# **Smart Attendance System Project Report**

This report details the architecture, components, and functionality of the smart attendance system based on the provided Python server code and Arduino ESP32-CAM client code. The system leverages face recognition and MQTT communication to automate attendance tracking for different classes.

### **1. Project Overview and Architecture**

The project follows a client-server architecture utilizing the MQTT protocol as the communication backbone.

* **Client:** ESP32-CAM devices, each dedicated to a specific class. They are responsible for capturing images, initiating class sessions, and sending data (start commands and images) to the server via MQTT. They also receive feedback (attendance confirmations) via MQTT.
* **Server:** A Python application running on a central machine (your laptop hosting the Mosquitto broker). It receives data from the ESP32-CAMs, performs face recognition, manages active class sessions, records attendance, generates attendance logs, and emails them to the respective doctors.
* **MQTT Broker:** A central message broker (your Mosquitto instance on Lenovo.local) facilitating communication between multiple ESP32-CAM clients and the single Python server.

The system assumes that an external mechanism, likely involving NFC/RFID scanning by a doctor, triggers the "start session" command, providing the class name and the doctor's email address. However, the provided code snippets themselves do not contain the NFC/RFID scanning logic. They handle the *result* of that action, which is the sending of the start command via MQTT.

### **2. Code 1: Python Server Analysis**

This Python script acts as the central processing unit and management hub of the system.

**Purpose:** To receive attendance requests and images via MQTT, perform face recognition against pre-trained models for specific classes, record attendance, manage class session timers, and email attendance logs to the designated doctor.

**Key Libraries Used:**

* os, csv, time: Standard libraries for file system operations, CSV handling, and time/date.
* base64: For decoding images received in Base64 format.
* cv2 (OpenCV): Essential for face detection (CascadeClassifier) and face recognition (LBPHFaceRecognizer).
* numpy: Used with OpenCV for image data manipulation.
* paho.mqtt.client: The library for connecting to the MQTT broker and handling messages.
* smtplib, email.mime.\*: Libraries for sending emails with attachments.
* datetime, timedelta: For managing session start/end times and durations.
* threading: Used to run image processing in a separate thread and schedule session finalization (Timer).
* io, PIL (Pillow): For handling image data in memory before processing with OpenCV.
* json: For parsing incoming JSON payloads from MQTT messages and constructing outgoing JSON payloads.

**Configuration:**

* **Directories:** Defines REGISTERED\_DIR, ATTENDANCE\_DIR, SAVED\_IMAGES\_DIR for managing local files.
* **SMTP:** SMTP\_SERVER, SMTP\_PORT, SMTP\_USERNAME, SMTP\_PASSWORD, FROM\_EMAIL are set for configuring outgoing emails. **Note:** Using Gmail with SMTP\_PASSWORD likely requires generating an App Password if two-factor authentication is enabled.
* **MQTT:** BROKER (Lenovo.local), TOPIC\_IMAGE, TOPIC\_START, TOPIC\_CONFIRM\_BASE define the communication endpoints and message types.

**Global Variables:**

* recognizers: Dictionary {class\_name: cv2.face.LBPHFaceRecognizer} storing the trained recognition model for each class.
* label\_map: Nested dictionary {class\_name: {label\_id: name}} mapping the numerical output of the recognizer back to the person's name.
* active\_classes: Dictionary {class\_name: end\_time (datetime)} tracking which class sessions are currently active and when they are scheduled to end.
* recorded\_today (Should be recorded\_this\_session): Dictionary {class\_name: set(names)} storing the names of individuals who have already been marked present in the current session for a given class to avoid duplicates.
* doctor\_emails: Dictionary {class\_name: doctor\_email} storing the email address provided in the START message for each active class.
* client: The global instance of the paho.mqtt.client.Client.

