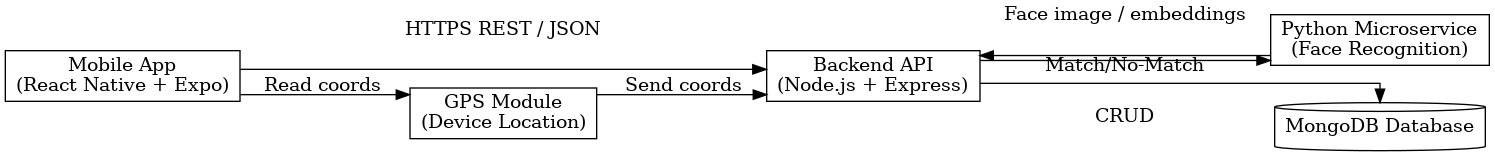
# CHAPTER 3: METHODOLOGY

## 3.1 Introduction

This chapter presents the methodology adopted for the design and development of ClockMate, a hybrid mobile attendance management system that integrates modern web technologies and advanced computer vision to enable secure, reliable, and efficient employee time tracking. The chapter details the system architecture, component design, stakeholder analysis, requirements elicitation and specification, Unified Modeling Language (UML) diagrams, algorithms with supporting flowcharts, development methodology, software process model, and design considerations for the user interface and database. The implementation integrates React Native with Expo for the cross‑platform mobile frontend, Node.js with Express for the backend API, MongoDB with Mongoose for data persistence, and a Python microservice for facial recognition using the face\_recognition library.

## 3.2 Architecture of the Proposed System

ClockMate follows a modular, service‑oriented architecture. The mobile application communicates securely with a RESTful backend API. Facial recognition workloads are offloaded to a Python microservice to leverage optimized computer vision libraries. A MongoDB datastore persists user profiles, face encodings, and attendance events. The device GPS provides location coordinates to enforce geofencing policies before attendance is recorded.



*Figure 3.1: High-Level Architecture of ClockMate*

## 3.3 Component Designs and Descriptions

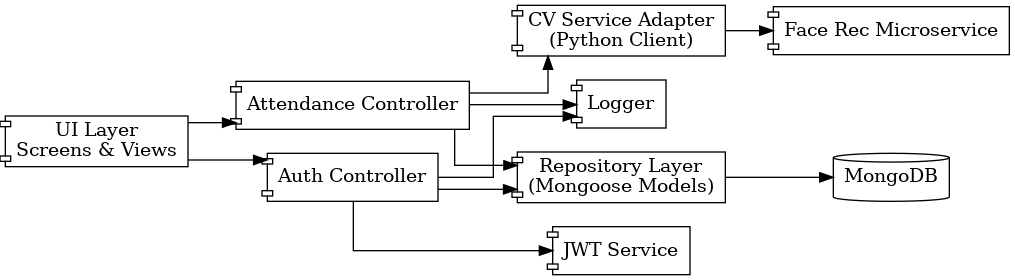
The following subsections describe how each component functions within the architecture.

### 3.3.1 Mobile Application (React Native + Expo)

The mobile app implements the primary user experience for employees and administrators. It provides a face capture interface, punch in/out controls, an attendance dashboard, and profile management. It acquires GPS coordinates from the device, packages them with the captured image and a JWT, and submits them to the backend over HTTPS. The camera flow includes real‑time guidance (bounding box, illumination checks) to improve capture quality and reduce false negatives.

### 3.3.2 Backend API (Node.js/Express)

The backend exposes REST endpoints for authentication, attendance logging, reporting, and administration. It performs JWT verification, validates geofences, orchestrates calls to the Python service for recognition, and persists results via Mongoose models. Business rules (e.g., preventing duplicate punch‑ins without a punch‑out) are enforced in controller logic, keeping the API stateless.



*Figure 3.2: Component Interaction and Responsibilities*

### 3.3.3 Python Microservice (Face Recognition)

The Python service encapsulates compute‑intensive vision tasks. It detects faces (HOG/CNN), encodes faces into 128‑D embeddings, and performs similarity matching against stored encodings. A configurable threshold determines acceptance, balancing false acceptance and rejection rates. The service returns the best match (if any) with a confidence score for the API to finalize the decision.

