May 26, 2025

UNIVERSITY OF BUEA
Buea, South West Region
Cameroon
P.O. Box 63,
Tel: (237) 3332 21 34/3332 26 90
Fax: (237) 3332 22 72

REPUBLIC OF CAMEROON

PEACE-WORK-FATHERLAND

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

NAME	MATRICULE
BILLA SOPHIA	FE22A176
EKANE METUGE AKAME FAVOUR	FE22A199
EYONG GODWILL NGANG	FE22A214
NEBOTA ISMAEL OWAMBA	FE22A256
ONYA MARTHA . O	FE22A292

Task 4

Course Master:
Dr. Nkemeni Valery
2024/2025

System Modelling and Design Report

Mobile-Based Attendance Management System using Geofencing and Facial Recognition

Group: 24
CEF440 - Internet Programming and Mobile Programming
Version: 1.0

CONTENTS

1. Introduction

This report presents the system modelling and design for **AuraCheck**, a Mobile-Based Attendance Management System developed for the University of Buea. The primary goal of AuraCheck is to automate and enhance the process of student attendance tracking by leveraging modern technologies such as geofencing and facial recognition.

System modelling is a crucial phase in the software development lifecycle. It involves creating abstract representations of a system to understand, analyze, communicate, and document its various aspects. This process helps in identifying requirements, visualizing the system's structure and behavior, and ensuring that all stakeholders have a common understanding before development commences.

This document details various Unified Modeling Language (UML) diagrams and other modelling tools used to represent the AuraCheck system. These include:

- Context Diagram
- Dataflow Diagram (Level 1)
- Use Case Diagram
- Sequence Diagram (Illustrating the Check-in Process)
- Class Diagram
- Deployment Diagram

Each section will describe the purpose of the respective diagram, present its specification as it applies to the AuraCheck system, provide a placeholder for the diagram image, and offer a brief explanation of its key components.

2. Context Diagram

2.1 Purpose of a Context Diagram

A Context Diagram (also known as a Level 0 Dataflow Diagram) is the highest-level view of a system. It defines the system boundary, identifies the external entities (terminators) that interact with the system, and shows the major data flows between these entities and the system. It provides a concise overview of the system's scope without detailing its internal workings.

2.2 Context Diagram for AuraCheck

System Name:

- Mobile-Based Attendance Management System (AuraCheck)

External Entities:

- **Student:** Enrolls facial data, checks into classes, views attendance history.
- **Instructor:** Manages class sessions, monitors attendance, generates reports, overrides attendance records.
- **Administrator:** Manages users (students, instructors, other administrators), courses, geofences, system settings, and generates system-wide reports.

Key Data Flows:

- **Student to System:**
  - Login credentials (Matricule Number/Email, Password)
  - Facial enrollment data (Feature Vector)
  - Check-in request (Session ID, Location Data, Facial Feature Vector)
  - Attendance history request (Course ID, Date Range)
- **System to Student:**
  - Login response (Success/Failure)
  - Enrollment feedback (Success/Failure, Reason)
  - Check-in result (Success/Failure, Reason)
  - Attendance history data (Records, Summaries)
- **Instructor to System:**
  - Login credentials (Staff ID/Email, Password)
  - Start session command (Course ID)
  - End session command (Session ID)
  - Monitor attendance request (Session ID)
  - Generate report request (Course ID, Date Range)
  - Manual override request (Student ID, Session ID, Status, Justification)
- **System to Instructor:**
  - Login response (Success/Failure)
  - Session start confirmation
  - Session end confirmation
  - Real-time attendance data (Student Names, IDs, Status, Timestamps)
  - Report data (CSV/PDF, Student Details, Summaries)
  - Override confirmation
- **Administrator to System:**
  - Login credentials (ID/Email, Password)
  - User management commands (Create/Update/Delete User)
  - Course management commands (Create/Update/Delete Course)
  - Geofence management commands (Create/Update/Delete Geofence)
  - Enrollment management commands (Enroll/Unenroll Student)
  - System settings configuration (Thresholds, Time Windows)
  - Audit log request (Time Range, User, Event Type)
- **System to Administrator:**
  - Login response (Success/Failure)
  - Management operation confirmations
  - Audit log data (Logs, Details)
  - Configuration confirmation

