Face-Recognition and attendance system

FA23-BCS-084 Nouman Ahmad

FA23-BCS-094 Syed Muhammad Sharjeel Farsheed

Lab Mid-Term Project Report

# Table of Contents

[Table of Contents 1](#_Toc211892522)

[1. Introduction 4](#_Toc211892523)

[Purpose and objectives: 4](#_Toc211892524)

[2. System Overview 5](#_Toc211892525)

[Client Layer (Browser) 5](#_Toc211892526)

[Server Layer (Flask + Socket.IO 5](#_Toc211892527)

[Storage Layer (SQLite + Disk) 5](#_Toc211892528)

[High-level objectives of each layer: 5](#_Toc211892529)

[Client: 5](#_Toc211892530)

[Server: 5](#_Toc211892531)

[Storage: 5](#_Toc211892532)

[Key constraints & assumptions: 5](#_Toc211892533)

[3. Features 6](#_Toc211892534)

[Real-time recognition 6](#_Toc211892535)

[Student registration: 6](#_Toc211892536)

[Face dataset creation: 6](#_Toc211892537)

[Embedding-based matching: 6](#_Toc211892538)

[Automatic attendance logging: 6](#_Toc211892539)

[Web client: 6](#_Toc211892540)

[Backend APIs / Socket events: 6](#_Toc211892541)

[Pre-trained detection models included 6](#_Toc211892542)

[Lightweight persistence: 6](#_Toc211892543)

[4. Project Structure 7](#_Toc211892544)

[5. System Architecture 9](#_Toc211892545)

[5.1 Component Diagram (conceptual) 9](#_Toc211892546)

[5.2 Key Modules & Dataflow 9](#_Toc211892547)

[6. Implementation Details 11](#_Toc211892548)

[6.1 Face Detection 11](#_Toc211892549)

[6.2 Face Embedding Extraction 11](#_Toc211892550)

[6.3 Matching / Recognition 11](#_Toc211892551)

[6.4 Registration Flow 12](#_Toc211892552)

[6.5 Database Schema (high level) 12](#_Toc211892553)

[6.6 Server & Socket Events 13](#_Toc211892554)

[6.7 Important Configuration & Tuning Parameters 14](#_Toc211892555)

[7. Working of the System 15](#_Toc211892556)

[7.1 Student Registration (single registration flow) 15](#_Toc211892557)

[7.2 Marking Attendance (recognition flow) 15](#_Toc211892558)

[7.3 Viewing Records 16](#_Toc211892559)

[8. Database and Models 17](#_Toc211892560)

[8.1 Models Provided 17](#_Toc211892561)

[8.2 How embeddings are stored 17](#_Toc211892562)

[8.3 Database best-practices for this project 18](#_Toc211892563)

[9. User Interface 19](#_Toc211892564)

[9.1 Layout & Navigation 19](#_Toc211892565)

[9.2 Client-side Implementation Notes 20](#_Toc211892566)

[9.3 Security & Privacy in UI 20](#_Toc211892567)

[10. Testing and Results 21](#_Toc211892568)

[10.1 Test Scenarios 21](#_Toc211892569)

[10.2 Example Evaluation Metrics 21](#_Toc211892570)

[10.3 Performance Tips 22](#_Toc211892571)

[11. Troubleshooting & Deployment Notes 23](#_Toc211892572)

[11.1 Common Problems & Fixes 23](#_Toc211892573)

[11.2 Deployment Advice 24](#_Toc211892574)

[11.3 Backup & Maintenance 24](#_Toc211892575)

[12. Conclusion 25](#_Toc211892576)

[Key takeaways: 25](#_Toc211892577)

# 1. Introduction

This document describes the Attendance System Using Facial Recognition — a local-network, web-enabled application that captures faces from a camera (webcam or browser camera), recognises registered users (students), and marks attendance automatically. The system combines classical computer-vision components (face detection, alignment, embeddings) with a lightweight web stack (Flask + Socket.IO) to provide a near real-time attendance logging solution.

## Purpose and objectives:

Provide a simple, practical demonstration of face-recognition-based attendance marking.

Offer a functional web client for phone/desktop access and a server-side recognition engine.

Preserve reproducibility and ease-of-use for classroom or small-lab deployment.

Intended audience: undergraduate students and developers who want a working example combining face detection, embedding-based recognition, and a simple web UI tied to a persistent database.

