# Geofence Tracking Main Service – Draft Submission

## 1. Project Metadata

Project / Module Name: Geo-fencing Attendance & Tracking System  
Developer Name: Sandeep Kandula  
Date & Version: 18-12-2025 v 1.0  
Approvers: Tech Lead, Architect, Product Owner

## 2. Problem Statement & Objectives

**Problem being solved**

Organizations that deploy field or mobile employees lack a reliable system to **restrict operations to predefined geographic areas**, **verify employee identity**, and **track work sessions in real time**.  
Manual attendance methods and unrestricted check-ins lead to location spoofing, proxy attendance, inaccurate travel reporting, and limited visibility into employee movement and task completion.

There is a need for a centralized backend system that can **enforce geofence-based restrictions**, **validate employee presence and identity**, and **maintain a complete audit trail of employee activity**.

**Business Objective**

The objective of this system is to build a **secure, automated geofence-enabled employee tracking platform** that ensures:

* Employees can **check in only when physically present inside their assigned operating area**
* Employee identity is **verified through facial recognition** before starting a work session
* Work sessions are **tracked from check-in to check-out**, including live location updates
* Administrators can **monitor live activity**, review **past sessions**, and analyze movement and compliance data
* Travel distance is **captured and validated using odometer uploads**

This improves **operational discipline, accountability, and data accuracy** for field operations.

**Success Criteria / Acceptance Criteria**

The solution will be considered successful if:

* Employees are **prevented from checking in outside the configured geofence**
* Face verification is **mandatory and successfully validated** before session start
* Each employee session correctly records:
  + Check-in and check-out timestamps
  + Live GPS movement
  + Geofence completion status
  + Odometer readings
* Admin users can:
  + View live employee locations
  + Access historical sessions and travel paths
  + Generate daily and weekly summaries
* The system operates **reliably under continuous location updates** with minimal latency

**Out of Scope**

* Mobile application or frontend UI development
* Advanced route optimization or predictive analytics
* Payroll, billing, or incentive calculations
* Offline GPS tracking without network connectivity
* Manual identity verification without face recognition

## 3. Proposed Solution

**Overview (1–2 paragraphs)**

The proposed solution is a **backend-driven geofence-enabled employee tracking system** designed to control where employees can operate, authenticate their identity, and record their work activity in a structured and auditable manner.

The system allows administrators to define **geographic operating areas (geofences)** and assign them to employees. An employee can initiate a work session **only if they are physically located within the permitted geofence** and **successfully verified through an external face recognition microservice**. Once a session is started, the system continuously processes live GPS updates, records movement paths, detects geofence entry events, and maintains a complete session history. Additional validation such as **odometer image uploads and OCR-based mileage extraction** is used to ensure accurate travel reporting and prevent misuse.

The solution is implemented as a **main tracking service** that manages business logic and data persistence, while delegating compute-intensive facial verification to a **separate face recognition microservice**, ensuring scalability and fault isolation.

**Alternatives Considered (Brief)**

1. **Hexagonal Geofencing using Uber H3 + Redis**

An alternative approach considered was converting geographic regions (for example, areas derived from pincodes) into **H3 hexagonal indexes**, hashing latitude–longitude coordinates into hex cells, and storing valid cell IDs in **Redis** for constant-time (O(1)) location validation.

While this approach offers **extremely fast lookups** and is well-suited for **very large-scale, high-frequency location checks**, it introduces additional complexity in:

* + Precomputing and managing large sets of hex indexes
  + Handling boundary precision and resolution tuning
  + Operating and maintaining Redis infrastructure

Given the current scale and requirement for clear spatial accuracy, this approach was deferred.

The chosen approach uses **circular geofences** defined by a center latitude, longitude, and radius, with **Haversine distance calculations** to determine whether an employee is inside or outside a permitted area.