**Key Functions:**

* train\_models(): Iterates through the REGISTERED\_DIR, loads grayscale images for each class, uses haarcascade\_frontalface\_default.xml to detect faces (though LBPH training typically uses pre-cropped faces, the code *attempts* detection here), assigns labels based on filenames, trains an LBPHFaceRecognizer for each class, and populates recognizers and label\_map.
* send\_email(to\_email, subject, body, attachment\_path): Constructs an email with the provided details, attaches the specified CSV file, and sends it using the configured SMTP server. Includes basic error handling for file access and SMTP connection/authentication.
* mark\_attendance(class\_name, name): Appends the name and current timestamp to the CSV file for the given class in ATTENDANCE\_DIR. It checks the recorded\_today set to prevent duplicate entries within the same session. Crucially, it also **publishes an MQTT confirmation message** to attendance/confirm/{class\_name} containing a JSON payload { "status": "success", "name": name } (if connected to the broker).
* finalize\_and\_email(class\_name): Called when a session timer expires. Retrieves the doctor's email, checks if the attendance CSV exists and has data, and calls send\_email to send the report. It then cleans up the session's state (active\_classes, doctor\_emails, recorded\_today) and optionally clears the attendance CSV file.
* process\_image(class\_name, b64\_image): The core image processing function. Checks if the class is active. Decodes the Base64 image string. Optionally saves the raw received image. Converts the image to grayscale and a NumPy array. Uses the Haar Cascade to detect faces. If faces are found, it crops the first detected face. Uses the recognizers and label\_map for the specific class\_name to predict the identity (label, confidence). Compares the confidence to CONFIDENCE\_THRESHOLD. If confidence is below the threshold and the label is mapped to a name, it calls mark\_attendance. Handles various potential errors (decoding, OpenCV, no face detected, no model found). Runs in a separate thread to avoid blocking the MQTT loop.
* on\_connect(client, userdata, flags, rc): MQTT callback function executed upon connecting to the broker. Subscribes to attendance/start and attendance/image.
* on\_message(client, userdata, msg): MQTT callback function executed when a message is received on a subscribed topic.
  + If topic is attendance/start: Parses JSON payload to get class and email. Checks if the class is already active or if a trained model exists. If valid and not active, it sets the session end time in active\_classes, stores the doctor\_email, resets the recorded\_today set for that class, and schedules finalize\_and\_email using a threading.Timer.
  + If topic is attendance/image: Parses JSON payload to get class and image (Base64 string). Checks if the class is active. Starts a new threading.Thread to run process\_image, passing the class name and image data.
* on\_disconnect(client, userdata, rc): MQTT callback for handling disconnections.
* on\_log(client, userdata, level, buf): Optional callback for detailed MQTT client logging.

**Face Recognition Model Parameters (LBPH):**

The code uses the Local Binary Patterns Histograms (LBPH) algorithm provided by OpenCV (cv2.face.LBPHFaceRecognizer\_create()). The default parameters for LBPHFaceRecognizer\_create() are used (typically radius=1, neighbors=8, grid\_x=8, grid\_y=8).

The crucial parameter controlled by your code is the **CONFIDENCE\_THRESHOLD = 60**. In LBPH, the confidence score represents a distance; *lower values mean a better match*. Your code considers a face recognized if the confidence score returned by recognizer.predict() is *less than* this threshold. A value of 60 is a common starting point, but tuning this threshold based on your environment and desired accuracy (balancing false positives vs. false negatives) is recommended.

### **3. Code 2: Arduino ESP32-CAM Client Analysis**

This C++ sketch runs on each individual ESP32-CAM device assigned to a specific class.

**Purpose:** To connect to the local WiFi network and the MQTT broker, capture images on demand (via serial command in this version), encode them as Base64 strings, publish session start commands and images to the MQTT broker, and receive and indicate attendance confirmations.

**Key Libraries Used:**

* WiFi.h: For connecting to the WiFi network.
* PubSubClient.h: For connecting to the MQTT broker and handling message publishing and subscribing.
* esp\_camera.h: The Espressif library for controlling the ESP32-CAM module.
* base64.h: A library for performing Base64 encoding on the ESP32 (needs to be installed separately in the Arduino IDE).

