

Internet of Things Semester Project: Smart Office Presence

by

Drakontidis Konstantinos

Abstract:

This project presents the design and implementation of a Smart Office Presence system based on Bluetooth Low Energy (BLE) technology. The system aims to detect user presence at room level using pseudonymized BLE tags, while ensuring privacy through a privacy-by-design approach. The proposed architecture integrates BLE scanners and tags with edge computing capabilities to enable real-time presence detection and decision-making at the network edge, with supporting services optionally offloaded to a remote server.

Keywords: BLE, Pseudonymization, Real-Time Systems

1 System Overview

OfficeSense is a Smart Office system that detects user presence and identifies the specific room they are located in. This is achieved by using BLE tags carried by users, which are detected by BLE scanners placed in different rooms. The collected data is processed locally on a Raspberry Pi, which makes decisions about presence and communicates with the cloud only when necessary.

In addition, the system uses a camera to recognize the user's face, which is collected as data for authentication purposes. This adds an extra layer of security by verifying that the detected BLE tag matches the actual user.

OfficeSense also provides two separate dashboards. A real-time public dashboard anonymously visualizes users' locations along with their RSSI values, while a private administrative dashboard enables administrators to manage and update the system's permanent database.

The system is particularly suitable for large office buildings with multiple rooms and employees. It can support automated attendance tracking without the need for traditional card systems, reducing issues such as employees forgetting to scan their cards. Furthermore, it enhances building security by checking the consistency between user identity and BLE tag, and by detecting cases where an employee attempts to use someone else's credentials to falsely register presence.

2 Hardware Architecture

The structure of the system is comprised of the following devices

- Raspberry Pi
- Camera
- Waveshare ESP32-S3 Zero – 4MB
- Seeed Studio XIAO ESP32-C6

Starting from the lowest layer, the Seeed Studio XIAO ESP32-C6 is an ESP32-based microcontroller with WiFi and Bluetooth capabilities. In this system, it is used as a BLE tag device that periodically broadcasts over Bluetooth its unique

identifier, which is stored in its flash memory.

The Waveshare ESP32-S3 Zero – 4MB is another ESP32-based microcontroller equipped with WiFi and Bluetooth connectivity. In this architecture, it functions as a BLE scanner, receiving BLE advertisements transmitted by the tags. The collected data is then published to an MQTT broker over WiFi.

A USB camera is connected directly to the Raspberry Pi and is used for face recognition-based user authentication. The Raspberry Pi acts as the central processing unit of the system, receiving BLE data from the scanners via MQTT and capturing video streams from the camera over USB.

All incoming data is processed and stored locally on the Raspberry Pi. Additionally, the system updates a real-time dashboard, providing live information about user presence and location within the office environment.

3 Data Flow & Communication

The OfficeSense system follows a multi-stage data pipeline, starting from data generation at the device level and ending with visualization at the application layer.

Data generation begins at the BLE tags, which periodically broadcast BLE packets containing a 128-bit UUID (Universally Unique Identifier). In addition to the identifier, signal-related information such as RSSI (Received Signal Strength Indicator) is implicitly generated during the transmission process.

The BLE scanners capture these broadcasts and extract relevant data fields, including the tag identifier and RSSI value. They apply EMA (Exponential Moving Average) filtering to smooth the RSSI signal, as well as UUID validation and caching mechanisms. UUID validation is performed through a public API endpoint that returns validity responses. These responses are cached to minimize HTTP traffic and expire after a predefined interval. Invalid UUIDs are discarded by the scanners, while valid ones are structured into lightweight messages and transmitted over WiFi using the MQTT protocol.

Listing 1: UUID Validation

```
1 Request :  
2   GET /check/{TagID}  
3
```

```
4   Response :  
5       200 OK  
6       404 NOT FOUND
```

Listing 2: MQTT Topic

```
1   BLE Scanners Publish at :  
2       scanners/{RoomID}  
3  
4   Raspberry Pi Subscribes at :  
5       scanners/#
```

The MQTT broker, hosted in a Docker container, on the Raspberry Pi, acts as the central communication layer, enabling asynchronous and decoupled data exchange between devices and backend services. Incoming messages are consumed by a processing service, which performs filtering, validation, and location estimation based on signal strength.

Processed data is stored across two layers. Persistent information, such as user identities and tag mappings, is stored in SQLite, while real-time state data, including current user location and recent RSSI values, is maintained in Redis for fast access.

At the application layer, a backend API built with Node.js and Express.js supports the private administrative dashboard and provides access to the system's persistent database. The real-time public dashboard, on the other hand, retrieves live data from a separate API, that follows the NGSI-LD standard, to visualize anonymous user presence and RSSI information.

In parallel to the BLE-based presence detection, a camera-based authentication process is integrated into the system. The camera, connected to the Raspberry Pi via USB, operates as part of a separate processing pipeline.

Rather than performing continuous face recognition, the system runs a periodic camera verification loop, decoupled from the BLE pipeline. At a configurable interval, the verification service collects all currently unverified user sessions and passes their stored face embeddings to a dedicated face recognition process, which captures a camera frame, extracts facial features from it, and compares them against the stored embeddings.

The result of this process is used to validate the correspondence between the

detected BLE tag and the actual user. This enables the system to detect inconsistencies, such as unauthorized use of another user's tag.

The authentication result is then integrated into the main data pipeline and stored in the system's real-time data layer, allowing it to be visualized in the dashboard alongside presence information.

Example

A user carrying a BLE tag enters Room A. The tag broadcasts its pseudonymized identifier, let it be a UUIDv4 string:

Listing 3: Example tag ID

```
1 550e8400-e29b-41d4-a716-446655440000
```

which is detected by the nearest BLE scanner. An HTTP GET Request is sent to the UUID validity endpoint, and the response is cached in memory. The scanner then creates a Tag object in memory, measures multiple RSSI measurements for the same UUID, produces a filtered RSSI value (e.g., -62 dBm). Finally, the following message is transmitted via MQTT:

Listing 4: Example MQTT data packet

```
1 Scanner in room with ID: 84d689fb-c8a2-42b3-b697-3f26f486708a
2 Publish at scanners/84d689fb-c8a2-42b3-b697-3f26f486708a
3
4 {
5   "tag_id": "550e8400-e29b-41d4-a716-446655440000",
6   "rssi": -62
7 }
```

The Raspberry Pi receives the message and processes it, determining that the user is located in Room A. This event triggers the camera-based authentication process. The camera captures an image frame, and the face recognition service identifies the user as "User A".

The system then verifies that the detected BLE tag corresponds to the recognized face. If the identities match, the user's presence is confirmed. The result is stored in Redis as the current state and made available to the dashboard, where the user

is displayed in Room A in real time.

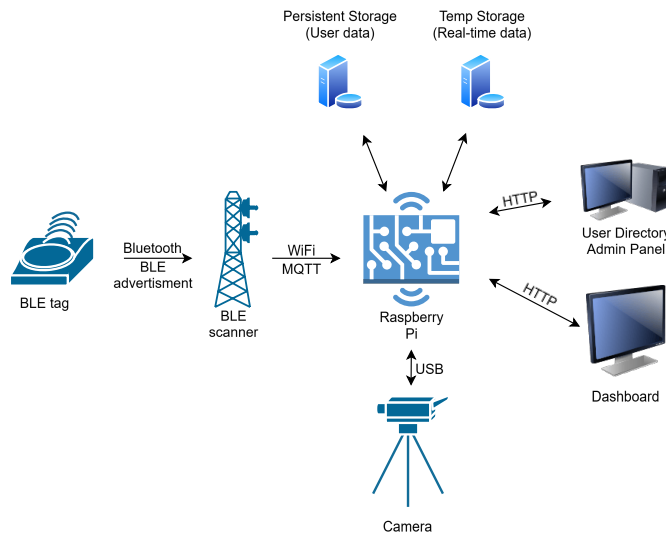


Figure 1: Example Data Flow & Communication

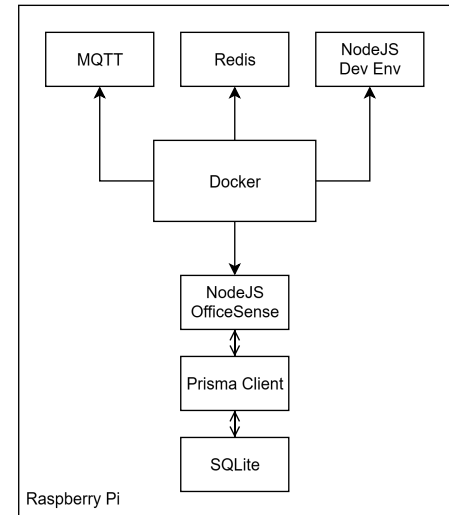


Figure 2: Services running on Raspberry Pi

4 Data Modeling

The system models real-world entities such as users and rooms using NGSI-LD. Sensor data from BLE devices and camera-based authentication is transformed into NGSI-LD compliant entities, enabling structured context representation and interoperability. The following NGSI-LD JSON messages are examples of data transmitted between the public API and the dashboard frontend.