2.3 Context Diagram

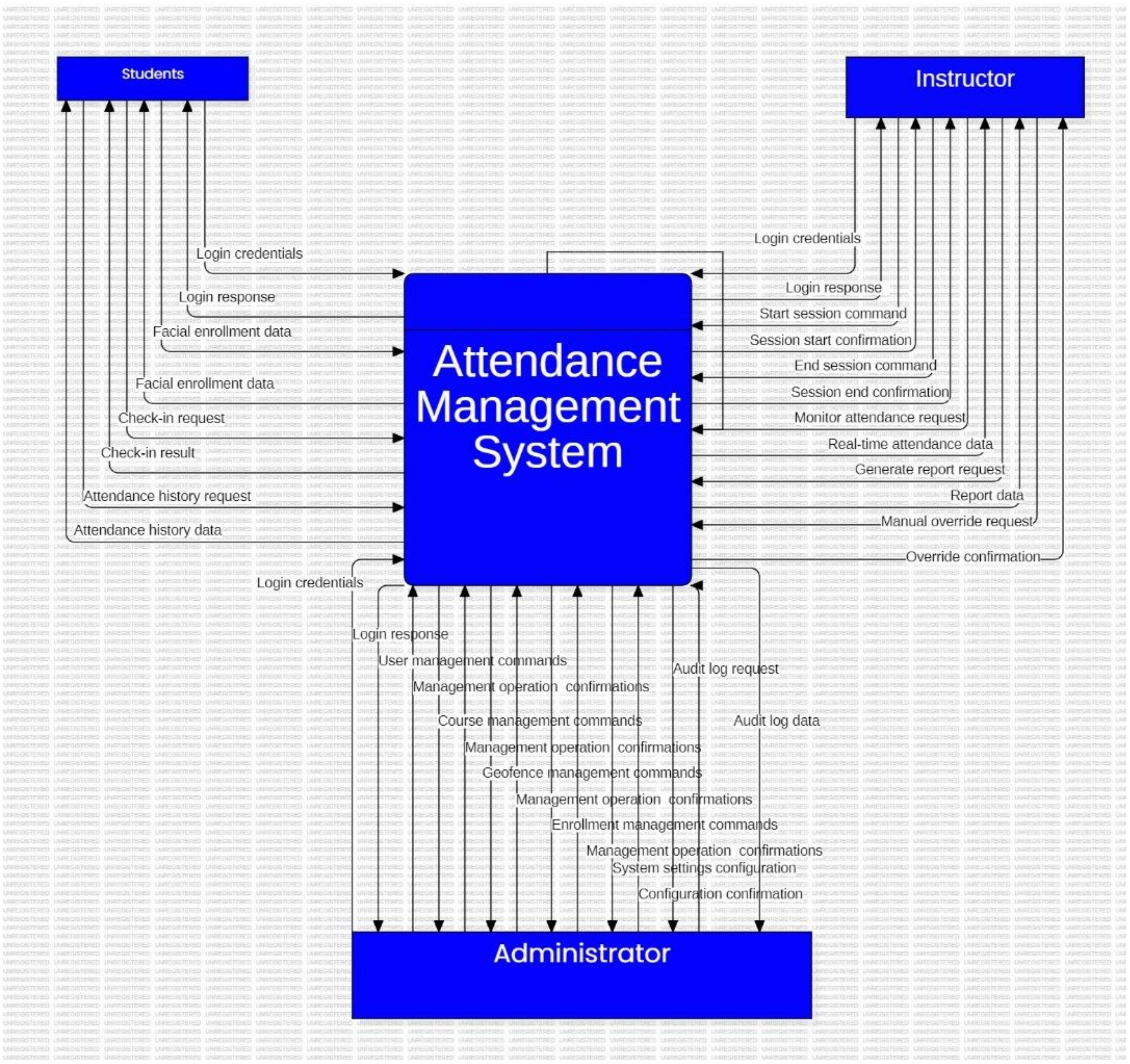

FIGURE 1 - CONTEXT DIAGRAM

2.4 Explanation of AuraCheck Context Diagram

The Context Diagram for AuraCheck illustrates the system as a central process. It clearly shows the three main external entities: Students, Instructors, and Administrators, interacting with the "Attendance Management System." Arrows indicate the direction of data flow, such as students submitting login credentials and facial data, and receiving attendance history. Instructors send commands to manage sessions and receive real-time data. Administrators perform various management tasks and receive confirmations and system-wide data like audit logs. This diagram effectively defines the system's operational environment and its primary interactions.