# 2. System Overview

The system consists of three primary layers:

Client Layer (Browser) — captures images via getUserMedia, shows live feed, lets users register and trigger recognition actions, and communicates with the server using Socket.IO/websockets.

Server Layer (Flask + Socket.IO) — receives images, runs detection and recognition pipelines, updates the attendance database, and streams results back to clients.

Storage Layer (SQLite + Disk) — stores student metadata, attendance records, and face image datasets / stored embeddings on disk.

## High-level objectives of each layer:

Client: simple capture/UI, minimal processing, compatibility with mobile browsers.

Server: robust face detection, embedding extraction, matching and attendance logic.

Storage: durable attendance logs, easy-to-query student records, small footprint for local deploy.

## Key constraints & assumptions:

Designed for LAN use (server and client on the same local network) — phone clients access server via local IP.

Works best with decent lighting and frontal face images.

Built for small-to-medium class sizes; SQLite chosen for simplicity.

# 3. Features

Core capabilities implemented in the repository:

Real-time recognition: Capture frames from a camera in the browser, send to server, receive recognition results quickly.

Student registration: Create student profiles with metadata (name, reg. no., semester, phone) and multiple face images.

Face dataset creation: Save captured face crops into dataset folders for each student to increase recognition robustness.

Embedding-based matching: Use a face embedding model to compute vectors; nearest-neighbour or threshold matching determines identity.

Automatic attendance logging: When a face is matched, an attendance record with timestamp is stored.

Web client: Responsive HTML/JS UI with tabs for registration, marking attendance, and viewing records.

Backend APIs / Socket events: Server emits and receives events for start/stop camera, register image, perform recognition, and refresh records.

Pre-trained detection models included: A Caffe-based face detector model (deploy.prototxt + res10 model) for initial face localization.

Lightweight persistence: SQLite database (attendance.db) for student and attendance tables.

# 4. Project Structure

A typical layout of the repository (folders and important files):

AttendaceSystemUsingFacialRecognition/

├── main/

│ ├── server.py # Flask + Socket.IO server entrypoint

│ ├── main.py # Optional desktop / legacy app launcher

│ ├── models.py # Model-loading helpers / wrappers

│ ├── database\_utils.py # DB connection and helper functions

│ ├── registration.py # Student registration flow (capture + save)

│ ├── faceDetection.py # Face detection utilities (DNN wrapper)

│ ├── faceEmbedding.py # Embedding extraction wrapper (facenet-like)

│ ├── recognition.py # Matching / recognition logic

│ └── real\_time.py # Real-time loop helpers & image processing

├── client/

│ ├── templates/

│ │ └── index.html # Main single-page client

│ └── static/

│ ├── css/

│ │ └── style.css

│ └── js/

│ └── app.js # Camera capture, socket emit/listen logic

├── models/

│ ├── deploy.prototxt

│ └── res10\_300x300\_ssd\_iter\_140000.caffemodel

├── dataset/ # Per-student face images stored here

├── database/

│ └── attendance.db # SQLite DB file (students, attendance tables)

├── requirements.txt # Python deps

└── README.md

Notes:

main/server.py is the recommended starting point for LAN/web usage.

dataset/ subfolders are typically named by registration number or a unique student id.

models/ contains detection network weights; embedding model weights are loaded by faceEmbedding.py (may download on first run or be included depending on setup).

# 5. System Architecture

This section explains the components, their responsibilities, and interactions.

## 5.1 Component Diagram (conceptual)

Client (Browser) ⇄ Server (Flask + Socket.IO) ⇄ Recognition Engine (Face Detection → Embedding → Matcher) ⇄ Storage (SQLite + Disk)

**Client ⇄ Server:** Real-time bidirectional communication via Socket.IO (image frames & recognition results).

**Server ⇄ Recognition Engine**: In-process function calls (Python). The server passes image arrays to detection/embedding functions.

**Recognition Engine ⇄ Storage:** Reads registered embeddings and student records for matching; writes attendance rows upon successful recognition.

## 5.2 Key Modules & Dataflow

**faceDetection.py**

Loads DNN detector from models/.

Receives an RGB/BGR frame, returns bounding boxes and confidence scores.

Preprocesses frames to the detector’s expected size before forward pass.

**faceEmbedding.py**

Loads embedding model (e.g., Facenet-style model).

Receives cropped/aligned face images, returns 128/512-dim embeddings (model dependent).

Provides normalization utilities (e.g., L2 normalization).