Listing 5: User Presence Entity

```
1 Request:
2   GET /entities/urn:ngsi-ld:User:{Pseudonymized UUID}
3
4 Response:
5   {
6     "id": "urn:ngsi-ld:User:{Pseudonymized UUID}",
7     "type": "User",
8     "name": {
9       "type": "Property",
10      "value": "{Pseudonymized Name}"
11    },
12    "locatedIn": {
13      "type": "Relationship",
14      "object": "urn:ngsi-ld:Room:84d689fb-c8a2-42b3-b697-3
f26f486708a"
15    },
16    "rssi": {
17      "type": "Property",
18      "value": -62
19    },
20    "authenticationStatus": {
21      "type": "Property",
22      "value": "verified"
23    },
24    "observedAt": {
25      "type": "Property",
26      "value": "2026-04-14T10:00:00Z"
27    }
28  }
```

Listing 6: Room Entity

```
1 Request:
2   GET /entities/urn:ngsi-ld:Room:84d689fb-c8a2-42b3-b697-3
f26f486708a
3
4 Response:
5   {
6     "id": "urn:ngsi-ld:Room:84d689fb-c8a2-42b3-b697-3
f26f486708a",
7     "type": "Room",
8     "name": {
9       "type": "Property",
10      "value": "Meeting Room A"
11    },

```

```
12         "occupancy": {  
13             "type": "Property",  
14             "value": 3  
15         }  
16     }
```

5 Data Storage Strategy

The OfficeSense system handles multiple types of data with different characteristics, requiring a hybrid storage approach.

Data Types

The system generates and processes three main categories of data:

- **Static (persistent) data:** Includes user information, pseudonymized identifiers, and face recognition embeddings. This data changes infrequently and must be stored reliably.
- **Real-time data:** Includes RSSI values, current user location, and presence status. This data is continuously updated and requires fast access with low latency.
- **Optional historical data:** Includes logs of user movements and room occupancy over time, which can be used for aggregated analysis and reporting.

Storage Solutions Comparison

Two main storage solutions are considered for the system:

- **Relational Database - SQLite**
SQLite is a lightweight, file-based relational database suitable for embedded systems such as the Raspberry Pi. It is used to store structured and persistent data, such as user records and tag mappings.

Advantages: Low resource consumption, serverless, consistent & reliable

Limitations: Not optimized for high-frequency updates, limited performance for real-time operations

- **In-Memory Data Store – Redis**

Redis is an in-memory key-value store designed for high-speed data access. It is used to manage real-time data such as current user location and latest RSSI values.

Advantages: Fast read & write, real-time

Limitations: Data is not persistent by default, not suitable for long-term storage

- **Time-series Database - InfluxDB**

InfluxDB is specifically designed for handling time-stamped data, making it suitable for storing historical RSSI measurements, user movement logs, and room occupancy trends. This enables aggregated analysis, such as identifying peak usage hours, average occupancy levels, and long-term behavioral patterns.

Advantages: Efficient querying of historical data, can be visualized using Grafana

- **NoSQL Database - MongoDB**

An alternative approach could involve using a NoSQL database such as MongoDB. NoSQL databases are flexible and schema-less, making them suitable for storing semi-structured or evolving data. However, in this system, the data model is relatively simple and well-defined (users, tags, rooms), making a relational database such as SQLite more appropriate. Additionally, SQLite is better suited for edge devices due to its lightweight nature and low resource requirements.

Final Design Choice

The system adopts a hybrid storage architecture combining SQLite and Redis to leverage the strengths of both technologies.

SQLite is used for persistent data storage, ensuring reliability and structured data management. Redis is used for real-time state management, enabling low-latency

updates and efficient handling of frequently changing data.

This separation allows the system to maintain high performance while ensuring data integrity. Real-time data is processed and updated quickly in Redis, while critical information is safely stored in SQLite.

Optionally, a time-series database such as InfluxDB can be integrated to support aggregated analysis and historical data visualization.

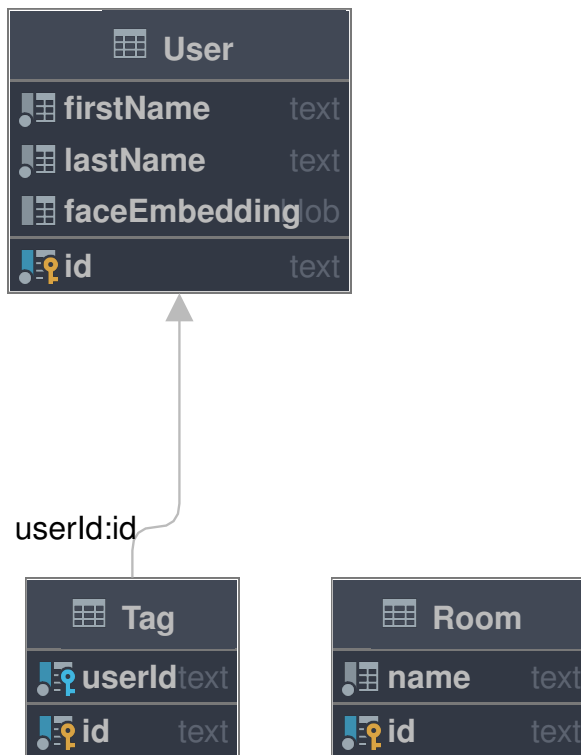


Figure 3: SQLite Database Schema

Listing 7: Redis Data Example

```

1 User Entities:
2   <key: {UserID},
3   value: {
4     "uuid": {Pseudonymized UUID},
5     "name": {Pseudonymized Name},
6     "rssi": -62,
7     "room": {RoomID},
  
```

```
8         "verified": true ,
9         "timestamp": "2026-04-14T10:00:00Z"
10     }>
11
12     Room Entities:
13         <key: {RoomID},
14         value: {Occupancy}>
```

6 Processing & System Logic

The OfficeSense system follows an event-driven architecture, where system actions are triggered by incoming sensor data. The core processing logic is responsible for detecting user presence, estimating location, and validating user identity.

System Triggers

The main trigger of the system is the reception of BLE data from scanners. Each time a BLE scanner detects a tag, it publishes a message containing the tag identifier, RSSI value, scanner ID, and timestamp. This event initiates the processing pipeline.

A secondary trigger is generated when a new presence event is detected or when a user appears in a different room. In such cases, the system activates the camera-based authentication process.

Processing Logic

The system performs the following steps:

1. **Data Filtering**

Incoming BLE messages are filtered to remove noise and invalid readings. Multiple RSSI values may be aggregated to improve accuracy.

2. **Location Estimation**

The system estimates the user's location by comparing RSSI values from multiple scanners. The room associated with the strongest signal is selected as the most probable location.

3. **User Identification**

The received tag identifier is matched with a pseudonymized user record stored in SQLite.

4. **Camera-Based Authentication**

When a presence change is detected, the camera is triggered to capture an image. A face recognition service processes the image and compares it with stored embeddings to verify the user's identity.

5. **Decision Making**

The system validates whether the BLE-based identity matches the face recognition result. If both match, the user is marked as verified. Otherwise, a potential mismatch or unauthorized access is detected.

6. **State Update**

The final result (location and authentication status) is stored in Redis for real-time access and visualization.

Example

A user enters Room A carrying a BLE tag. The nearest scanner detects the tag and sends a message with an RSSI value of -60 dBm. The system processes the data and determines that the user is located in Room A.

This event triggers the camera system, which captures an image of the user. The face recognition service identifies the person and compares the result with the expected user associated with the tag.

If the identities match, the system confirms the user's presence and updates the dashboard accordingly. If there is a mismatch, the system flags a potential security issue, such as unauthorized use of a tag.

Timeout-Based Absence Detection

In addition to event-based triggers, the system implements a timeout mechanism to detect user absence. Each time a BLE signal is received, the system updates the last-seen timestamp of the corresponding user.

If no new signal is received within a predefined time window, the system automatically marks the user as absent. This mechanism ensures robustness against signal loss and improves the accuracy of presence detection.

The timeout threshold is configurable, allowing the system to balance responsiveness and stability.

7 Reliability Considerations

OfficeSense is designed to handle common failure scenarios in IoT environments, such as network instability, device disconnections, and temporary data loss. Several mechanisms are implemented to ensure robustness and continuous operation.

Network Loss Handling

Due to the nature of BLE communication, data is continuously broadcast by the tags. As a result, occasional packet loss does not significantly affect system performance, since new data is frequently generated and received.

The system is designed to tolerate missing data by relying on repeated observations rather than individual messages. Additionally, a timeout-based absence detection mechanism ensures that users are correctly marked as absent when no signals are received for a defined period.

At the network level, MQTT provides reliable message delivery, and lightweight buffering mechanisms may be applied at the scanner level to handle temporary connectivity issues.