This approach was selected because it:

* Provides **clear geographic accuracy** and easy visualization
* Is **simple to configure and reason about** for administrators
* Avoids additional infrastructure dependencies such as Redis
* Performs efficiently at the current scale of GPS updates
* Integrates cleanly with session management and reporting logic

Identity verification is handled via a **dedicated face recognition microservice**, ensuring secure authentication without coupling heavy ML workloads into the main service. Odometer validation using image uploads and OCR further strengthens data integrity by validating travel distance.

Overall, this architecture balances **correctness, maintainability, and scalability**, while leaving room for future optimization (such as H3-based geofencing) if system scale increases.

4. Technical Specifications

**Programming Language / Framework**

* **Backend Language:** Python 3.x
* **Web Framework:** FastAPI
* **ORM:** SQLAlchemy
* **API Style:** RESTful APIs (JSON + multipart/form-data)
* **Distance Calculation:** Haversine formula for geofence validation
* **Architecture Style:** Modular monolith with external microservice integration (Face Recognition)

**Database Changes (if any)**

The system uses **Supabase PostgreSQL** as the primary database. The following core tables are implemented to support tracking, geofencing, and reporting:

* **admins** – Admin user records
* **employees** – Employee master data
* **employee\_faces** – Face embeddings and reference image URLs (linked via employee\_id)
* **user\_sessions** – Employee work sessions (check-in to check-out)
* **user\_locations** – Time-series GPS data for polylines and live tracking
* **geofences** – Circular geofence definitions
* **geofence\_status** – Per-session geofence completion tracking
* **geofence\_assignments** – Daily geofence targets assigned to employees
* **daily\_summary** – Aggregated daily reports (distance, geofences, locations)

No destructive schema migrations are performed at runtime. Tables are created safely using checkfirst=True to avoid modifying existing data.

**APIs to Use / Expose**

**Employee & Admin Management**

* POST /admin/create – Create admin user
* POST /employee/create – Create employee
* GET /admin/employees – List employees
* PUT /admin/employee/{employee\_id}/home – Set employee home geofence

**Face Enrollment (via Microservice Proxy)**

* POST /admin/employee/{employee\_id}/enroll-face  
  → Proxies request to Face Recognition Microservice

**Session Lifecycle**

* POST /session/start – Employee check-in (face verify + geofence + odometer)
* POST /session/checkout – Employee check-out with odometer OCR
* GET /admin/employee/{employee\_id}/sessions – View past sessions

**Tracking & Geofencing**

* POST /tracking/update-location – Live GPS updates
* GET /tracking/polyline/{session\_id} – Session route polyline
* POST /geofence/create – Create geofence
* DELETE /geofence/{geofence\_id} – Delete geofence
* POST /admin/assign-geofence – Assign geofence to employee

**Reports**

* GET /summary/{employee\_id} – Employee daily summary
* GET /admin/summary/today
* GET /admin/summary/weekly
* GET /admin/live-location/{session\_id}

**Dependencies (Internal / External)**

**Internal Dependencies**

* Face Recognition Microservice (HTTP-based integration)
* Shared Supabase PostgreSQL database
* Shared employee identifier (employee\_id)

**External Dependencies**

* **Supabase**
  + PostgreSQL (data persistence)
  + Storage buckets (selfies, odometers, face images)
* **Gemini API**
  + OCR for odometer image mileage extraction
* **Python Libraries**
  + fastapi
  + sqlalchemy
  + requests
  + haversine
  + python-dotenv
  + pydantic
* **Infrastructure**
  + REST-based inter-service communication
  + CPU-only compatible deployment