**Configuration:**

* ssid, password: WiFi network credentials.
* mqtt\_broker (Lenovo.local), mqtt\_port (1883): MQTT broker details.
* class\_name: **Hardcoded** string identifying which class this specific ESP32-CAM unit belongs to (e.g., "classA"). This is how the device knows its identity.
* doctor\_email: **Hardcoded** email address for the doctor associated with this class. This email is sent in the START command payload.
* FLASH\_GPIO\_NUM: The GPIO pin connected to the flash LED for visual feedback.
* Camera configuration (PWDN\_GPIO\_NUM to PCLK\_GPIO\_NUM): Pin definitions specific to the camera module on the ESP32-CAM board (AI Thinker model pinout assumed).
* MQTT\_BUFFER\_SIZE: Crucially sets the buffer size for the PubSubClient. This *must* be large enough to hold the full MQTT message payload, including the Base64 encoded image and JSON overhead. 50KB is a reasonable attempt but might need tuning based on image quality/resolution.

**Global Variables:**

* espClient: The WiFi client instance used by PubSubClient.
* client: The PubSubClient instance for MQTT communication.
* confirmationTopic: A String constructed in setup() as "attendance/confirm/" + class\_name. This is the unique topic this device subscribes to for receiving confirmations.

**Key Functions:**

* setup(): Initializes Serial communication, the flash LED pin, constructs confirmationTopic, initializes the camera module (esp\_camera\_init) with specified resolution and quality (adjusting based on PSRAM), sets the MQTT server and callback, sets the MQTT buffer size, connects to WiFi, and finally attempts to connect to the MQTT broker by calling reconnect().
* loop(): The main program loop. Continuously calls client.loop() to maintain the MQTT connection and process incoming messages. Also checks for serial input ('s' or 'q') to trigger actions.
* reconnect(): Handles connecting and reconnecting to the MQTT broker. Generates a unique client ID. Once connected, it subscribes to its specific confirmationTopic. Includes retry logic with a delay.
* mqttCallback(char\* topic, byte\* payload, unsigned int length): The callback function for incoming MQTT messages. It checks if the received topic matches the device's confirmationTopic. If it does, it prints a confirmation message to Serial and calls blinkFlash(). It includes commented-out code showing how to parse the JSON confirmation payload using the (required but not included) ArduinoJson library to potentially extract information like the recognized name.
* blinkFlash(int count, int delay\_ms): Toggles the FLASH\_GPIO\_NUM pin multiple times to provide visual feedback (a blinking light).
* sendStartCommand(): Checks WiFi and MQTT connection. Constructs a simple JSON payload {"class":"...", "email":"..."} using String concatenation. Publishes this payload to the attendance/start topic.
* captureAndSendImage(): Checks WiFi and MQTT connection. Attempts to capture an image using esp\_camera\_fb\_get(). If successful, it gets the raw image data (fb->buf, fb->len). It uses the base64::encode() function (from the included base64.h) to convert the JPEG image bytes into a Base64 string. It constructs a JSON payload {"class":"...", "image":"..."}. It calculates the estimated payload size and checks it against MQTT\_BUFFER\_SIZE. If the buffer is large enough, it publishes the payload to the attendance/image topic using client.beginPublish(), client.print(), and client.endPublish() to handle potentially large messages. Returns the camera frame buffer using esp\_camera\_fb\_return(). Includes error checks for capture, encoding, and publishing.
* Serial Input: The loop() function allows triggering sendStartCommand() by typing 's' and pressing Enter in the Serial monitor, and captureAndSendImage() by typing 'q' and pressing Enter.

### **4. Messages and Data Structures ("Shapes")**

Communication between the server and clients happens via MQTT messages. These messages use a specific structure, primarily JSON, which can be considered their "shape".