Device Disconnections

If a BLE scanner becomes unavailable (e.g., power loss or crash), the system continues operating using data from remaining scanners.

- Location estimation degrades gracefully, relying on fewer data points.
- The system avoids complete failure by not depending on a single device.

Similarly, if a BLE tag stops broadcasting (e.g., battery depletion), the user is automatically marked as absent after the configured timeout period.

Camera Failure Handling

The camera-based authentication operates as a secondary verification mechanism. If the camera fails or becomes unavailable:

- The system continues to perform BLE-based presence detection.
- Authentication status may be marked as “unverified” instead of causing a system failure.

This ensures that core functionality is preserved even when auxiliary components are not operational.

Data Consistency and Fault Tolerance

The system uses a hybrid storage approach:

- SQLite ensures reliable storage of persistent data such as user mappings and face embeddings.
- Redis manages real-time state with fast access.

To improve reliability:

- Redis data is continuously refreshed through incoming BLE events.
- Critical data is not solely stored in volatile memory.

8 Data Visualization

The system provides real-time and analytical visualization of user presence and room occupancy within the monitored environment.

Displayed Data

The dashboard presents:

- Current user location and RSSI (room-level presence)

- Room occupancy (number of users per room)
- Authentication status (verified / unverified)

Update Frequency

- Real-time data (presence, location, authentication) is updated every few seconds through continuous BLE scanning and backend updates.

End Users

The dashboard is designed for:

- Office administrators
- Facility managers
- Security personnel

These users can monitor occupancy, ensure compliance with schedules, and detect unauthorized access.

There is a separate control panel dedicated to system administrators, providing access to the User Directory.

Tool Comparison

- **Node-RED**

Node-RED provides a lightweight dashboard solution with real-time capabilities and seamless integration with MQTT.

Advantages: Easy setup, supports real-time data

Limitations: Limited visual capabilities

- **Grafana**

Grafana is a powerful visualization platform designed for monitoring and analytics.

Advantages: Advanced dashboards, historical & aggregated data, rich visuals

Limitations: Complex setup

Final Choice

The system primarily uses Node-RED for real-time visualization due to its simplicity.

Grafana can be optionally integrated for advanced analytics and historical data visualization, enhancing the system's capabilities without increasing core complexity.

The administrative panel is implemented using AdminJS and is configured to interact with the Express.js API, enabling administrators to manage and modify the User Directory.

Example Dashboard Panel

A typical dashboard panel displays real-time room occupancy. Each room is represented as a separate element showing:

- Room name
- Number of users currently present
- List of detected users (pseudonymized IDs)
- Authentication status indicator

This panel updates dynamically as new BLE data is processed, providing an accurate and immediate view of the office environment.

9 Design Justification

The system architecture is designed to balance performance, simplicity, and scalability, while adhering to privacy-by-design principles.

A hybrid storage approach is adopted, combining SQLite for persistent data and Redis for real-time state management. This separation ensures efficient handling of both static and dynamic data.

An event-driven architecture is used, where BLE detections act as triggers for processing and decision-making. This approach reduces unnecessary computation and improves system responsiveness.

The use of edge processing on the Raspberry Pi minimizes latency for the core presence detection and face recognition workloads. Supporting services (Redis, Mosquitto, and InfluxDB) are deployed on a remote server, offloading storage and brokering from the Pi while keeping all processing logic and camera-based verification local. This separation balances resource constraints on edge hardware with the operational convenience of managed remote services.

Trade-offs include:

- Reduced system complexity by avoiding full cloud integration
- Limited scalability compared to distributed systems
- Simpler pseudonymization instead of more complex rotating identifiers

These decisions prioritize reliability and ease of implementation.

10 Implementation Reality Check

One of the most challenging aspects of the system is accurately determining the user's location using BLE RSSI values.

RSSI-based localization is inherently unstable due to, signal interference, obstacles and variations in device orientation. These factors can lead to fluctuating signal strength and incorrect room estimation.

To address this challenge, the system applies:

- Signal filtering and averaging to smooth RSSI fluctuations
- Multiple scanner comparison, selecting the strongest signal

- Debouncing logic, requiring consistent readings before updating location
- Timeout mechanisms to avoid rapid state changes

This approach improves reliability while maintaining low computational complexity.

11 System Implementation

11.1 Tags

The BLE tag firmware runs on the Seeed Studio XIAO ESP32-C6, built with ESP-IDF and the NimBLE Bluetooth stack. At boot, the device initializes NVS flash storage, the USB-Serial-JTAG interface, and a status LED (active-low, lit once advertising is active), then loads its configuration (UUID, device name, and advertising interval) from NVS. If no configuration has been saved (e.g. after a flash erase), the firmware falls back to compile-time defaults defined in `main.h`.

Once configuration is loaded, the device initializes the NimBLE host stack and registers a sync callback that triggers non-connectable, general-discoverable BLE advertising (`BLE_GAP_CONN_MODE_NON / BLE_GAP_DISC_MODE_GEN`). The advertisement payload includes the device name and a 128-bit UUID, set via `ble_gap_adv_set_fields`. The advertising interval is configurable in milliseconds and internally converted to the 0.625 ms BLE time units expected by the stack. On a successful `ble_gap_adv_start`, the onboard LED is switched on to indicate the tag is actively broadcasting.

To support field provisioning and reconfiguration without reflashing firmware, the tag exposes a command-line interface over USB-Serial-JTAG, running as a dedicated FreeRTOS task. The CLI supports the following commands:

Listing 8: Tag CLI Commands

1	<code>set uuid <uuid></code>	set the tag's broadcast UUID
2	<code>set name <name></code>	set the advertised device name
3	<code>set adv_interval <ms></code>	set advertising interval (ms)
4	<code>show uuid / name / adv_interval</code>	
5	<code>save</code>	persist current config to NVS
6	<code>reset</code>	erase NVS config and reboot
7	<code>reboot</code>	restart the device
8	<code>help</code>	list available commands

UUIDs are accepted in standard 8-4-4-4-12 string form and parsed via `uuid_from_string`, which reverses byte order during parsing to match the little-endian byte layout NimBLE expects for 128-bit UUIDs. Configuration changes made via the CLI are held in RAM until explicitly persisted with `save`.

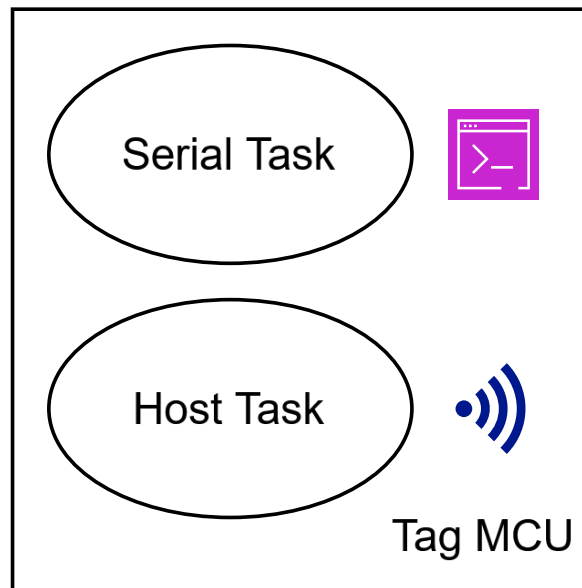


Figure 4: Tasks running in Tags

11.2 Scanners

The BLE scanner firmware runs on a Waveshare ESP32-S3 board, built with PlatformIO and the Arduino framework, using the NimBLE-Arduino and PubSubClient libraries. The firmware is structured around three concurrent FreeRTOS tasks (NimBLE scan callback, HTTP task for UUID validation, and MQTT task for publishing) coordinated through queues and a shared, mutex-protected UUID cache.

Scanning and Filtering

The scanner performs a continuous passive BLE scan with duplicate filtering disabled, so that every advertisement is processed as it arrives. On each result, the `CB::onResult` callback discards any advertisement that does not carry a 128-bit service UUID or whose advertised name does not match the configured tag name. Advertisements that pass this filter are packaged into a `BleEvent` (UUID and RSSI) and pushed into a queue for processing in the main loop, keeping the scan callback itself lightweight.

UUID Validation and Caching

Each unprocessed UUID is looked up in an in-memory cache before being trusted. The cache tracks each UUID state as one of `PENDING`, `RETRY`, `VALID`, or `INVALID` alongside the timestamp of its last check. A UUID seen for the first time, or returning from a failed HTTP request (`RETRY`), is marked `PENDING` and queued for validation against the lookup API. UUIDs previously marked `INVALID` are periodically re-checked after a configurable retry timeout, allowing tags that were not yet registered in the backend at scan time to become recognized later without requiring a scanner restart. Only UUIDs in the `VALID` state are forwarded to per-tag RSSI processing while others are silently dropped. Access to the cache is serialized with a mutex, since it is read and written from both the main loop and the HTTP task. A periodic cleanup pass evicts cache entries that have not been refreshed within a configurable timeout, preventing unbounded growth from transient or one-off tag sightings.