## 5. Architecture (Mermaid)

A diagram of a software company

AI-generated content may be incorrect.  
```

A diagram of a diagram

AI-generated content may be incorrect.

A diagram of a computer process

AI-generated content may be incorrect.

## 6. Requirements

**Functional Requirements (FR)**

* **FR-1:** The system shall allow administrators to create and manage employee records.
* **FR-2:** The system shall allow administrators to define circular geofences using latitude, longitude, and radius.
* **FR-3:** The system shall allow administrators to assign geofences to employees for a given day.
* **FR-4:** The system shall allow administrators to enroll an employee’s face image using the face recognition microservice.
* **FR-5:** The system shall restrict employee check-in to locations that fall inside the assigned geofence.
* **FR-6:** The system shall verify employee identity using face recognition before starting a session.
* **FR-7:** The system shall create a work session upon successful check-in and record start time and location.
* **FR-8:** The system shall continuously accept live GPS location updates from employees during an active session.
* **FR-9:** The system shall detect and record geofence entry completion during a session.
* **FR-10:** The system shall log GPS polylines at configured time intervals for route reconstruction.
* **FR-11:** The system shall allow employees to upload odometer images at check-in and check-out.
* **FR-12:** The system shall extract odometer mileage from images using OCR and compute total distance traveled.
* **FR-13:** The system shall allow employees to check out and close an active session.
* **FR-14:** The system shall generate daily summaries including distance traveled and geofence completion.
* **FR-15:** The system shall allow administrators to view live employee locations and historical session data.

**Non-Functional Requirements (NFR)**

**Performance Target**

* The system shall process **live location updates every 10 seconds per active session** without data loss.
* Face verification during check-in shall complete within **acceptable latency (≤ 2 seconds under normal load)**.
* GPS polyline storage shall be optimized to avoid excessive database writes.

**Security Considerations**

* Employee check-in shall be permitted only after successful face verification.
* Face recognition functionality shall be isolated in a separate microservice.
* Direct access to the face recognition service shall be restricted to the main service.
* All uploaded images (selfies, face images, odometer images) shall be stored securely in object storage.
* Sensitive credentials shall be managed using environment variables and secrets.

**Scalability / Reliability**

* The system shall support multiple concurrent employee sessions.
* Database connections shall use pooling and automatic reconnection.
* Failure of the face recognition service shall not corrupt main service data.
* The architecture shall allow independent scaling of the face recognition microservice.
* The system shall maintain session and tracking data integrity in case of partial failures.

## 7. Testing Plan

**7. Testing & Validation Plan**

**Unit Tests: Planned Coverage**

Formal unit test automation is **minimal in the current phase**. Core logic was validated through **manual and controlled test executions**, focusing on correctness rather than test volume.

The following components were manually verified:

* Geofence distance calculation using the Haversine formula
* Session creation and state transitions (check-in, active, check-out)
* GPS location logging and polyline generation
* Odometer value extraction from uploaded images
* API request validation and error handling

This approach was chosen to validate system behavior end-to-end during initial development.

**Integration Tests: Main Flows**

The system was tested using a **single complete employee session** to validate integration across all components:

* Admin creates an employee
* Admin enrolls the employee’s face image via the face recognition microservice
* Employee performs check-in from within the assigned geofence
* Face verification is successfully validated