**MQTT Messages:**

* **START Message:**
  + **Topic:** attendance/start
  + **Sender:** ESP32-CAM client (or gateway triggered by NFC/RFID).
  + **Purpose:** To signal the server that a new attendance session is starting for a specific class and to provide the doctor's email for reporting.
  + **Shape (JSON Payload):**  
    {  
     "class": "string", // The name of the class (e.g., "classA")  
     "email": "string" // The doctor's email address (e.g., "doctor@example.com")  
    }
  + **Used For:** Initiating a timed attendance session on the server. The server uses class to identify which model/log file to use and email for the final report recipient.
* **IMAGE Message:**
  + **Topic:** attendance/image
  + **Sender:** ESP32-CAM client.
  + **Purpose:** To send a captured image to the server for face detection and recognition.
  + **Shape (JSON Payload):**  
    {  
     "class": "string", // The name of the class this image is from  
     "image": "string" // The Base64 encoded JPEG image data  
    }
  + **Used For:** Providing the server with visual data to perform face recognition and potentially mark attendance. The server uses class to select the correct recognition model and the Base64 image data for processing.
* **CONFIRMATION Message:**
  + **Topic:** attendance/confirm/{class\_name} (where {class\_name} is dynamically replaced by the specific class name, e.g., attendance/confirm/classA)
  + **Sender:** Python Server.
  + **Purpose:** To notify the specific ESP32-CAM client that attendance has been successfully marked for someone from that class.
  + **Shape (JSON Payload):**  
    {  
     "status": "string", // Indicates the outcome (e.g., "success")  
     "name": "string" // The name of the person whose attendance was marked (optional, included by server)  
    }
  + **Used For:** Providing real-time feedback to the ESP32-CAM device. The client receives this and triggers a visual indicator (blinking flash) to show that a recognition/attendance event occurred.

**Other Data Structures ("Shapes"):**

* **Attendance Log CSV:**
  + **Location:** Server's attendance\_logs directory.
  + **Shape (CSV Format):**  
    Name,Timestamp  
    Alice,2023-10-27 10:30:00  
    Bob,2023-10-27 10:35:15  
    Charlie,2023-10-27 10:40:00
  + **Used For:** Storing the attendance records persistently on the server until the session ends and the email is sent (and optionally the file is cleared).
* **recognizers, label\_map (Server):** These dictionaries store the trained face recognition data and mappings in specific structures required by the OpenCV LBPH implementation and the server's logic.
* **active\_classes, recorded\_today, doctor\_emails (Server):** These dictionaries maintain the state of the active sessions, recorded attendees, and doctor contacts using class names as keys.

### **5. System Workflow**

1. **Training:** Before deployment, face images for each class are placed in the registered\_users/{class\_name} directories on the server. The Python server is started, runs train\_models(), creating and storing an LBPH recognizer and label map for each class.
2. **ESP32-CAM Setup:** Each ESP32-CAM is flashed with the Arduino code, configured with its specific ssid, password, mqtt\_broker, class\_name, and doctor\_email. They connect to WiFi and the MQTT broker, subscribing to their confirmation topic (attendance/confirm/{class\_name}).
3. **Session Start:** An external mechanism (e.g., a separate device reading an NFC/RFID card associated with a class and doctor email) sends an MQTT message to the attendance/start topic with the class name and doctor's email. In the provided code, this is simulated by typing 's' in the ESP32-CAM's serial monitor.
4. **Server Receives Start:** The Python server's on\_message callback receives the attendance/start message. It parses the JSON, records the class as active, stores the doctor's email, resets the session's recorded\_today set, and starts a timer to call finalize\_and\_email after a set duration (10 minutes in the code).
5. **Image Capture & Send:** During the active session, the ESP32-CAM captures images (simulated by typing 'q' in the serial monitor). The captureAndSendImage() function gets a frame, encodes it in Base64, constructs a JSON payload with the class\_name and the Base64 image, and publishes it to the attendance/image topic.
6. **Server Receives Image:** The Python server's on\_message receives the attendance/image message. It parses the JSON and passes the class name and image data to process\_image in a separate thread.
7. **Face Recognition:** The process\_image function decodes the image, detects faces, and uses the trained LBPH model for the specified class to predict the identity of detected faces.
8. **Attendance Marking:** If a face is recognized with sufficient confidence (below the CONFIDENCE\_THRESHOLD) and hasn't been recorded in the current session, the server's mark\_attendance function appends the name and timestamp to the class's CSV file.
9. **Server Sends Confirmation:** After marking attendance, mark\_attendance publishes a confirmation message ({"status": "success", "name": "..."}) to attendance/confirm/{class\_name} for that specific class.
10. **Client Receives Confirmation:** The relevant ESP32-CAM receives the confirmation message in its mqttCallback. It checks the topic and triggers the blinkFlash() function to provide visual feedback that attendance was recorded.
11. **Session End & Report:** When the timer set in step 4 expires, finalize\_and\_email is called on the server. It retrieves the attendance CSV for the class and the doctor's email. If attendance was recorded, it sends an email to the doctor with the CSV attached. It then cleans up the session's active state and optionally clears the CSV for the next session.