RSSI Processing and Publishing

Each valid tag is tracked by a `Tag` object, which maintains a fixed-size sliding window of the five most recent RSSI samples. On each new sample, the window median is computed first, and the result is fed into an exponential moving average (EMA) with a smoothing factor of 0.2 for the new value and 0.8 for the running average. Computing the median before applying the EMA suppresses transient outliers (e.g. a single anomalous reading caused by interference) before they can influence the smoothed signal.

A tag publishes a new MQTT message once its sample window is full and either of two conditions is met: a configurable time interval has elapsed since the last publish, or the EMA has moved by more than a configurable threshold since the last published value. This dual trigger keeps the published RSSI reasonably current even when the signal is stable, while still reacting promptly to significant movement. Tags that are not seen again within a configurable timeout are removed from memory by the same periodic cleanup pass that prunes the UUID cache.

Networking

WiFi connectivity is established at startup using configured credentials, with an optional static IP/gateway/subnet. The main loop periodically checks the WiFi status and reconnects automatically on failure. The MQTT task maintains the

broker connection independently, reconnecting whenever it drops, and publishes RSSI updates as compact JSON messages of the form `{"tagID": "...", "rssi": ...}` on a per-device topic derived from the scanner device name. UUID validation requests are issued over HTTP from a dedicated task to avoid blocking the scan loop or the MQTT task while waiting on the lookup API.

Configuration

As with the tag firmware, scanner configuration (cleanup intervals, tag timeout, UUID cache retry/cleanup timeouts, the validation API endpoint, MQTT broker details, and WiFi credentials) is persisted using the ESP32 Preferences (NVS) API and falls back to compile-time defaults when unset. A serial command-line interface exposes generic `set <key> <value>` and `show <key>` commands operating on these configuration keys, along with `save`, `reset`, `reboot`, and `help`, allowing scanners to be configured and re-deployed without reflashing firmware.

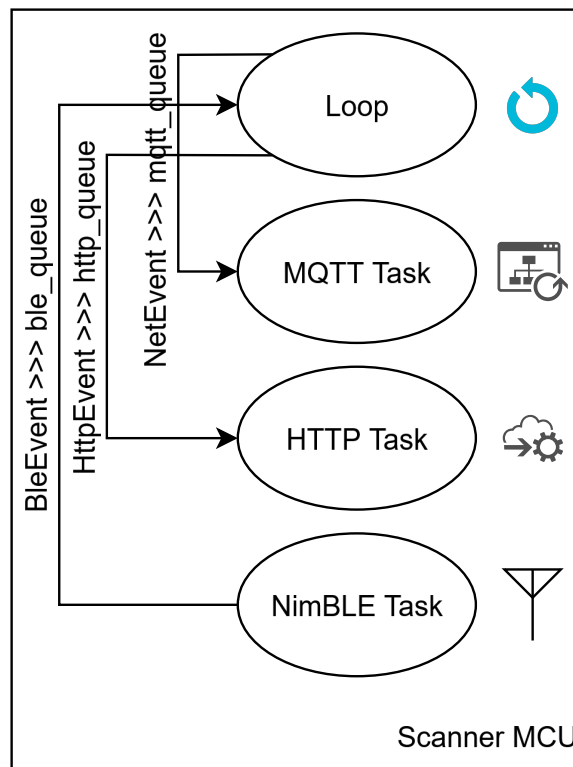


Figure 5: Tasks running in Scanners and data flow in queues

11.3 Pi

The Raspberry Pi runs the central decision-making service as a Node.js/TypeScript application, packaged as a Docker image and deployed alongside separate containers for Redis, the Mosquitto MQTT broker, and Node-RED.

Startup

On startup, the service connects to Redis, flushes any stale state from a previous run, and re-seeds room occupancy counters to zero from the set of rooms defined in SQLite. It then establishes a second, dedicated Redis connection used exclusively to subscribe to keyspace expiration events, connects to the MQTT broker and subscribes to the scanner topic, starts a periodic cleanup worker for stale transition state, and begins the camera verification loop.

Message Ingestion and Decision Logic

Each MQTT message published by a scanner is routed to the decision engine together with the room identifier, which is derived directly from the per-scanner topic the message was received on. The tag identifier carried in the payload is resolved to a persistent user record via a SQLite lookup. A per-tag lock prevents two messages for the same tag from being processed concurrently if they arrive in quick succession.

If the resolved user has no existing session in Redis, a new one is created: a random pseudonymous identifier and a randomly generated nickname are assigned, the session (pseudonym, nickname, RSSI, room, verification status, timestamp) is stored under a time-limited key, a reverse mapping from pseudonym to real user ID is stored separately, and the room's occupancy counter is incremented. If a session already exists and the reported room matches the user's current room, the session's RSSI and timestamp are simply refreshed, extending its expiry.

If the reported room differs from the user's current room, the decision is delegated to a per-user room transition state machine rather than being applied immediately, in order to avoid flickering between rooms caused by momentary signal fluctuations near room boundaries.

Room Transition Logic

Each user with an active session that has reported a different room is tracked by a `RoomTransition` instance, which maintains a single transition candidate consisting of a candidate room, an RSSI value, a sample count, and a timestamp marking when that candidate was first observed.

An incoming reading is only considered for a transition if either its RSSI exceeds the user's current room's RSSI by more than a hysteresis margin, or the current room has not produced a fresh reading within a configurable loss threshold (i.e. the signal in the current room appears to have been lost). If neither condition holds, any in-progress candidate is discarded and no transition occurs.

If a reading does qualify, it is compared against the existing candidate: if there is no candidate yet, or the new room's RSSI exceeds the existing candidate's RSSI by more than a separate candidate hysteresis margin, the candidate is replaced (or initialized) with the new room and the sample count reset to one. Otherwise, the sample count for the existing candidate is incremented and its RSSI updated.

A transition is finally committed once the candidate has accumulated enough confirming samples and has persisted for at least a configurable debounce period, or as a faster path, once the original room signal has been lost and the candidate has enough confirming samples, without necessarily waiting for the full debounce period. On commit, the user's occupancy is moved from the old room to the new one in Redis, the session is updated with the new room and a reset verification status, and the candidate state is cleared. Transitions belonging to users that have not produced any reading for an extended period are periodically purged by a background cleanup worker.

When a transition is committed (or when a user appears for the first time after a period of absence) a corresponding event is written to InfluxDB for post-hoc analysis. Each event records the pseudonymized user identifier, the room UUID and human-readable room name, the event type (`enter` or `leave`), the RSSI value at the time of the transition, and a timestamp. A room entry and a room exit event are written together on every room change: a `leave` event for the previous room and an `enter` event for the new one. A sole `enter` event is written when a user is first detected after absence.

Listing 9: InfluxDB Event Schema

```
1 measurement: room_events
```

```
2
3     tags:
4         pseudo_id      (pseudonymized user UUID)
5         room_id        (room UUID)
6         room_name      (human-readable name, resolved at write time)
7         event          ("enter" | "leave")
8
9     fields:
10         rssi           (float, signal strength at time of event)
11
12         timestamp:     (event time, InfluxDB line protocol)
```

Presence Expiry

User sessions in Redis are stored with a time-to-live rather than being explicitly deleted, so that a user who stops being detected (e.g. leaves the building, or their tag loses signal) is automatically considered absent once their session key expires. This expiration is observed through Redis keyspace notifications: when a session key expires, a listener decrements the occupancy counter for the user's last known room and removes the corresponding pseudonym mapping, keeping room occupancy counts consistent without requiring an explicit absence-detection poll.

Camera-Based Verification

Face-based verification runs as an independent polling loop, decoupled from the BLE/MQTT presence pipeline. On each interval, the service collects all currently unverified user sessions in Redis that have a stored face embedding in SQLite, and passes their identifiers and embeddings to a separate compiled verification process (`cameraSession`) over standard input as newline-delimited JSON. This process captures a frame from the camera, performs face recognition, and returns over standard output which of the supplied identifiers, if any, were matched against a detected face.

Users confirmed by this process have their session's verification status set to true in Redis, without affecting the session's existing expiry. To avoid a verification remaining valid indefinitely after a user has left the camera's view, each successful verification schedules an automatic reversion to unverified after a configurable timeout, independent of the verification polling interval itself.

cameraSession Binary

Face capture and recognition are implemented as a standalone Go binary, `cameraSession`, invoked by the Pi core service as a short-lived subprocess for each verification cycle, communicating over standard input/output rather than a network interface.

On startup, the binary loads a face recognizer using the `go-face` library (Go bindings for `dlib`), then captures a single frame from the camera over `Video4Linux2` (`/dev/video0`) using the `blackjack/webcam` library, configured for MJPEG capture at 1280×720 . The captured frame is passed through `dlib` CNN-based face detector and recognizer, which returns, for every face found in the frame, a 128-dimensional face descriptor. If no face is detected in the frame, the binary prints `no face found.` and exits without further processing.