* Odometer image is uploaded and mileage extracted
* Live GPS location updates are sent during the session
* Geofence entry is detected and recorded
* Employee checks out and session is closed
* Daily summary is generated and viewable by admin

This test confirmed correct interaction between:

* Main Tracking Service
* Face Recognition Microservice
* Database and object storage
* External OCR service

**Acceptance Scenarios: Business Sign-off Criteria**

The solution is considered acceptable for sign-off if the following conditions are met:

* An employee cannot check in outside the assigned geofence
* Face verification is mandatory and correctly validates employee identity
* A complete session lifecycle (check-in → tracking → check-out) is recorded without errors
* Admin users can view session history and tracking data
* Distance traveled and geofence completion are accurately reflected in reports

Successful execution of the single end-to-end session test satisfies the current acceptance criteria.

## 8. Open Questions

Auth & RBAC integration pending

H3 logic Pending

ROUTES EXPLANATION

**1️⃣ Health Check**

**GET /**

**Purpose**  
Verify that the main tracking service is running.

**Input**  
None

**Behavior**  
Returns a static response indicating the backend is active.

**Output**

{

"message": "Tracking backend running (Day 1 complete)"

}

**2️⃣ Admin Management**

**POST /admin/create**

**Purpose**  
Create a new administrator account.

**Input (JSON)**

{

"name": "Admin Name",

"email": "admin@example.com",

"password": "optional"

}

**Behavior**

* Stores admin details in the admins table
* Password hashing/authentication is planned for future phases

**Output**

{

"message": "Admin created",

"admin\_id": 1

}

**GET /admin/employees**

**Purpose**  
Retrieve all registered employees for admin view.

**Input**  
None

**Behavior**  
Fetches employee records sorted by creation time.

**Output**

[

{

"id": 10,

"name": "John Doe",

"employee\_code": "EMP123",

"is\_active": true,

"created\_at": "2025-01-01T10:00:00"

}

]

**3️⃣ Employee Management**

**POST /employee/create**

**Purpose**  
Create a new employee record.

**Input (JSON)**

{

"name": "John Doe",

"employee\_code": "EMP123"

}

**Behavior**

* Stores employee data
* Employee must exist before face enrollment or session start

**Output**

{

"message": "Employee created",

"employee\_id": 10

}

**PUT /admin/employee/{employee\_id}/home**

**Purpose**  
Set the employee’s allowed operating (home) geofence.

**Input (JSON)**

{

"home\_lat": 17.385,

"home\_lng": 78.486,

"home\_radius\_m": 500

}

**Behavior**

* Saves geofence center and radius
* Used to restrict employee check-in location

**Output**

{

"message": "Employee home updated",

"employee\_id": 10

}

**4️⃣ Face Enrollment (Admin → Main → Face Service)**

**POST /admin/employee/{employee\_id}/enroll-face**

**Purpose**  
Enroll an employee’s face image.

**Input (multipart/form-data)**

* file → face image  
  **OR**
* image\_url → URL of face image

**Behavior**

* Validates employee existence
* Forwards the image to Face Recognition Microservice
* Stores returned reference image URL

**Output**

{

"message": "Employee face enrolled successfully",

"employee\_id": 10,

"image\_url": "https://storage/face/abc.jpg"

}

**5️⃣ Session Lifecycle**

**POST /session/start**

**Purpose**  
Employee check-in and session creation.

**Input (multipart/form-data)**

* data (JSON string):

{

"employee\_id": 10,

"lat": 17.385,

"lng": 78.486

}

* selfie → live selfie image
* odometer → odometer image

**Behavior**

1. Validates employee
2. Checks geofence using Haversine distance
3. Verifies face via Face Microservice
4. Uploads images to storage
5. Extracts odometer value using OCR
6. Creates session entry

**Output**

{

"message": "Session started",

"session\_id": 101,

"employee\_id": 10,

"odometer\_value": 12500.5

}

**POST /session/checkout**

**Purpose**  
Close an active session.

**Input (multipart/form-data)**

* session\_id: 101
* odometer → final odometer image

**Behavior**

* Uploads final odometer image
* Extracts mileage
* Calculates distance traveled
* Marks session as completed