3. Dataflow Diagram (Level 1)

3.1 Purpose of a Dataflow Diagram

A Dataflow Diagram (DFD) shows how data moves through a system. Unlike a flowchart, it doesn't show control flow or decision logic. A Level 1 DFD decomposes the single process from the Context Diagram into major sub-processes, showing how data flows between these processes and data stores within the system.

3.2 Dataflow Diagram (Level 1) for AuraCheck

Key Processes:

- **Authentication:** Validates user credentials for all roles.
- **Facial Enrollment:** Processes facial data capture and storage.
- **Check-in:** Validates student check-ins using geofencing and facial recognition.
- **Session Management:** Handles session start/end by instructors.
- **Attendance Monitoring:** Displays real-time attendance for instructors.
- **Reporting:** Generates attendance reports for instructors and administrators.
- **Manual Override:** Processes instructor overrides with justifications.
- **User Management:** Manages user account CRUD operations.
- **Course Management:** Manages course CRUD operations.
- **Geofence Management:** Manages geofence definitions.
- **Enrollment Management:** Manages student-course enrollments.
- **System Settings Configuration:** Adjusts system parameters.
- **Audit Logging:** Logs system events for accountability.

Key Data Stores:

- **Users:** Stores user details (userID, fullName, email, passwordHash, role, status).
- **Courses:** Stores course details (courseID, courseCode, courseName, description, instructorID).
- **Enrollments:** Tracks student-course associations (enrollmentID, studentID, courseID).
- **Geofences:** Stores geofence definitions (geofenceID, name, latitude, longitude, radius).
- **Sessions:** Stores session details (sessionID, courseID, startTime, endTime, geofenceID).
- **Attendance Records:** Stores attendance data (attendanceID, studentID, sessionID, status, checkInTimestamp).
- **FacialTemplates:** Stores encrypted biometric templates (templateID, studentID, featureVector).
- **AuditLogs:** Stores event logs (logID, timestamp, userID, eventType, affectedResource).

Key Data Flows (Examples):

- **Authentication Process:**
    - Input: Credentials from Students, Instructors, Administrators.

    - Reads from: `Users` data store.
    - Output: Login response to users.
- **Check-in Process:**
    - Input: Check-in request (Session ID, Location Data, Facial Feature Vector) from Student.
    - Reads from: `Sessions`, `Geofences`, `FacialTemplates`, `Users` data stores.
    - Writes to: `AttendanceRecords`, `AuditLogs` data stores.
    - Output: Check-in result to Student.
- **Session Management Process:**
    - Input: Start/End session command from Instructor.
    - Reads from: `Courses`, `Users` data stores.
    - Writes to: `Sessions`, `AuditLogs` data stores.
    - Output: Session start/end confirmation to Instructor.
- **Reporting Process:**
    - Input: Report request from Instructor or Administrator.
    - Reads from: `AttendanceRecords`, `Courses`, `Users`, `Sessions` data stores.
    - Writes to: `AuditLogs` data store.
    - Output: Report data to Instructor or Administrator.

3.3 Dataflow Diagram (Level 1)

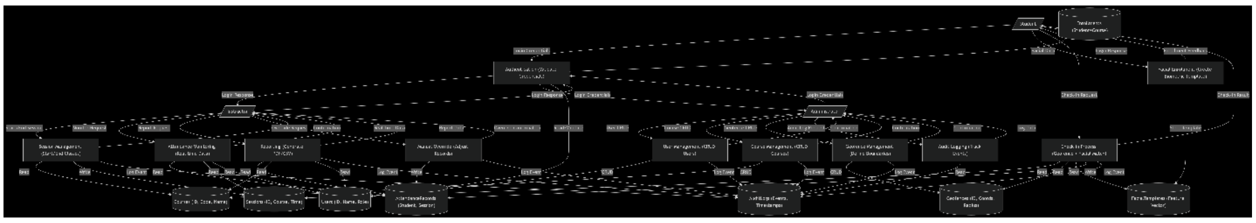

FIGURE 2-LEVEL 1 DATA FLOW DIAGRAM

3.4 Explanation of AuraCheck Dataflow Diagram (Level 1)

The Level 1 DFD for AuraCheck breaks down the "Attendance Management System" into its major functional components. For instance, it shows processes like "Authentication," "Facial Enrollment," "Check-in," "Session Management," and "Reporting." Data flows connect these processes to external entities (Students, Instructors, Administrators) and internal data stores like `Users`, `Courses`, `Attendance Records`, and `FacialTemplates`. This diagram provides a more detailed view of how data is processed within AuraCheck, highlighting the interactions between different functions and where data is stored and retrieved.