If at least one face descriptor was extracted, the binary reads newline-delimited JSON packets from standard input, each containing a user identifier and a previously stored face descriptor (computed at enrollment time and supplied by the caller). Each packet is verified concurrently in its own goroutine: the squared Euclidean distance is computed between the packet's stored descriptor and every descriptor extracted from the captured frame, and the user is considered verified if any comparison falls below a fixed distance threshold (0.35). Verification results are accumulated in a mutex-protected slice shared across goroutines and, once all input packets have been processed, are marshaled to JSON and printed to standard output as a single array of `{uuid, verified}` results, which the calling Node.js process parses to update verification status in Redis.

This design allows an arbitrary number of unverified users to be checked against a single captured frame in one process invocation, since the comparison is dominated by descriptor distance calculations rather than additional camera captures or face detection passes.

Lookup API

The lookup endpoint, used exclusively by BLE scanners to validate tag UUIDs, exposes a single route, `GET /check/:uuid`. It checks for the existence of a matching tag record in SQLite via Prisma and responds with HTTP 200 if found or 404 otherwise, with no response body in either case, matching the lightweight check-only contract expected by the scanner firmware's UUID cache.

NGSI-LD Live Data API

The live data API exposes the system real-time state, sourced from Redis, as NGSI-LD compliant entities, and is consumed by the Node-RED dashboard. Two entity types are supported: `User`, representing an active, pseudonymized presence session, and `Room`, representing a room's current occupancy.

User and room state is read from Redis by scanning for keys matching the `user:*` and `room:*` prefixes respectively, then converted into NGSI-LD entities: each user's stored RSSI, room, verification status, and timestamp are mapped into NGSI-LD `Property` and `Relationship` fields, with verification status surfaced as the string `"verified"` or `"unverified"`.

Three routes are exposed:

- `GET /entities` returns all User and Room entities together.
- `GET /entities?type=user` or `type=room` returns entities of a single type.
- `GET /entities/:urn` returns a single entity by its NGSI-LD URN (e.g. `urn:ngsi-ld:Room:{RoomID}`), looking up the corresponding user or room directly by ID.

Since pseudonymized user sessions in Redis are keyed by the real (non-exposed) user ID, with a separate reverse mapping from pseudonym to real ID, single-user lookups by URN first resolve the pseudonym back to the underlying Redis key before fetching the session data, keeping the pseudonymized identifier as the only one visible through the API.

Node-RED Dashboard

The public-facing real-time dashboard is implemented as a Node-RED flow using the `node-red-dashboard` (FlowFuse) UI nodes, running in its own container and polling the live data API directly over HTTP rather than connecting to Redis or SQLite.

Two independent polling branches run on a fixed interval: one requests all user entities (`/livedata/entities`) and feeds them through a function node that resolves each user's room relationship to a human-readable room name before populating a table widget. The other requests room entities only (`/livedata/entities?type=room`) and feeds a parallel table alongside a bar chart visualizing occupancy per room.

Both widgets are laid out on a single dashboard page, giving administrators and facility staff a live, anonymized view of user locations and room occupancy without exposing the underlying persistent identities.

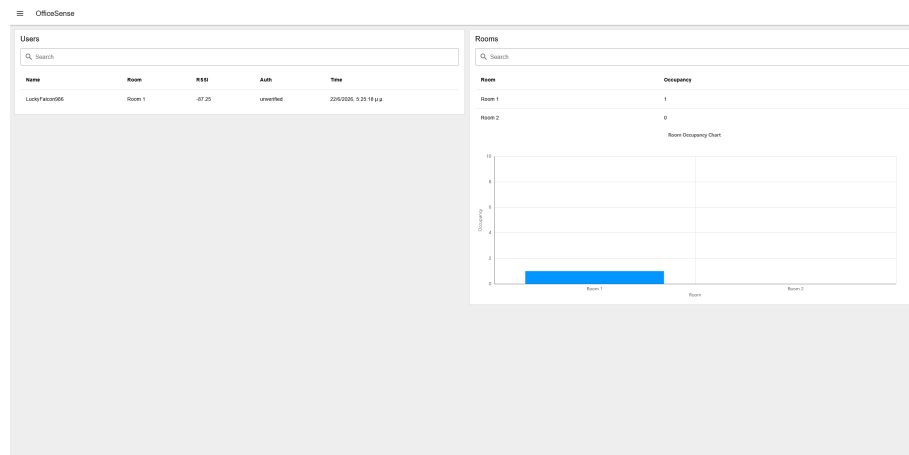


Figure 6: Node-RED Dashboard with one active user

Administrative Dashboard (AdminJS)

The private administrative dashboard is built with AdminJS, mounted under `/admin` and backed directly by the system's SQLite database through the `@adminjs/prisma` adapter. Three resources are exposed: `User`, `Tag`, and `Room`, each with curated list/show/edit/filter properties rather than exposing the full Prisma schema, and the `Tag` resource referencing its owning `User` as a relationship. Access is protected by AdminJS's built-in authenticated router, checking submitted credentials against environment-configured admin email/password, with session state backed by a cookie-based Express session shared with the rest of the API.

Custom behavior is added through AdminJS component and custom-action system. The `User` resource defines a custom `faceEmbedding` property renderer that displays only whether an embedding is set, rather than the raw stored vector, and a custom `uploadFace` record action that opens a dedicated upload panel. This panel lets an administrator select a JPEG photo, which is base64-encoded client-side and submitted to the action's handler. The handler spawns a separate compiled binary, `extractEmbeddings`, passing it the image data over standard input and parsing a JSON response containing the extracted face descriptor (or an error) from

standard output, storing the resulting descriptor on the user's record in SQLite on success.

The AdminJS dashboard landing page is also replaced with a custom component that polls a dedicated admin-only endpoint every ten seconds, displaying a live table that joins active Redis presence sessions with their corresponding registered users, showing side by side, each user's real identity, their real and pseudonymized identifiers, and their current pseudonymous nickname. This view exists purely as an administrative tool for verifying the BLE pseudonymization mapping and is not exposed anywhere outside the authenticated admin panel.

In production, AdminJS React components must be pre-bundled rather than compiled on demand, since the development-mode `admin.watch()` flow is not available. A separate bundler script (`bundler.ts`) invokes `@adminjs/bundler` ahead of time to produce a static bundle, which is then served by the Express application as static files under `/admin/frontend/assets`. Getting this production bundling and static asset path configured correctly (so that the custom components were emitted into the bundle and resolvable by AdminJS frontend at runtime) was the most involved part of getting the administrative dashboard working outside of development mode.

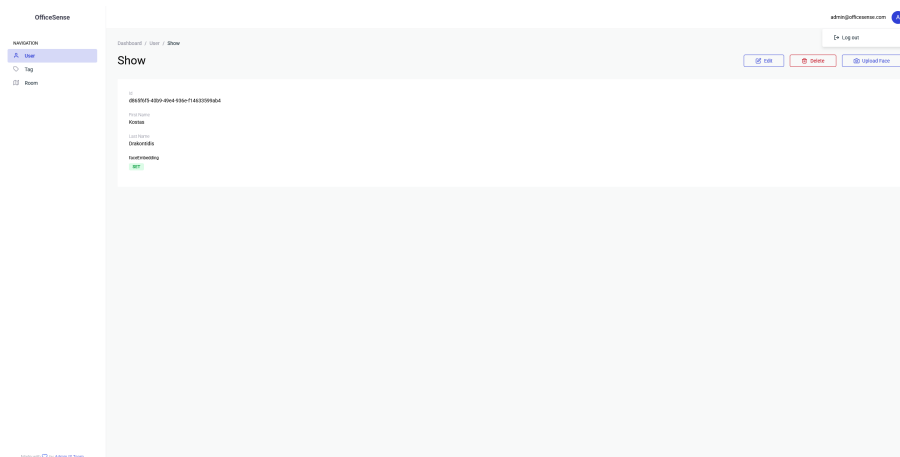


Figure 7: AdminJS user resource

Admin Mapping API

A small internal endpoint, `GET /admin/map`, supports the dashboard session-mapping view. It is gated by a session-check middleware that rejects unauthenticated re-

quests, independent of AdminJS own authentication, since it is mounted outside AdminJS router. On each request, it scans all active `user:*` keys in Redis, and for each session looks up the corresponding real user record in SQLite by ID, returning a combined list of real names, real user IDs, pseudonymous IDs, and pseudonymous nicknames for every currently active session.