### 3.3.4 Database Layer (MongoDB + Mongoose)

MongoDB stores denormalized documents for performance and scalability. Collections include Users, FaceEncodings, Roles, and Attendance. Mongoose schemas enforce structure, indexes optimize queries (e.g., userId+timestamp), and TTL/indexing policies can be used for log retention and archiving.

### 3.3.5 GPS Module and Geofencing

The GPS module reads device latitude/longitude and the backend verifies membership within an allowed radius for the employee’s site. Distances are computed using the Haversine formula against the site center coordinates, with tolerance for GPS accuracy variations.

## 3.4 The Proposed System Main Functionalities (Functional Requirements)

The system provides: (i) biometric face‑based authentication; (ii) punch in/out with geofencing; (iii) attendance history and reporting; (iv) role‑based administration; and (v) secure session and token lifecycle management.

## 3.5 Identification of Stakeholders

Primary stakeholders: Employees (attendance marking), Administrators (oversight and configuration), and HR Managers (reporting and payroll inputs). Secondary stakeholders: IT Support/DevOps (deployment and monitoring) and Executive Management (policy and compliance).

## 3.6 Requirement Gathering Process

Requirements were elicited through semi‑structured interviews with HR managers and administrators, observation of current attendance workflows, and review of comparable systems. Findings were transcribed, categorized, and converted into user stories and acceptance criteria prioritized in a product backlog.

## 3.7 Requirement Specification

### 3.7.1 Functional User Requirements

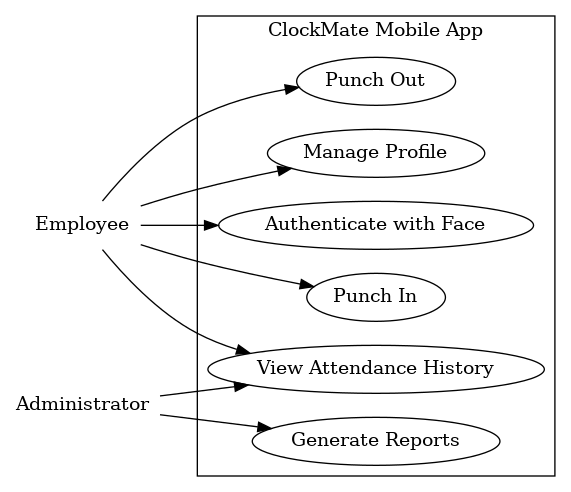
FUR‑1: User can authenticate using face biometric.  
FUR‑2: User can punch in and punch out.  
FUR‑3: User can view personal attendance history.  
FUR‑4: Admin can generate date‑range reports.  
FUR‑5: Admin can manage users and roles.

### 3.7.2 Functional System Requirements

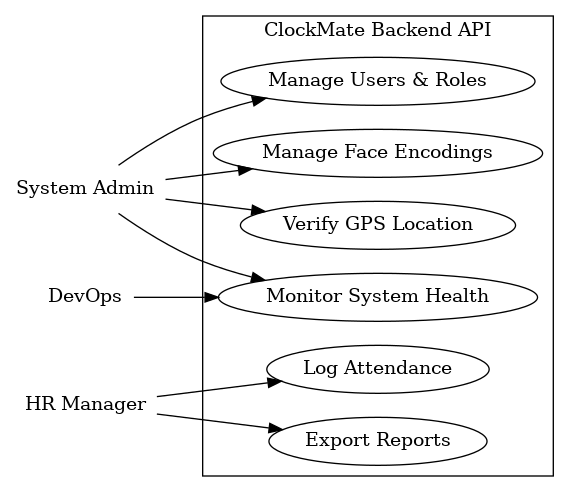
FSR‑1: System must detect and recognize faces above a configurable threshold.  
FSR‑2: System must verify GPS location within the geofence before logging attendance.  
FSR‑3: System must issue and verify JWTs for all protected endpoints.  
FSR‑4: System must prevent inconsistent events (e.g., duplicate consecutive IN events).  
FSR‑5: System must support exporting reports in CSV/PDF.