4. Use Case Diagram

4.1 Purpose of a Use Case Diagram

A Use Case Diagram describes the functional requirements of a system from the user's perspective. It identifies the different "actors" (users or external systems) that interact with the system and the "use cases" (actions or goals) they can perform. It helps to define the system's scope and its interactions with the outside world.

4.2 Use Case Diagram for AuraCheck

Actors:

- **Student:** Enrolled users who check into classes and view attendance records.
- **Instructor:** Faculty members who manage sessions and monitor attendance.
- **Administrator:** System managers who configure users, courses, and settings.

Key Use Cases:

- **Student:**
    - Login (includes Authenticate Users)
    - Enroll Facial Data (includes Capture Facial Image, Create Biometric Template)
    - Check-in to Class (includes Record Attendance, Verify Geofence Location, Verify Facial Features)
    - View Attendance History
- **Instructor:**
    - Login (includes Authenticate Users)
    - Start Class Session
    - End Class Session
    - Monitor Real-time Attendance
    - Generate Attendance Reports (can extend to Filter Reports, Export Report as PDF/CSV)
    - Manually Override Attendance
- **Administrator:**
    - Login (includes Authenticate Users)
    - Manage User Accounts
    - Manage Courses
    - Manage Geofences
    - Manage Student Enrollments
    - Generate System-wide Reports (can extend to Filter Reports, Export Report as PDF/CSV)
    - View Audit Logs (includes Filter Audit Logs)
    - Configure System Settings

Relationships:

- **<>:** Used when one use case incorporates the functionality of another (e.g., Login *includes* Authenticate Users).
- **<>:** Used when one use case provides optional functionality for another (e.g., Generate Reports *extends* to Export Report as PDF).