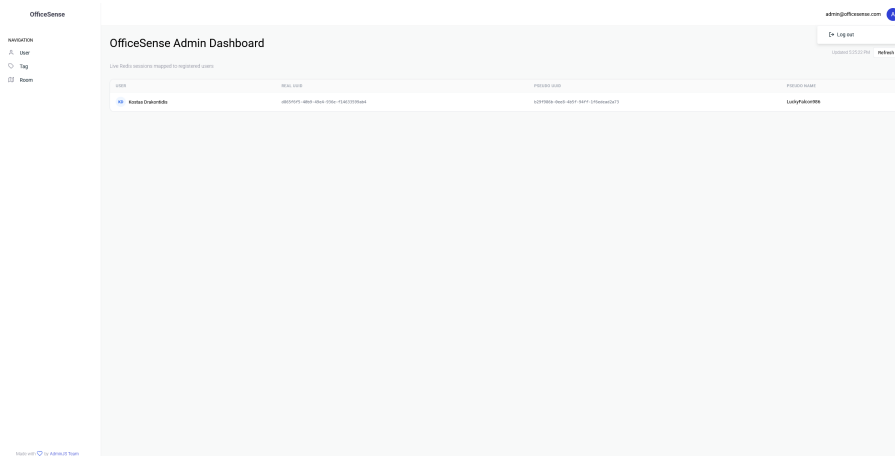


Figure 8: AdminJS utilizing map endpoint

extractEmbeddings Binary

Face embedding extraction, used when an administrator enrolls a user's photo through the AdminJS dashboard, is implemented as a second standalone Go binary, `extractEmbeddings`, structurally similar to `cameraSession` but operating on a single uploaded image rather than a live camera frame.

The binary reads a base64-encoded JPEG image from standard input, decodes it, and resizes it down to a maximum dimension of 600px using bilinear scaling if larger, re-encoding it before passing it to `go-face` CNN-based detector and recognizer. Unlike `cameraSession`, which tolerates any number of detected faces by comparing each candidate independently, `extractEmbeddings` requires exactly one face to be present in the submitted image. Zero faces or more than one are both treated as failure conditions, since the binary purpose is to enroll an unambiguous reference descriptor for a single known user rather than to identify an unknown person among several.

On success, the extracted 128-dimensional face descriptor is returned as JSON

on standard output, alongside a success flag. On any failure (missing or malformed input, an undecodable image, or a face-count mismatch) the same JSON structure instead carries a descriptive error message and a success flag of false. This uniform `{success, descriptor, error}` contract on stdout is parsed directly by the AdminJS `uploadFace` action handler, which stores the descriptor on the user's record in SQLite only when `success` is true.

12 Installation

OfficeSense has three independently built and deployed components: the tag firmware, the scanner firmware, and the Raspberry Pi software stack.

Tag Firmware (ESP-IDF)

The BLE tag firmware targets the Seeed Studio XIAO ESP32-C6 and is built with Espressif ESP-IDF framework, which must be installed and sourced (`get_idf / export.sh`) prior to building. With the IDF environment active, the firmware is built and flashed as follows:

Listing 10: Building and Flashing Tag Firmware

```
1 idf.py set-target esp32c6
2 idf.py build
3 idf.py -p <PORT> flash
```

`<PORT>` is the serial device the XIAO ESP32-C6 enumerates as when connected over USB (e.g. `/dev/ttyACM0` on Linux). The same serial connection used for flashing also exposes the configuration CLI described in Section 11.1.

Scanner Firmware (PlatformIO)

The BLE scanner firmware targets the Waveshare ESP32-S3 board and is built with PlatformIO under the Arduino framework. With the PlatformIO CLI installed, the firmware is built and uploaded using the environment already defined in `platformio.ini`:

Listing 11: Building and Uploading Scanner Firmware

```
1 pio run -t upload
```

PlatformIO resolves the correct upload port and toolchain automatically based on the `esp32-s3-devkitc-1` environment configuration and the required libraries (`NimBLE`, `-Arduino`, `PubSubClient`) are fetched automatically on first build.

Raspberry Pi Application

The Raspberry Pi application is packaged as a multi-stage Docker image, while Redis, the Mosquitto MQTT broker, and Node-RED run as separate containers defined in a standalone `docker-compose.yml`. A `Makefile` wraps the build and run commands for the application image itself.

Go Binaries

The two native components used by the application, `cameraSession` and `extractEmbeddings`, are written in Go and depend on `go-face`, which in turn requires `dlib`. Since `dlib` is resource-intensive to compile and the Raspberry Pi runs on an ARM64 architecture, both binaries are cross-compiled ahead of time on a development machine using a dedicated Dockerfile and Makefile per binary, rather than being built inside the Pi application's own Docker build.

Each cross-compile Dockerfile uses a `golang:bookworm` build stage with the `arm64` architecture added via `dpkg --add-architecture`, installing `arm64` cross-toolchains and `arm64` builds of `dlib`, `libjpeg`, `BLAS`, `OpenBLAS`, `ATLAS`, and `LAPACK`. A small wrapper script substitutes `aarch64-linux-gnu-gcc/g++` as the active C/C++ compiler, with `-march=native` stripped from compiler flags and replaced with `-march=armv8-a`, since native CPU targeting is meaningless when cross-compiling. With `CGO_ENABLED=1`, `GOOS=linux`, and `GOARCH=arm64` set, Go's CGO toolchain links against the `arm64` `dlib` libraries to produce an `arm64` binary, despite running on an `amd64` build host. The final stage uses a `scratch` image purely as an export target, from which the compiled binary is extracted to the host.

Listing 12: Cross-Compiling the Go Binaries

```
1  make all          # cross-compiles the binary, output written to ./
   output
2  make models      # downloads the dlib model files required at
   runtime
```

The required `dlib` model files are downloaded once via the `models` target and placed alongside the binaries under `./models`, since both `cameraSession` and `extractEm`

beddings load models from a relative `models` directory at runtime. The resulting binaries (and the `models` directory) are copied into `./bin` in the Pi application build context before the application image itself is built.

Application Image

The application image build is split into two stages. The build stage, based on `node:24-slim`, installs all dependencies (including dev dependencies needed for compilation), generates the Prisma client from the schema, pre-bundles the AdminJS custom components ahead of time using `@adminjs/bundler`, and compiles the TypeScript source with `tsc`. The pre-built Go binaries under `./bin` are carried through into the final image alongside the compiled application.

The production stage, also based on `node:24-slim`, installs only the native runtime libraries required by the Go face-recognition binaries (`dlib`, `libjpeg`, and `BLAS/LAPACK/OpenBLAS`) along with production Node dependencies only (`npm ci --omit=dev`). It then copies over the compiled JavaScript output, the Prisma schema and generated client, the pre-built AdminJS bundle, and the `bin/` directory containing the two Go binaries and their models. The original TypeScript source under `src/` is also copied into the final image, since AdminJS component loader validates that the source file paths registered at build time still exist on disk at startup. File ownership is set to the non-root `node` user before the container drops privileges, and the application listens on port 80.

Listing 13: Building and Running the Application Image

```
1  make build      # docker build -t officesense:latest .
2  make run        # run the production image
3  make dev        # run against live-mounted source, for development
4  make migrate    # apply Prisma migrations against the SQLite
                    database
5  make format     # run Prettier inside a throwaway container
```

The `run` target starts the image with the camera device (`/dev/video0`) passed through, the SQLite database directory bind-mounted from the host so data persists across container restarts, and the administrative credentials, cookie secret, and database URL supplied as environment variables. The `dev` target instead mounts the project source directly into a plain `node:24-slim` container, installs the same native libraries on the fly, and runs the application with `tsx` for fast iteration without rebuilding the image.

Supporting Services

Redis, Mosquitto, Node-RED, and InfluxDB are brought up independently via `docker-compose.yml`, decoupling their lifecycle from the application image and allowing them to be restarted, reconfigured, or upgraded without rebuilding the Pi application itself.

Listing 14: `docker-compose.yml`

```
1  services:
2    mosquitto:
3      image: eclipse-mosquitto:latest
4      ports: ["1883:1883"]
5      volumes:
6        - ./mosquitto/config:/mosquitto/config
7        - ./mosquitto/data:/mosquitto/data
8        - ./mosquitto/log:/mosquitto/log
9
10   nodered:
11     image: nodered/node-red:latest
12     ports: ["1880:1880"]
13     volumes:
14       - nodered_data:/data
15
16   redis:
17     image: redis:7
18     ports: ["6379:6379"]
19     command: ["redis-server", "/etc/redis/redis.conf"]
20     volumes:
21       - ./redis/data:/data
22       - ./redis/redis.conf:/etc/redis/redis.conf
23
24   influxdb:
25     image: influxdb:2
26     ports: ["8086:8086"]
27     volumes:
28       - ./influxdb/data:/var/lib/influxdb2
29       - ./influxdb/config:/etc/influxdb2
```

All four services use official upstream images and persist their state through bind mounts or named volumes, so that broker configuration, retained MQTT state, dashboard flows, and time-series event data all survive container restarts and image upgrades. The application image connects to all four over the Pi's local network at startup, as described in Section 3: Mosquitto on port 1883 for MQTT, Redis on port 6379 for session and occupancy state, Node-RED on port 1880

for the public dashboard (which itself polls the application's HTTP API rather than communicating with Redis or MQTT directly), and InfluxDB on port 8086 for room transition event logging.

