions. Attaching token validation and role checks to these routes is required before deployment.

VIII. CONCLUSION AND FUTURE WORK

We implemented a student bus-tracking system in which MQTT telemetry from an ESP32/GPS unit, or from a simulator, passes through validation and an alert engine to Supabase storage, a live browser map, a historical route view and FCM

 This work is licensed under a Creative Commons Attribution 4.0 International License

DOI: 10.17148/IJARCCE.2026.1511105

push notifi-cations, with sign-in and bus subscriptions. The validation rules and alert thresholds are documented, the stop-crossing case ex-posed by simulator output was corrected with a segment-based check, and the core logic is covered by 24 passing unit tests. The hardware trial established that the data path works with real GPS data at walking speed.

Future work includes trials on an actual college bus route, a private TLS-secured broker with per-device credentials, times-tamps and sequence numbers with QoS 1 and buffering to en-able latency and delivery measurements, enforced authorisation and role checks on the tracking endpoints, alert targeting to an individual rider's own stop, and an independent cellular uplink.

REFERENCES

[1] M. Harini Vasaki, R. KamalaHarini, V. Ganthimathi, and J. Naskath, "An IoT based college bus trailing and surveil-ing system," *AIP Conf. Proc.*, vol. 2444, no. 1, Art. no. 070001, 2022, doi: 10.1063/5.0078357.

[2] S. Yeruva, A. Grandhi, S. D. Katta, S. S. Palutla, and G. D. Amruthaluri, "EduBusGuru: Enhancing col-lege bus tracking and management with real-time mobile app," in *Proc. 15th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, 2024, pp. 1–8, doi: 10.1109/ICC-CNT61001.2024.10724760.

[3] B. Sandhiya, V. V. K. Ravichandra, V. Siddhartha, M. Vanathi, J. Karthika, and D. Saral Jeeva Jothi, "Real-time college bus tracking and notification system for student convenience," in *Proc. 7th Int. Conf. Intell. Sustain. Syst. (ICISS)*, 2025, doi: 10.1109/ICISS63372.2025.11076399.

[4] J. G. Jayashubha, M. Dhatri, R. Pranitha, and R. Supriya, "AI-based smart bus live tracking system using IoT and machine learning: A survey of real-time ETA prediction and transportation analytics approaches," *Int. J. Adv. Res. Comput. Commun. Eng.*, vol. 15, no. 5, May 2026, doi: 10.17148/IJARCCE.2026.155250.

[5] A. Banks and R. Gupta, Eds., "MQTT version 3.1.1," OASIS Standard, Oct. 2014. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

[6] I. Fette and A. Melnikov, "The WebSocket protocol," IETF RFC 6455, Dec. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6455

[7] HiveMQ, "Public MQTT broker." [Online]. Available:https://www.hivemq.com/mqtt/public-mqtt-broker/

[8] S. Ramírez, "FastAPI documentation." [Online]. Avail-able: https://fastapi.tiangolo.com/

[9] Eclipse Foundation, "Eclipse Paho MQTT Python client." [Online]. Available: https://github.com/eclipse-paho/paho. mqtt.python

[10] Supabase, "Supabase documentation." [Online]. Avail-able: https://supabase.com/docs

[11] Google, "Firebase Cloud Messaging documentation." [Online]. Available: https://firebase.google.com/docs/cloud-messaging

[12] T. Knio, "pynmea2: Python library for the NMEA 0183 protocol." [Online]. Available: https://github.com/Knio/pynmea2

[13] M. Hart, "TinyGPSPlus." [Online]. Available: https:// github.com/mikalhart/TinyGPSPlus

[14] N. O'Leary, "PubSubClient: Arduino client for MQTT." [Online]. Available: https://github.com/knolleary/pubsubclient

[15] B. Blanchon, "ArduinoJson." [Online]. Available: https:// arduinojson.org/

[16] R. W. Sinnott, "Virtues of the haversine," *Sky and Tele-scope*, vol. 68, no. 2, p. 159, 1984.

 This work is licensed under a Creative Commons Attribution 4.0 International License