**recognition.py**

Maintains a registry of known embeddings (loaded from disk / built at runtime).

Compares a query embedding with known embeddings using cosine or Euclidean distance.

Uses a threshold or KNN nearest neighbour (k small) for decision.

Returns identity and confidence/distance score.

**database\_utils.py**

Manages SQLite connection.

CRUD operations for students and attendance tables.

Ensures safe inserts (timestamps, preventing duplicate attendance within short window).

**server.py**

Orchestrates: accepts images from client socket, calls detection + embedding + recognition, writes attendance and emits results back to client.

Hosts HTTP endpoints for the web UI.

Contains error handling and optional logging.

# 6. Implementation Details

This section describes how the major pieces were implemented, configuration options, and important code-level decisions.

## 6.1 Face Detection

**Model:** res10\_300x300\_ssd\_iter\_140000.caffemodel with corresponding prototxt.

**Preprocessing:** Resize to 300×300, subtract mean values as in model spec, and forward pass through OpenCV DNN.

Filter detections by confidence (default threshold e.g., 0.5).

**Output:**

Bounding boxes scaled back to original image coordinates.

For each box, optionally run face alignment (if alignment code exists) or pass raw crop to embedding model.

## 6.2 Face Embedding Extraction

**Model choices:** Facenet-like model or other pre-trained embedding model (the repo wraps a chosen model).

**Preprocessing:** Resize face crop to the model’s input (e.g., 160×160).

Convert to RGB, scale pixel values, apply any required normalization.

**Postprocessing:**L2-normalize embedding vectors for cosine distance matching.

## 6.3 Matching / Recognition

**Strategy:** Store embeddings for each registered student (either in a serialized index file or compute on startup scanning dataset/).

When a query embedding arrives, compute distances to all stored embeddings.

Compute minimum distance per student (if multiple embeddings per student exist) and choose the student with smallest distance.

**Thresholding:** If smallest distance < threshold (configurable e.g., 0.4 for cosine or 0.6 for Euclidean depending on model), declare match.

Otherwise return “Unknown”.

**Debouncing:** Optionally the server prevents multiple markings in quick succession for the same student (e.g., within 30 seconds) to avoid duplicate records.

**Edge cases:** Multiple faces in image: process each face independently and produce separate matches/records.

**Face too small:** ignore if width/height below minimum pixel threshold.

## 6.4 Registration Flow

The client captures multiple face images for a student and sends them to the server.

**registration.py**

Saves raw images or face-crops into dataset/<student\_id>/.

Optionally computes embeddings and stores them alongside images for faster matching.

Inserts student metadata into students table.

**Best practice:** capture varied images (angles, lighting) and at least 5-10 samples.

## 6.5 Database Schema (high level)

**students table:**

id INTEGER PRIMARY KEY AUTOINCREMENT

reg\_no TEXT UNIQUE

name TEXT

semester TEXT

phone TEXT

dataset\_path TEXT (optional)

created\_at TIMESTAMP

**attendance table:**

id INTEGER PRIMARY KEY AUTOINCREMENT

student\_id INTEGER (foreign key -> students.id)

reg\_no TEXT

name TEXT

timestamp DATETIME

status TEXT (e.g., "present")

confidence REAL (matching score)

image\_path TEXT (optional; saved crop of recognized face)

**Notes:**

Foreign key constraints may be omitted in a lightweight SQLite schema but recommended for data integrity.

Indexing student\_id and timestamp can speed common queries.

## 6.6 Server & Socket Events

**Socket events emitted by client:**

start\_camera — user wants to open camera (UI action).

capture\_frame — client sends a base64-encoded image/frame for recognition.

register\_images — payload: images + student metadata.

refresh\_records — request fresh attendance entries.

**Socket events emitted by server:**

recognition\_result — identity label(s), confidence(s), and optional image URLs.

registration\_success / registration\_failed

records\_data — JSON list of attendance rows.

## 6.7 Important Configuration & Tuning Parameters

detection\_confidence\_threshold — controls false positives/negatives of face detector.

embedding\_distance\_threshold — crucial to tune per embedding model.

min\_face\_pixel\_size — ignore small detections to reduce noise.

attendance\_debounce\_seconds — prevents duplicate marks in a short window.

port — server port (default 5000).

allowed\_hosts / network interface — set binding interface to local IP to allow phone access.

# 7. Working of the System

This section explains typical user flows and the internal steps taken from camera capture to attendance logging.