13 Configuration

Each component of the system is configured independently, reflecting its deployment context.

Tags

Tag configuration is managed through the serial CLI described in Section 11.1. After flashing, the tag boots with compile-time defaults (`main.h`) and can be reconfigured over USB without reflashing:

Listing 15: Tag Configuration via CLI

```
1  set uuid <uuid>          assign the tag's broadcast UUID
2  set name <name>          set the advertised device name
3  set adv_interval <ms>    set advertising interval in
    milliseconds
4  save                     persist changes to NVS flash
5  reboot                  restart with new configuration
```

Each tag must be assigned a UUID that matches a registered entry in the Pi's SQLite database, otherwise its advertisements will be rejected by the scanner's UUID validation step.

Scanners

Scanner configuration follows the same NVS-backed CLI approach, accessed over the USB-Serial interface. The key settings that must be configured per deployment are the WiFi credentials, the MQTT broker address, and the lookup API URL:

Listing 16: Scanner Configuration via CLI

```
1  set wf.ssid <ssid>        WiFi network name
2  set wf.pass <password>    WiFi password
3  set wf.ip / wf.gw / wf.subnet <value> static IP (optional)
4  set mq.host <host>        MQTT broker IP address
```

```
5   set mq.port <port>           MQTT broker port (default: 1883)
6   set mq.user / mq.pass       MQTT credentials
7   set api.url <url>           Pi application base URL
8   set dev.name <uuid>         scanner's room identifier (UUID)
9   save
10  reboot
```

The `dev.name` value is particularly important: it becomes the last segment of the MQTT topic the scanner publishes on (`scanners/<dev.name>`), and the Pi decision engine extracts it as the room identifier from incoming messages. Each scanner must therefore be assigned the UUID of the room it is physically installed in, as registered in the Pi database.

Camera and Go Binaries

Camera configuration (the capture device path and frame resolution) is defined in the Go source code of `cameraSession`. Changing these values requires modifying the source, cross-compiling the binary, copying the updated binary into `./bin` in the Pi application build context, and rebuilding the Pi application Docker image with `make build`.

The device path used at container runtime can be overridden without rebuilding by passing the camera device as an environment variable when starting the container, which the `run` Makefile target forwards to the `--device` flag passed to Docker. This allows the correct `/dev/video*` node to be set per deployment without touching the image.

Raspberry Pi Application

The Pi application reads its configuration from a layered config file with built-in defaults, which can be selectively overridden by setting environment variables at container startup without requiring an image rebuild.

The administrative credentials (`ADMIN_EMAIL`, `ADMIN_PASSWORD`, `ADMIN_COOKIE_SECRET`) and the database URL (`DATABASE_URL`) must always be supplied as environment variables at runtime and are never baked into the image.

Prisma database migrations must be applied once before first run, or after any schema change, using:

Listing 17: Applying Database Migrations

```
1 make migrate
```

Mosquitto

The Mosquitto MQTT broker is configured via a `mosquitto.conf` file bind-mounted into the container from `./mosquitto/config`. At minimum, the listener port, authentication settings (username/password file), and persistence options should be configured here before starting the container.

Redis

Redis is configured via a `redis.conf` file bind-mounted from `./redis/redis.conf`. The configuration must enable keyspace expiration event notifications, since the Pi application relies on Redis publishing key expiry events to detect user session timeouts:

Listing 18: Required Redis Configuration

```
1 notify-keyspace-events Ex
```

A password should also be set in `redis.conf` and matched in the Pi application's environment configuration.

Node-RED

The Node-RED dashboard is configured through its web UI, accessible on port 1880. After deployment, the two HTTP request nodes in the OfficeSense flow must be updated to point to the Pi application live data API address. Once the correct IP is set, the flow is deployed via the Node-RED editor Deploy button, after which the dashboard begins polling and updating automatically.

14 Problem Resolution

The system was designed to address three interconnected requirements: detecting user presence at room level using BLE technology, ensuring user data privacy

in compliance with GDPR principles, and verifying user identity through camera-based authentication to prevent tag misuse. The implemented system addresses each of these directly.

Room-Level Presence Detection

Room-level presence is achieved through a network of BLE scanners, each installed in a specific room and uniquely identified by the UUID of the room it occupies. BLE tags carried by users broadcast their identifier continuously. Scanners capture these advertisements, apply a median-then-EMA filtering pipeline to produce a stable RSSI estimate, and publish the result to the Pi over MQTT. The Pi decision engine determines which room a user is located in by identifying which scanner is reporting a signal for their tag.

To prevent spurious room transitions caused by signal fluctuations near room boundaries, the system implements a hysteresis-and-debounce state machine: a new room is only accepted as the user's location once its signal has consistently and significantly exceeded the current room's signal for a minimum number of samples over a minimum time window, or once the current room's signal has been lost entirely. Absence is detected passively through Redis key expiration, requiring no explicit "leave" event.

Privacy by Design

No personally identifiable information is exposed outside the system's trusted boundary at any point during normal operation. BLE tags broadcast only a pseudonymous UUID, with no name, MAC address, or other identifying attribute. When a tag is first detected, the Pi generates a fresh random pseudonym (a randomly chosen nickname and a new UUID) for that session, and all data visible through the public-facing NGSI-LD API and Node-RED dashboard uses only these pseudonyms. The mapping between a real user identity and their current pseudonym is computed on demand and exposed only through an authenticated, session-protected administrative endpoint, never through any public interface.

Real user identities are stored exclusively in SQLite, accessed only server-side, and never placed in Redis or transmitted to the dashboard. Face embeddings, which could constitute biometric data under GDPR, are stored in the same protected SQLite database and are never returned through any API endpoint. This

architecture ensures that an observer of the public dashboard, the MQTT broker traffic, or the Redis state cannot recover any personally identifiable information.

Identity Verification and Tag Misuse Prevention

The camera-based verification pipeline addresses the risk of a user presenting another person's BLE tag to falsely register their presence. Periodically, the system gathers all unverified active sessions, passes their stored face embeddings to the `cameraSession` binary alongside a live camera frame, and marks users as verified only if their stored descriptor matches a face actually detected in the frame. A successful verification expires automatically after a configurable timeout, requiring continued physical presence in front of the camera to maintain a verified status.

Users whose tag is detected but whose face cannot be matched to the stored embedding remain marked as unverified, and this status is surfaced explicitly in the dashboard through the authentication status field. This allows administrators and security personnel to identify sessions where the BLE-detected identity and the camera-detected identity are inconsistent, flagging potential cases of tag misuse without requiring manual monitoring.

System Integration

Beyond the three core requirements, the implementation delivers a complete, deployable IoT system: an edge computing architecture where all processing logic and face recognition run locally on a Raspberry Pi, with supporting services hosted on a remote server for scalability and operational convenience, a hybrid storage layer (SQLite for persistent data, Redis for real-time state), two separate dashboards (a public anonymized presence view and a private administrative panel), a standards-compliant NGSI-LD data API, a full administrative interface for managing users, rooms, and tags and enrolling face embeddings and finally, a containerized deployment stack that can be brought up on any Raspberry Pi with Docker installed.

15 Usage

This section describes the end-to-end operator workflow for deploying and running OfficeSense, from initial startup through user enrollment to verifying correct

system operation.

Step 1: Start Supporting Services

On the Raspberry Pi, navigate to the directory containing `docker-compose.yml` and bring up all supporting services:

Listing 19: Starting Supporting Services

```
1 docker compose up -d
```

This starts Mosquitto, Redis, Node-RED, and InfluxDB as background containers. Verify all four are running with `docker compose ps` before proceeding.

Step 2: Apply Database Migrations

Before starting the application for the first time (or after any schema change), apply Prisma migrations to initialize the SQLite database:

Listing 20: Applying Migrations

```
1 make migrate
```

Step 3: Start the Application

Start the Pi application container, passing the required environment variables:

Listing 21: Starting the Application

```
1 make run
```

The application will flush Redis, seed room occupancy counters, connect to Mosquitto and Redis, and start the camera verification loop.

Step 4: Flash and Configure Tags

For each BLE tag, flash the firmware using ESP-IDF as described in Section 12, then connect over USB and configure its identity via the serial CLI:

Listing 22: Configuring a Tag

```
1  set uuid <uuid>          must match a Tag record in the database
2  set name X6TAG
3  set adv_interval 500
4  save
5  reboot
```

The UUID assigned here must be registered in the database in Step 6 before the tag advertisements will be accepted by scanners.

If a scanner receives a UUID that is not registered in the database, it will be marked as invalid in the cache and won't be processed until the configured TTL runs out and the record is discarded.

Step 5: Flash and Configure Scanners

For each BLE scanner, flash the firmware using PlatformIO, then configure it over USB serial:

Listing 23: Configuring a Scanner

```
1  set wf.ssid <ssid>
2  set wf.pass <password>
3  set mq.host <pi-ip>
4  set mq.user <user>
5  set mq.pass <password>
6  set api.url http://<pi-ip>
7  set dev.name <room-uuid>    must match a Room record in the
  database
8  save
9  reboot
```

Each scanner `dev.name` must match the UUID of the room it is physically installed in, as registered in the database.