4.3 Use Case Diagram

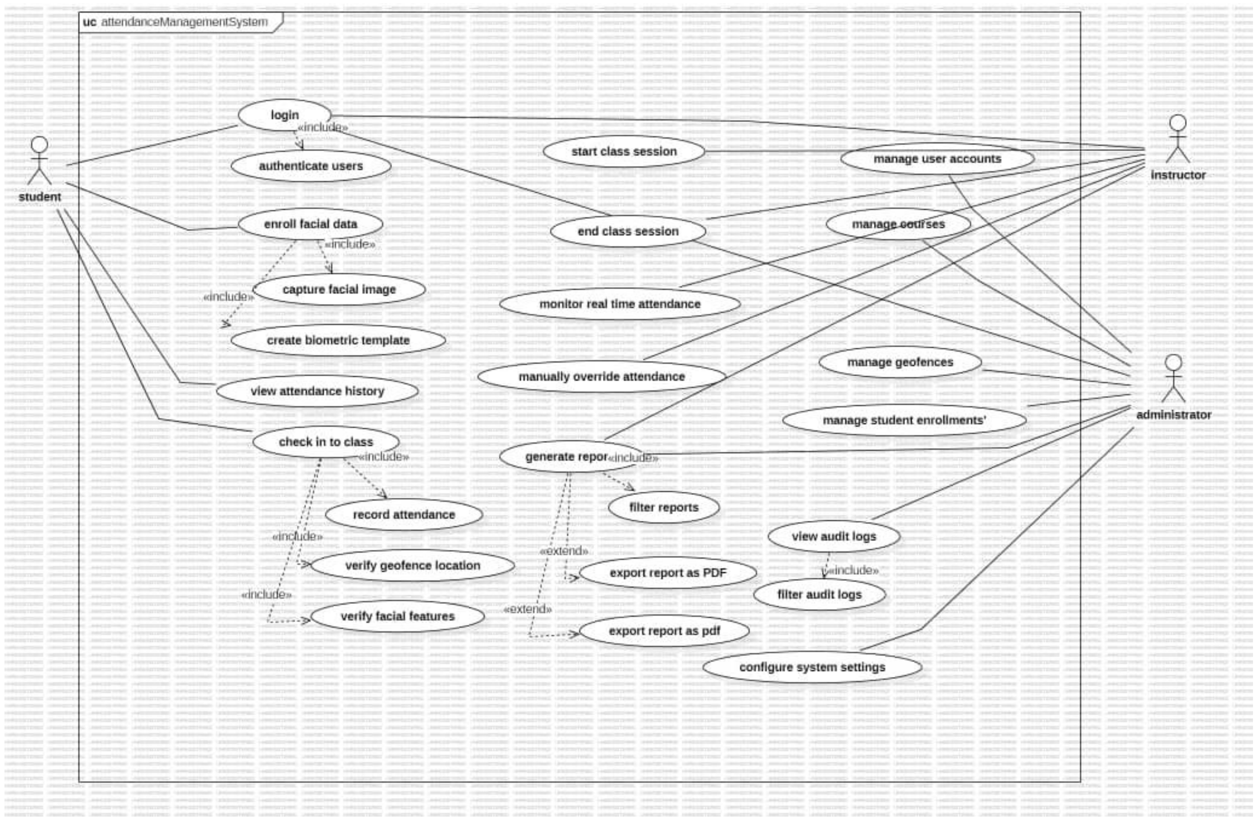

FIGURE 3-USE CASE DIAGRAM

4.4 Explanation of AuraCheck Use Case Diagram

The Use Case Diagram for AuraCheck depicts the interactions between the three actors (Student, Instructor, Administrator) and the system. Each actor is associated with a set of use cases representing their goals. For example, a Student can "Enroll Facial Data," "Check-in to Class," and "View Attendance History." An Instructor can "Start Class Session," "Monitor Real-time Attendance," and "Generate Attendance Reports." Administrators have broader capabilities like "Manage User Accounts," "Manage Courses," and "Configure System Settings." Relationships like `<<include>>` (e.g., "Check-in to Class" includes "Verify Geofence Location" and "Verify Facial Features") and `<<extend>>` (e.g., "Generate Report" can be extended by "Export report as PDF") further clarify the functional dependencies and options.

5. Sequence Diagram (Check-in Process)

5.1 Purpose of a Sequence Diagram

A Sequence Diagram is an interaction diagram that shows how objects interact with each other and the order in which these interactions occur over time. It focuses on the timeline of messages exchanged between objects to accomplish a specific scenario or use case.

5.2 Sequence Diagram for AuraCheck Check-in Process

Objects/Participants:

- **Student:** The user initiating the check-in.
- **MobileApp:** The Flutter application running on the student's device.
- **BackendAPI:** The FastAPI backend server handling business logic and data validation.
- **Firestore:** The Firebase Firestore database used for data storage.

Sequence of Messages/Interactions:

1. **Student -> MobileApp:** Initiate check-in for session (sessionID).
2. **MobileApp -> MobileApp (self-call):** Acquire device location (using Geolocator).
3. **MobileApp -> BackendAPI:** `GET /sessions/{sessionID}/geofence` (Request geofence parameters for the session).
4. **BackendAPI -> Firestore:** Get session document (`db.collection('Sessions').doc(sessionID).get()`).
5. **Firestore --> BackendAPI:** Return Session data (including geofenceID).
6. **BackendAPI -> Firestore:** Get geofence document (`db.collection('Geofences').doc(geofenceID).get()`).
7. **Firestore --> BackendAPI:** Return Geofence data (latitude, longitude, radius).
8. **BackendAPI --> MobileApp:** Return Geofence parameters.
9. **MobileApp -> MobileApp (self-call):** Verify location is within the geofence boundary.
10. **MobileApp -> MobileApp (self-call):** If within geofence, capture face (using Google ML Kit).
11. **MobileApp -> MobileApp (self-call):** Extract facial feature vector from the captured image.
12. **MobileApp -> BackendAPI:** `POST /checkin` (Send sessionID, studentID, and the extracted featureVector).
13. **BackendAPI -> Firestore:** Get student's stored facial template (`db.collection('FacialTemplates').where('studentID', '=', studentID).get()`).
14. **Firestore --> BackendAPI:** Return stored facial feature vector.
15. **BackendAPI -> BackendAPI (self-call):** Compare the newly extracted feature vector with the stored template (similarity score > threshold).
16. **BackendAPI -> Firestore:** If both geofence and facial recognition are successful, add attendance record (`db.collection('Attendance Records').add({studentID, sessionID, status: 'Present', checkInTimestamp})`).
17. **Firestore --> BackendAPI:** Return confirmation of record creation.
18. **BackendAPI --> MobileApp:** Return Check-in success response.
19. **MobileApp --> Student:** Display success message.