* Creates daily summary

**Output**

{

"message": "Session checked out successfully",

"session\_id": 101,

"odometer\_distance\_km": 12.4,

"geofence\_count": 3

}

**6️⃣ Live Tracking & Geofencing**

**POST /tracking/update-location**

**Purpose**  
Process live GPS updates during an active session.

**Input (JSON)**

{

"session\_id": 101,

"lat": 17.386,

"lng": 78.487

}

**Behavior**

* Validates session
* Updates last known location
* Logs polyline every 60 seconds
* Checks geofence entry using Haversine logic
* Marks geofence completion if entered

**Output**

{

"message": "Location processed",

"session\_id": 101,

"polyline\_logged": true,

"geofences\_completed": 1

}

**GET /tracking/polyline/{session\_id}**

**Purpose**  
Retrieve GPS route of a session.

**Input**  
Path parameter: session\_id

**Behavior**  
Returns ordered GPS points.

**Output**

[

{

"lat": 17.385,

"lng": 78.486,

"timestamp": "2025-01-01T10:05:00"

}

]

**🔷 FACE RECOGNITION MICROSERVICE – API ROUTES**

**1️⃣ Enroll Face**

**POST /face/enroll/{employee\_id}**

**Purpose**  
Enroll an employee’s facial data.

**Input**

* employee\_id (path)
* file OR image\_url

**Behavior**

* Detects face
* Generates embeddings using InsightFace
* Stores embedding in database
* Uploads reference image
* Updates FAISS index

**Output**

{

"message": "Employee face enrolled successfully",

"employee\_id": 10,

"image\_url": "https://storage/face/abc.jpg"

}

**2️⃣ Verify Face**

**POST /face/verify**

**Purpose**  
Verify a live selfie against enrolled faces.

**Input (multipart/form-data)**

* file → selfie image

**Behavior**

* Extracts face embedding
* Searches FAISS index
* Matches against stored embeddings

**Output (Match Found)**

{

"match": true,

"employee\_id": 10

}

**Output (No Match)**

{

"match": false,

"confidence": null

}

**3️⃣ Delete Face**

**DELETE /face/delete/{employee\_id}**

**Purpose**  
Delete employee face data.

**Input**

* employee\_id (path)

**Behavior**

* Removes face embedding
* Deletes stored image
* Rebuilds FAISS index

**Output**

{

"message": "Employee face deleted successfully",

"employee\_id": 10

}

**📦 ORM MODELS EXPLANATION (models.py)**

These models define the **database structure** using **SQLAlchemy ORM**.  
Each class maps directly to a PostgreSQL table.

**1️⃣ Admin**

class Admin(Base):

**Purpose**

Stores administrator accounts who manage employees, geofences, and reports.

**Key Fields**

* id → Primary key
* name → Admin name
* email → Unique identifier for admin
* password\_hash → Placeholder for future authentication
* created\_at → Auto timestamp

**Used In**

* Admin creation
* Admin dashboard access (future)

**2️⃣ Employee**

class Employee(Base):

**Purpose**

Represents a field employee who performs check-ins, tracking, and sessions.

**Key Fields**

* id → Primary key
* name → Employee name
* employee\_code → Unique code used for login/check-in
* is\_active → Soft enable/disable flag
* home\_lat, home\_lng, home\_radius\_m → Home/operating geofence
* created\_at → Creation timestamp

**Used In**

* Session creation
* Face enrollment
* Geofence restriction
* Reports

**3️⃣ EmployeeFace**

class EmployeeFace(Base):

**Purpose**

Stores facial embeddings used for identity verification.

**Key Fields**

* employee\_id → One-to-one mapping with Employee
* embedding → 512-dimension face embedding (array of floats)
* reference\_image\_url → Stored face image
* created\_at, updated\_at

**Used In**

* Face enrollment
* Face verification
* FAISS index rebuilding

⚠️ **Important:**  
This table is shared logically with the **Face Microservice**, even though it resides in the same database.

**4️⃣ CheckinOTP *(Future fallback)***

class CheckinOTP(Base):

**Purpose**

Fallback authentication method if face verification is unavailable.

**Key Fields**

* employee\_id
* otp\_code
* valid\_for\_date
* is\_used

**Used In**

* Currently unused
* Reserved for future OTP-based login

**5️⃣ UserSession**

class UserSession(Base):

**Purpose**

Represents **one complete work session** (check-in → check-out).

**Key Fields**

* employee\_id
* check\_in\_time, check\_out\_time
* check\_in\_selfie\_url