### 3.7.3 UML Diagrams

Use Case Diagrams (Frontend and Backend) are provided below.

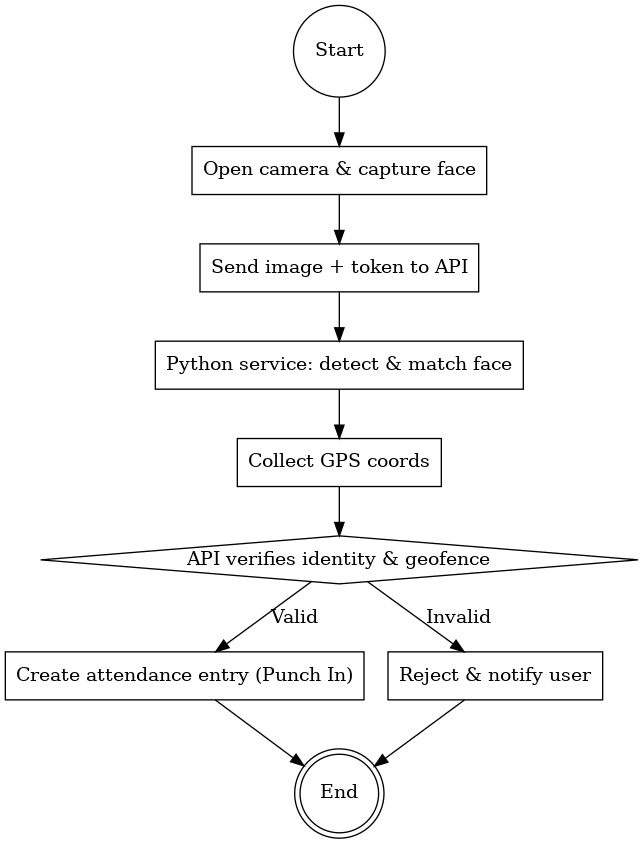


*Figure 3.3: Use Case Diagram – Frontend Models*

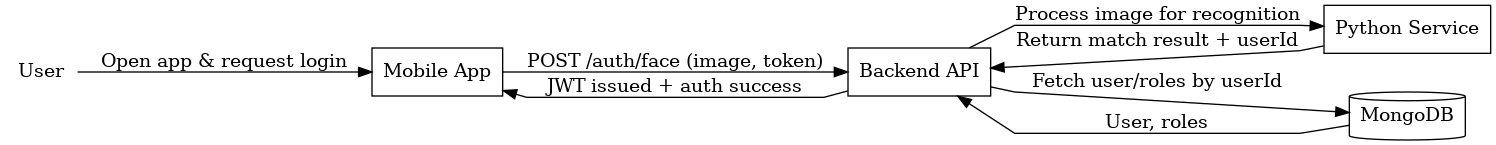


*Figure 3.4: Use Case Diagram – Backend Models*

Sequence and Activity Diagrams are provided to explain runtime behavior.



*Figure 3.5: Activity Diagram – Punch In Workflow*



*Figure 3.6: Sequence Diagram – Face Authentication*

### Use Case Descriptions

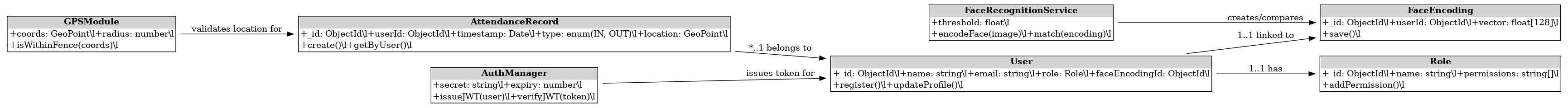
Actor: Employee — Performs authentication and attendance actions.  
Use Case: Authenticate with Face — The user opens the app, captures a face image, and receives a session token upon successful verification.  
Use Case: Punch In/Out — After authentication and geofence validation, the system records an IN or OUT event.  
Use Case: View Attendance History — The user retrieves a paginated list of personal attendance records.  
Actor: Administrator — Generates reports and manages users/roles.  
Use Case: Generate Reports — Admin selects a date range and exports attendance aggregates.  
Use Case: Manage Users & Roles — Admin creates, updates, or disables accounts and assigns permissions.