Step 6: Enroll Rooms, Users, and Tags via AdminJS

Navigate to the administrative dashboard at <http://<pi-ip>/admin> and log in with the configured admin credentials. Enrollment is performed in the following order:

Create Rooms

In the Rooms resource, create one record per physical room. The UUID generated by AdminJS on record creation is the value that must be assigned to the corresponding scanner `dev.name` in Step 5.

Create Users

In the Users resource, create one record per employee, providing their first and last name.

Assign Tags to Users

In the Tags resource, create one Tag record per physical tag and associate it with the correct User. The Tag UUID generated on creation is the value that must be assigned to the corresponding tag in Step 4.

Upload Face Photos

For each User record, open the record's detail view and select the Upload Face action. Upload a clear JPEG photo of the user containing exactly one face. The `extractEmbeddings` binary processes the image, extracts a 128-dimensional face descriptor, and stores it on the user's record. A green **SET** indicator on the record confirms a descriptor has been saved successfully. Users without a stored embedding will still be detected via BLE but will remain permanently unverified by the camera pipeline.

Step 7: Configure Node-RED

Navigate to Node-RED at `http://<pi-ip>:1880`, open the OfficeSense flow, and update the IP address in both HTTP request nodes to point to the Pi application live data API. Deploy the flow. The dashboard at `http://<pi-ip>:1880/dashboard` will begin updating automatically.

Step 8: Verify System Operation

With tags powered on and carried by enrolled users, the system should reach a fully operational state within a few seconds of the first BLE advertisement being

received. The following indicates correct end-to-end operation:

- **Node-RED dashboard** (`/dashboard`): the Users table shows active sessions with pseudonymized names, room assignments, RSSI values, and authentication status. The Rooms table and bar chart show non-zero occupancy for rooms with detected users.
- **AdminJS dashboard** (`/admin`): the custom landing page shows a live mapping of active Redis sessions to real user identities, confirming that pseudonymization is functioning and that real names are visible only within the authenticated administrative interface.
- **Authentication status**: users with a stored face embedding should transition from `unverified` to `verified` within one camera polling interval after appearing in front of the camera, and revert to `unverified` after the configured verification timeout elapses.
- **Room transitions**: carrying a tag between rooms should result in the user's room assignment updating in the dashboard after the debounce period, with occupancy counters adjusting accordingly.
- **Absence detection**: powering off a tag or moving it out of range should result in the user's session expiring from Redis after the configured TTL, with the room occupancy counter decrementing automatically.

16 Improvements

Several targeted improvements could extend the system capabilities, reliability, and scalability beyond its current implementation.

Movement Analysis with InfluxDB and Grafana

The system currently writes room transition events to InfluxDB, recording enter and leave events per user (pseudonymized), per room, with RSSI and timestamp, but does not yet perform any analysis on this data. This historical record is a foundation for a range of data-driven insights that could be built on top of the existing infrastructure without modifying the core system. Examples include: peak occupancy hours per room, average time spent per room per user session, movement

pattern analysis across the working day, detection of anomalous behavior (e.g. a tag appearing in two rooms simultaneously due to signal leakage), and long-term attendance trends. Grafana, which integrates natively with InfluxDB, could be configured to provide a ready-made visualization and alerting layer for these analyses with minimal additional configuration.

Supporting Services on a Secondary Server

Rather than running all services on the Raspberry Pi, the deployment separates concerns between two machines: the Pi handles all real-time processing (BLE message ingestion, room transition logic, face recognition via the camera), while Redis, Mosquitto, and InfluxDB run on a remote VPS. Because all three services are connected to the Pi application purely over IP (hostname and port, configurable via environment variables), this separation required no code changes. The result is that the Pi can dedicate its constrained resources to latency-sensitive workloads, while the VPS handles brokering, session storage, and time-series logging independently. Fault isolation also improves as a broker or database restart on the VPS does not interrupt the Pi local processing pipeline.

Rotating Pseudonyms

The current pseudonymization scheme assigns a random pseudonym to each user session and retains it for the session's lifetime. A stronger privacy guarantee could be achieved by rotating pseudonyms periodically within an active session, making it harder to correlate a user's movements over time even from the public dashboard. This would require coordinating pseudonym rotation across Redis, the NGSI-LD API, and the InfluxDB event log, but the current architecture's clean separation between real identity (SQLite) and session state (Redis) makes such a change feasible without restructuring the data model.

Improved Redis Concurrency and Data Structures

The current implementation accesses Redis through sequential `await` calls, processing one user's session state at a time. At low user counts this is acceptable, but it becomes a bottleneck as the number of concurrent sessions grows. A significant improvement would be replacing sequential calls with Redis pipelines or

`MULTI/EXEC` transactions, batching multiple reads and writes into a single round-trip per processing cycle.

Beyond concurrency, the session storage model itself can be improved. Currently, each user session is stored as a JSON-serialized string under a single key. Replacing this with Redis Hashes (`HSET` / `HGET`) allows individual session fields — RSSI, room, verification status, timestamp — to be updated atomically in isolation without deserializing and reserializing the entire object on every BLE message. Room occupancy counters already use atomic `INCR` and `DECR` operations, applying the same field-level atomicity to session data would reduce the risk of partial updates under concurrent writes.

These changes are also a prerequisite for running multiple instances of the Pi core service. As it stands, the single-instance model means a restart of the application leaves a window during which incoming MQTT messages are not processed. A multi-instance deployment, with each instance subscribed to the same MQTT topic and writing to a shared Redis, requires that all session state mutations be atomic and idempotent. Using Hash fields with conditional writes (e.g., `SET ... NX` for session initialization, `HSET` for field updates) ensures that two instances processing the same tag reading concurrently produce a consistent result rather than corrupting shared state.

Distributed Face Recognition via a Worker Pool

Camera-based verification is currently performed by a single `cameraSession` subprocess invoked on the Raspberry Pi. This couples verification throughput directly to the Pi CPU, which must simultaneously handle BLE message processing, MQTT I/O, and Redis state management. As the number of enrolled users grows, the face descriptor comparison loop (even though parallelized within a single process invocation) runs on the same constrained hardware.

A more scalable architecture would separate face recognition into a dedicated worker pool running on one or more external machines. Rather than spawning a local subprocess, the Pi core service would push verification tasks to a message queue, each task containing the set of stored face descriptors to be compared along with a session token. Workers on remote machines would consume tasks from the queue, capture or receive a camera frame, perform descriptor comparison using the same `dlib`-based pipeline as the current `cameraSession` binary, and

publish results back to a response queue. The Pi service would then consume these results asynchronously and update verification status in Redis.

This design decouples verification latency from Pi load and allows the worker pool to be scaled horizontally by adding more worker machines. It also allows higher-resolution captures or more computationally expensive recognition models to be used without impacting the Pi real-time BLE processing pipeline. The camera itself may remain attached to the Pi (with frames forwarded to workers as JPEG payloads in queue messages) or be moved to a dedicated capture device with direct network access to the worker pool.

Per-Room Cameras for Presence Confirmation and Location Disambiguation

The current architecture relies on a single camera connected to the Raspberry Pi, which provides verification only for users within its field of view. This creates a fundamental limitation: a user detected by BLE in a room that is not visible to the central camera will remain permanently unverified, and the camera pipeline offers no additional signal for resolving ambiguous room assignments.

Deploying one camera per room addresses both limitations simultaneously. Each room's camera would observe only the users physically present in that room, making face verification meaningful regardless of which room a user is in. More importantly, camera-based presence confirmation becomes a second, independent input to the location estimation logic, complementing the RSSI-based signal from BLE scanners.

In the current system, room assignment is determined exclusively by comparing RSSI values across scanners, which is inherently noisy near room boundaries. A per-room camera introduces a high-confidence presence signal: if a user's face is recognized in the camera feed of Room A, that constitutes strong evidence that the user is physically in Room A, regardless of what the RSSI values suggest. This confirmation could be integrated into the room transition state machine as a fast-path override (bypassing the hysteresis and debounce requirements when camera identity and BLE tag identity agree on the same room) or as a tiebreaker when two scanners report similar signal strengths for the same tag.

The main architectural change required is extending the verification pipeline to be

room-aware. Rather than a single polling loop that collects all unverified sessions and passes them to one `cameraSession` invocation, the service would maintain one verification worker per room, each consuming frames from its assigned camera and checking only the users currently believed to be in that room. Results from each worker would carry the room identifier alongside the verification outcome, allowing the decision engine to cross-check whether the room in which a face was recognized matches the room currently assigned to that user's BLE session.

This improvement integrates naturally with the distributed worker pool described above: each per-room camera worker could run on the same remote worker machines, with the camera either attached directly to the worker or streamed over the network to it, keeping the Raspberry Pi free of all image processing workload.