### **6. Role of NFC/RFID**

As mentioned, the provided code snippets **do not include the logic for reading NFC or RFID cards**.

However, they *are* designed to integrate with a system where NFC/RFID is used. The **result** of a doctor scanning their card (which presumably contains or is linked to their email and associated class) is expected to be the triggering of an MQTT message published to the attendance/start topic with the required JSON payload ({"class": "...", "email": "..."}).

This could be achieved by:

* A separate micro controller (like another ESP32 or Arduino) with an NFC/RFID reader that reads the card data and then publishes the MQTT message.
* The ESP32-CAM itself having an NFC/RFID reader connected, and its code modified to read the card and then call sendStartCommand() automatically instead of waiting for a serial command.

The current code relies on this attendance/start message appearing on the broker, however it gets there.

### **7. Comparison Table**

Here is a comparison of the two code snippets:

| **Feature** | **Code 1: Python Server** | **Code 2: Arduino ESP32-CAM Client** |
| --- | --- | --- |
| **Role** | Central Server, Processing, Management | Client Device, Capture, Communication |
| **Platform/Device** | PC/Server (Linux, Windows, macOS) | ESP32-CAM microcontroller |
| **Language** | Python | C++ (Arduino Framework) |
| **Key Libraries** | cv2, paho.mqtt.client, smtplib, etc. | esp\_camera, PubSubClient, WiFi, base64 |
| **Core Logic** | Face Recognition, Session Timer, Emailing | Image Capture, Base64 Encoding, MQTT Pub/Sub |
| **MQTT Connection** | Connects to broker, Subscribes to START/IMAGE, Publishes CONFIRMATION | Connects to broker, Publishes START/IMAGE, Subscribes to unique CONFIRMATION |
| **Class ID** | Identified from incoming MQTT message payload ("class" field) | Hardcoded as a const String variable (class\_name) |
| **Doctor Email** | Received in START message payload, stored for session | Hardcoded as a const String variable (doctor\_email), sent in START message |
| **Session Management** | Manages session start/end times, active status, recorded attendance | Sends START command (currently manual trigger) |
| **Face Recognition** | Performs face detection and recognition | Does not perform face recognition |
| **Attendance Data** | Records attendance to CSV files | Does not store attendance data locally |
| **Reporting** | Generates and emails CSV reports | Does not generate reports |
| **Feedback** | Sends MQTT confirmation message | Receives confirmation, triggers flash blink |
| **Configuration** | Via constants at the top of the script, directory structure | Via constants at the top of the sketch |
| **Trigger (Start)** | Receives MQTT message | Currently triggered by Serial input 's' |
| **Trigger (Image)** | Receives MQTT message | Currently triggered by Serial input 'q' |

### **8. Conclusion**

The smart attendance system effectively utilizes MQTT to create a distributed system where ESP32-CAMs act as data collectors and the Python script as the central processing and reporting server. Each ESP32-CAM is uniquely identified by its hardcoded class\_name, enabling the server to process data and manage sessions on a per-class basis. The system successfully implements face recognition using OpenCV's LBPH, records attendance in CSV files, and automates the reporting process via email.

The provided code snippets demonstrate the core communication and processing pipeline, assuming the MQTT START message with class and email is initiated by an external mechanism like an NFC/RFID reader system, which would need to be implemented separately to fulfill that part of the system's requirements. The client's use of a dedicated confirmation topic and visual feedback (flash blink) provides a useful layer of interaction and confirmation at the device level. The design considerations around MQTT buffer size for image payloads in the client code are crucial for reliable operation.