## 3.8 Non‑Functional Requirements (with Justifications)

Performance: Face verification end‑to‑end within ~2 seconds to sustain smooth UX.  
Security: HTTPS/TLS for transport, secure storage of encodings, RBAC with least privilege.  
Reliability: Graceful handling of network variability and retries; centralized logging.  
Scalability: Horizontal scaling of API and Python service with container orchestration.  
Usability: Clear feedback during capture and concise error messaging; accessible UI.  
Maintainability: Module boundaries between UI, API, CV service, and data access.  
Portability: Cross‑platform app via React Native (iOS/Android).

## 3.9 Candidate Classes and UML Class Diagram

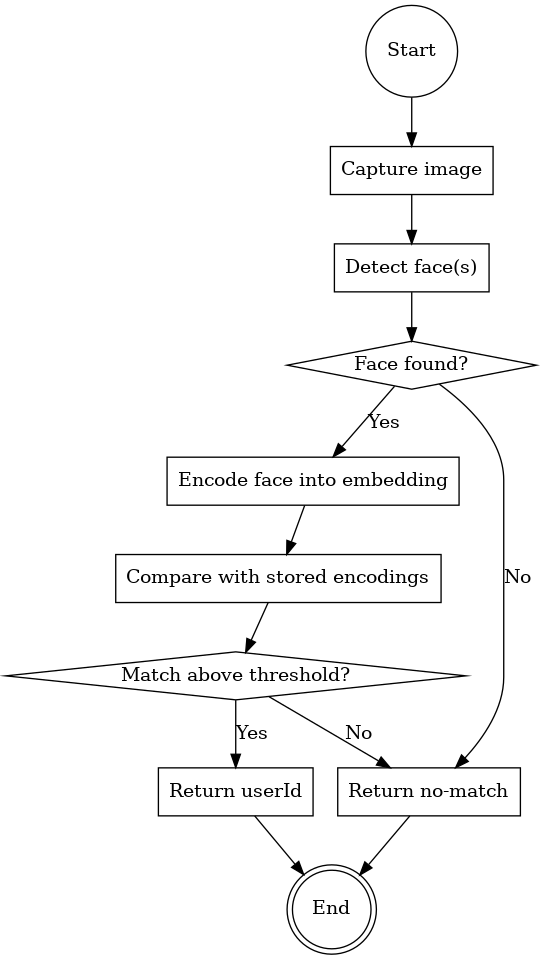
The following candidate classes structure the domain and services: User, Role, FaceEncoding, AttendanceRecord, AuthManager, GPSModule, and FaceRecognitionService. Their attributes, methods, and relationships are depicted below.



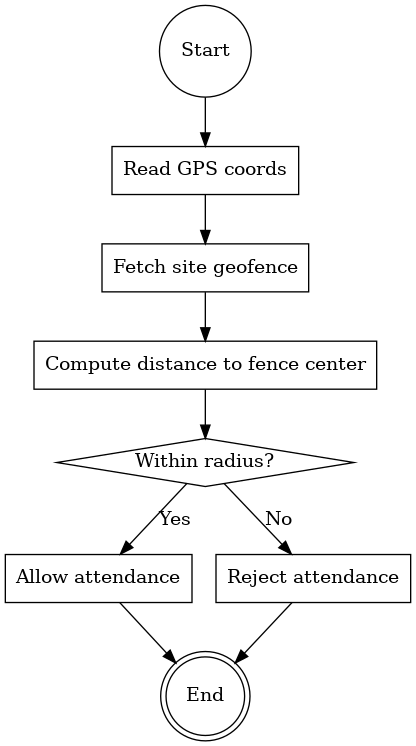
*Figure 3.7: UML Class Diagram with Attributes, Methods, and Relationships*