* odometer\_start\_image\_url, odometer\_end\_image\_url
* odometer\_start\_value, odometer\_end\_value
* start\_lat, start\_lng, end\_lat, end\_lng

**Used In**

* Session lifecycle
* Distance calculation
* Daily summaries
* Admin reports

**6️⃣ UserLocation**

class UserLocation(Base):

**Purpose**

Stores **time-series GPS points** for route tracking (polyline).

**Key Fields**

* employee\_id
* session\_id
* lat, lng
* timestamp

**Used In**

* Live tracking
* Route reconstruction
* Geofence entry logging

**7️⃣ Geofence**

class Geofence(Base):

**Purpose**

Defines a **circular geofence**.

**Key Fields**

* name
* center\_lat, center\_lng
* radius\_m
* created\_by

**Used In**

* Geofence creation
* Geofence assignment
* Live location checks

**8️⃣ GeofenceStatus**

class GeofenceStatus(Base):

**Purpose**

Tracks whether a geofence was **completed during a session**.

**Key Fields**

* employee\_id
* session\_id
* geofence\_id
* completed
* completed\_at

**Used In**

* Prevent duplicate geofence completion
* Daily reports

**9️⃣ GeofenceAssignment**

class GeofenceAssignment(Base):

**Purpose**

Assigns geofences to employees **for a specific date**.

**Key Fields**

* employee\_id
* geofence\_id
* assigned\_date
* assigned\_by

**Used In**

* Daily targets
* Employee task planning

**🔟 DailySummary**

class DailySummary(Base):

**Purpose**

Stores **aggregated daily results** per employee.

**Key Fields**

* employee\_id
* date
* total\_distance
* geofence\_count
* start\_lat, start\_lng, end\_lat, end\_lng

**Used In**

* Admin reports
* Performance tracking

**📑 SCHEMAS EXPLANATION (schemas.py)**

Schemas define **request/response validation** using **Pydantic**.  
They **do NOT create tables** — they only validate API inputs.

**Admin Schemas**

**AdminCreate**

class AdminCreate(BaseModel):

**Used In:** POST /admin/create

**Fields**

* name
* email
* password (optional)

**Employee Schemas**

**EmployeeCreate**

**Used In:** POST /employee/create

{

"name": "John",

"employee\_code": "EMP001"

}

**EmployeeHomeUpdate**

**Used In:**  
PUT /admin/employee/{id}/home

Defines home/operating geofence.

**Session Schemas**

**SessionStart**

**Used In:**  
POST /session/start

Contains:

* lat, lng
* employee\_id OR employee\_code

**LocationUpdate**

**Used In:**  
POST /tracking/update-location

Contains:

* session\_id
* lat, lng

Handles string → number coercion safely.

**Geofence Schemas**

**GeofenceCreate**

**Used In:**  
POST /geofence/create

Defines circular geofence.

**GeofenceAssignmentCreate**

**Used In:**  
POST /admin/assign-geofence

Assigns geofence for a given date.

**OTP Schema (Future)**

**OTPCreate**

Reserved for OTP-based authentication fallback.

**🧠 Final Notes for the New Developer**

* **models.py = database truth**
* **schemas.py = API contract**
* Face logic lives in **Face Microservice**, not here
* Geofence logic is **Haversine-based**, not H3
* Session is the **core entity** — everything revolves around it

PROBLEMS:

* + Can only delete the employees if we enroll the face image(can be solved by fixing the main service routes)
  + Face Service is not deployable(If possible use another face service there are plenty)
  + All the code is made with pydantic so be careful with the data types you will get many errors

IMP POINTS TO BE NOTED

CREATE POLICY "Allow public uploads" ON storage.objects

FOR INSERT

WITH CHECK (bucket\_id = 'selfies') -- or 'odometers' for the other bucket

CREATE POLICY "Allow public uploads to odometers"

ON storage.objects

FOR INSERT

WITH CHECK (

bucket\_id = 'odometers'

);

CREATE POLICY "Allow public uploads to face"

ON storage.objects

FOR INSERT

WITH CHECK (

bucket\_id = 'face'

);

you dont have to create a policy for buckets if you use service role key(which is a secret) which bypass the Role based access

http requests wait for the response until it gets a response you can see anything in the terminal(so if a microservice is slow then the main service request is not shown in the terminal)

github\_link\_for\_main\_service - <https://github.com/Sandeepkandula2004/Geofence_Cerevyn>

github\_link\_for\_Face\_micro\_service - <https://github.com/Sandeepkandula2004/Face_For_Cerevyn>