## 7.1 Student Registration (single registration flow)

**Open Registration UI:**

Client connects to server and navigates to "Register Student".

**Enter Student Metadata:**

Fill fields: Registration No, Name, Semester, Phone.

Capture Images:

Click “Start Camera” → camera feed displayed.

Capture multiple images (recommended 5–15). Client sends each captured frame via socket event register\_images (or batched).

**Server-side Handling:**

registration.py or server.py receives images.

For each image: detect face, crop, save to dataset/<reg\_no>/.

Optionally compute embedding and append to embeddings/<reg\_no>.npy or a global index.

Insert/Update students table with metadata and dataset path.

**Confirmation:**

Server emits registration\_success with summary.

Client shows success message.

## 7.2 Marking Attendance (recognition flow)

**Open Mark Attendance:**

Client navigates to "Mark Attendance" and starts camera.

**Capture Frame(s):**

User clicks “Capture & Recognize” or client streams periodic frames.

Frame(s) sent to server as base64-encoded images through capture\_frame.

**Server Processing:**

Run face detection on frame. **For each bounding box:**

Crop and resize the face region.

Compute embedding vector via faceEmbedding.py.

Compare embedding with stored embeddings in recognition module.

Determine the best candidate and distance.

If match and distance < threshold, mark as recognized.

**Attendance Logging:**

Insert a row into attendance table with timestamp and confidence.

If multiple faces are recognized in one frame, multiple attendance rows created.

Apply debounce logic to avoid duplicate inserts for the same student in same session.

**Feedback to Client:**

Emit recognition\_result event with detected names and confidence.

Client displays recognition result (e.g., “John Doe — Present — 0.32”).

## 7.3 Viewing Records

Client requests fresh records by refresh\_records.

Server queries the attendance table (optionally with filters: date range, student id).

Server emits records\_data containing rows to display in a table on UI.

UI supports CSV export (if implemented) or simple table view.

# 8. Database and Models

This section contains more concrete details about storage and pretrained models.

## 8.1 Models Provided

**Face detector:**

**Files:** deploy.prototxt and res10\_300x300\_ssd\_iter\_140000.caffemodel.

**Purpose:** locating faces quickly and reliably in a frame using OpenCV’s DNN module.

**Embedding model:** Either included or loaded from a known checkpoint (implementation-dependent).

Typical models: Facenet, MobileFaceNet, or TF/Keras-based embedding nets.

Expected output: vector (commonly 128- or 512-dimensional) that represents face features.

## 8.2 How embeddings are stored

**Two common strategies:**

**On-disk serialized per-student embeddings**

For every registered image, compute embedding and save as a .npy file in dataset/<reg\_no>/embeddings.npy or a central embeddings file with mapping to student IDs.

**Advantage**: fast load during recognition, simple file organization.

**Compute at runtime from saved images**

On server start, scan dataset/ folders and compute embeddings for all images into an in-memory dictionary {student\_id: [embeddings...]}.

**Advantage:** less risk of storing inconsistent data; recompute ensures consistent preproc.

**Disadvantage:** longer startup time.

The repo’s implementation tends to follow option (2) for simplicity, optionally caching embeddings.

## 8.3 Database best-practices for this project

Use parameterized SQL queries to avoid injection.

Wrap writes in transactions for integrity.

Periodically vacuum SQLite if the database will grow significantly.

If many concurrent clients are expected, consider migrating to PostgreSQL or MySQL.

# 9. User Interface

This section explains the web UI pages/components and how they map to functionality.

## 9.1 Layout & Navigation

**Header/Tabs:**

Register Student

Mark Attendance

View Records

**Register Student Page:**

Input fields for student metadata.

Camera view and capture button(s).

Thumbnails of captured images and a “Register” button.

**Mark Attendance Page:**

Camera view and “Capture & Recognize” button.

Live list of recent recognitions (name +Confidence + timestamp).

Option to toggle continuous recognition vs single-shot.

**View Records Page:**

Table with columns: Serial, Reg No, Name, Timestamp, Confidence, Image (thumbnail).

Buttons: Refresh, Export CSV (if implemented), Filter by date/student.

## 9.2 Client-side Implementation Notes

Camera capture: uses navigator.mediaDevices.getUserMedia({video: true}).

Frame encoding: canvas toDataURL('image/jpeg', quality) for smaller payloads.

Socket.IO: used for efficient low-latency transport. Events properly namespaced to avoid collisions.