## 3.10 Algorithms and Flowcharts

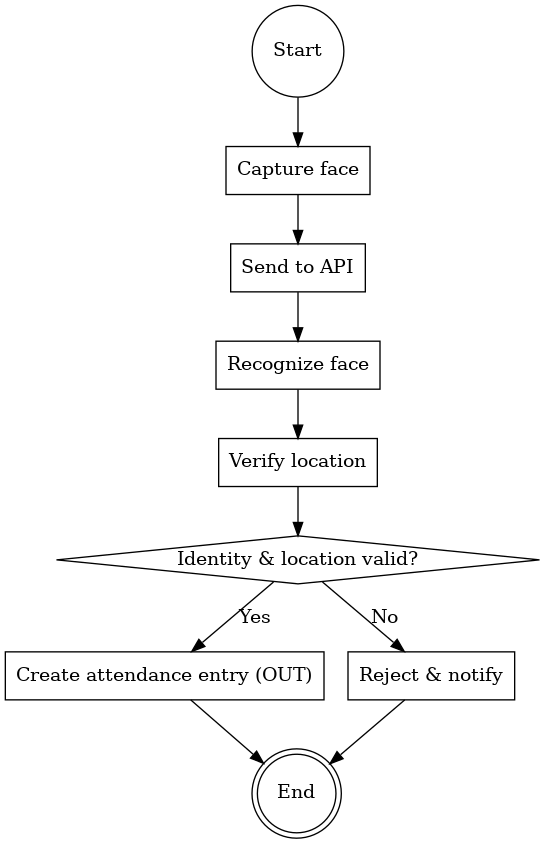
Algorithm 1 – Face Recognition: Detect faces, encode into embeddings, and compare with stored vectors to identify a subject.  
Algorithm 2 – Location Verification: Compute great‑circle distance to the site center and evaluate radius.  
Algorithm 3 – Punch Out: Mirror of Punch In with state validation to prevent inconsistent sequences.



*Figure 3.8: Flowchart – Face Recognition*



*Figure 3.9: Flowchart – Location Verification*



*Figure 3.10: Flowchart – Punch Out*

## 3.11 Project Methods Employed

An Agile (Scrum) approach is adopted to iteratively refine requirements and deliver increments. Two‑week sprints, daily stand‑ups, sprint reviews, and retrospectives support continuous improvement. Definition of Done includes unit tests, integration tests, and updated documentation.

## 3.12 Software Process Model and Justification

Scrum is selected due to the evolving nature of biometric thresholds, UX iteration for capture quality, and the need to incrementally validate performance at scale. This model accommodates stakeholder feedback and rapid prototyping of the Python microservice without blocking core app development.

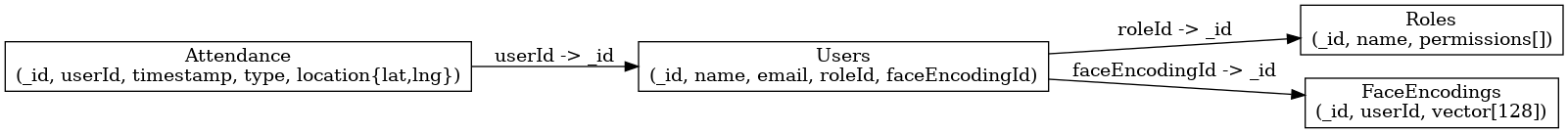
## 3.13 Project Design Considerations (Logical Designs)

### 3.13.1 UI Design (Wireframes)

Key wireframes include: (a) Face Authentication Screen with camera preview and capture guidance; (b) Attendance Dashboard displaying current state (IN/OUT) and quick actions; (c) History Screen with filters and pagination; and (d) Admin Panel for report generation. (Wireframes can be created in Figma or Lucidchart and inserted here.)

### 3.13.2 Database Design (E‑R Diagram and Schema)

The E‑R model below captures the core collections and references suitable for MongoDB. Schemas enforce required fields, indexes, and validation rules.



*Figure 3.11: E‑R Diagram for MongoDB Collections*

## 3.14 Developmental Tools

React Native (Expo) for cross‑platform UI; Node.js and Express for RESTful APIs; Python for computer vision; MongoDB with Mongoose for persistence; Postman for API testing; Docker for containerization; Git/GitHub for version control; and Lucidchart/Figma for diagrams and wireframes. The methodology section details how each tool integrates into the workflow, including local development via Expo and Dockerized services for parity with production.