**UX considerations:**

Show progress indicators while server processes frames.

Provide clear success/failure messages during registration.

Handle camera permission denial gracefully.

## 9.3 Security & Privacy in UI

Warn users that images are stored and used for recognition (consent).

For sensitive environments, require authentication before accessing registration or viewing records.

Provide an option to not save images (use recognition only) if privacy is a concern.

# 10. Testing and Results

This section summarizes recommended testing procedures, sample results, and how to evaluate performance.

## 10.1 Test Scenarios

**Registration Integrity:**

Register multiple students and verify that images are saved in their respective dataset/ folder.

Verify students table contains correct metadata.

**Recognition Accuracy:**

Test with controlled frontal images (same lighting) for baseline accuracy.

Test with varied angles, glasses/no-glasses, different lighting conditions.

Measure false positives (recognizing wrong student) and false negatives (failing to recognize).

**Concurrency & Latency:**

Single-shot recognition: measure end-to-end latency (client capture → server recognition → client result).

**Continuous recognition:** measure CPU & memory under periodic frame streams.

**Database correctness:** Verify timestamp accuracy and no duplicate entries for short intervals thanks to debounce logic.

## 10.2 Example Evaluation Metrics

**Accuracy:** fraction of correctly labeled recognized faces among attempted recognitions.

**Precision:** of cases labeled as present, how many were correct (helps assess false positives).

**Recall:** proportion of actual present students that were correctly recognized.

**Average latency:** typical roundtrip time in ms.

**Example run (illustrative):** Test set: 100 captured face instances from registered students.

**Correct recognitions:** 86 → Accuracy 86%

**Avg latency:** 450 ms (on a mid-range laptop)

**False match rate:** 3%

(Your actual numbers will vary based on embedding model, dataset size, hardware, and lighting.)

## 10.3 Performance Tips

Use smaller image quality / JPEG compression for faster network transfer.

Precompute and cache embeddings to avoid repeated computation.

If many students, organize embeddings into an index (Annoy, Faiss) for faster nearest-neighbor lookup.

Use GPU (if available) for embedding model inference to reduce latency.

# 11. Troubleshooting & Deployment Notes

This section gives practical notes for common problems and deployment tips.

## 11.1 Common Problems & Fixes

**Camera permission denied:**

Ensure site is served via http://localhost or https. Some browsers block getUserMedia on insecure origins.

Check browser console for errors and prompt user to allow camera.

Phone cannot connect to server: Confirm server bound to 0.0.0.0 or the host’s LAN IP and not just 127.0.0.1.

Disable local firewall or open the server port (5000 by default).

Ensure phone and server are on the same network.

**Slow recognition / high CPU usage:** Reduce frame size or frequency.

Pre-compute embeddings; use efficient nearest-neighbour library.

Offload embedding inference to GPU if available.

Poor recognition accuracy:

Add more varied samples per student.

Improve lighting and ensure frontal face captures.

Tune matching threshold and experiment with cosine vs Euclidean.

Database locked errors (SQLite):

Use short transactions; close connections promptly.

For heavy concurrent writes, consider using a client-server DB (Postgres/MySQL).

## 11.2 Deployment Advice

**Local LAN Deployment:**

Start server: python main/server.py (or flask run --host=0.0.0.0).

Access from phone: http://<server-ip>:5000.

Production / Internet-facing Deployment:

Use HTTPS (TLS) — browsers increasingly block camera capture on non-secure origins.

Put the application behind a reverse proxy (Nginx) and use Gunicorn/Uvicorn for WSGI/ASGI.

Add authentication & role-based access (admins for registration).

Consider moving SQLite → PostgreSQL for concurrency and reliability.

## 11.3 Backup & Maintenance

Regularly export attendance table to CSV for archival.

Backup dataset/ folder and attendance.db periodically.

Maintain a script to re-index embeddings after adding new students.

# 12. Conclusion

This documentation described the architecture, components, and operation of the Attendance System Using Facial Recognition. The project demonstrates a practical integration of face detection, embedding-based recognition, and a web interface for real-time attendance logging. It is suitable for laboratory use and as a learning project for students exploring applied computer vision and lightweight web systems.

## Key takeaways:

The repo is a working example bridging browser-based capture and server-side recognition.

Success depends on good-quality training images per student and sensible thresholding.

For larger scale or production use, consider stronger storage, secured transport (HTTPS), and faster nearest-neighbour indexing for embeddings.