5.3 Sequence Diagram

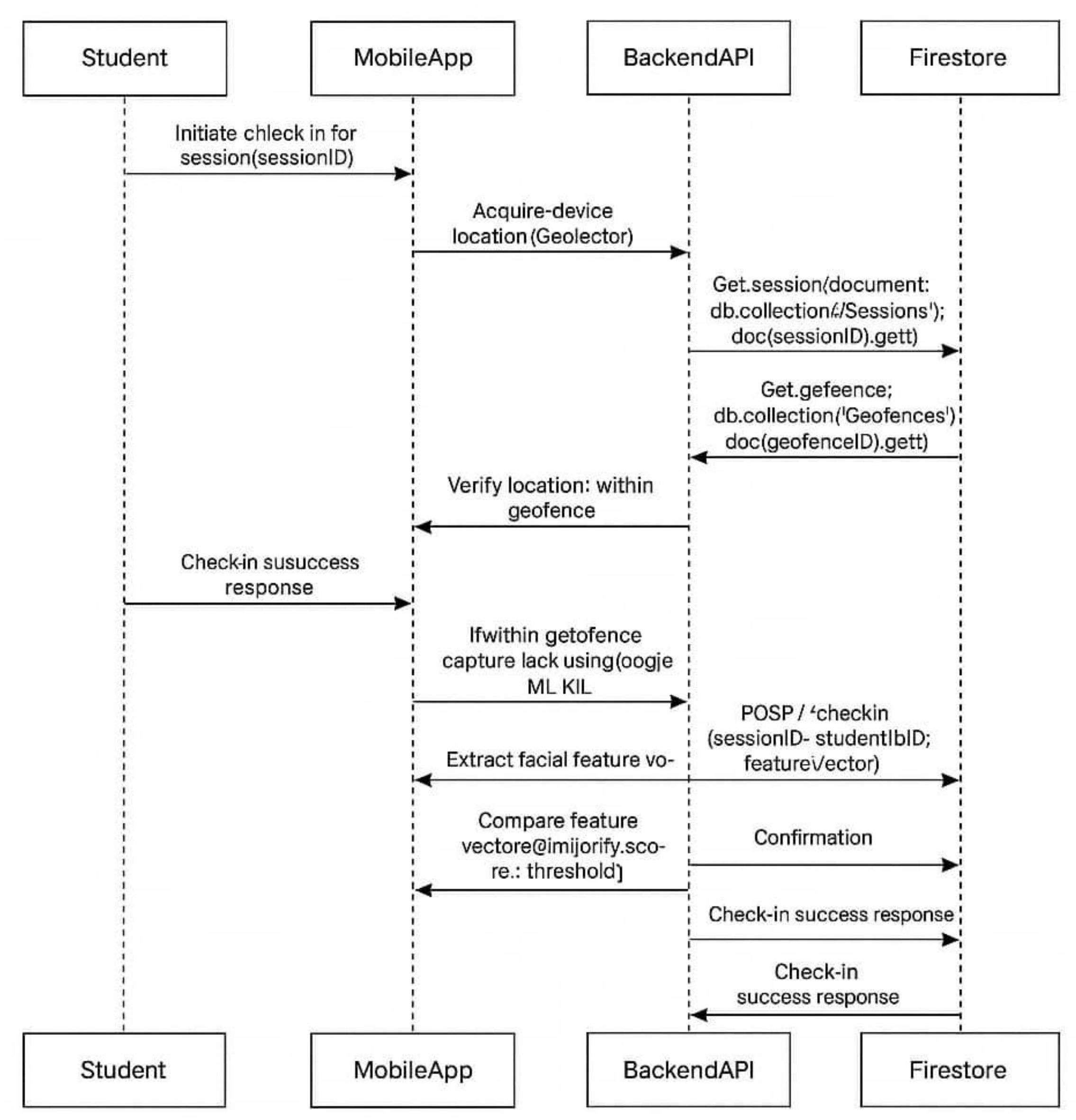
Student
MobileApp
BackendAPI
Firestore
Initiate chleck in for session(sessionID)
Acquire-device location (Geolector)
Get.session(document: db.collection(/Sessions'); doc(sessionID).gett)
Get.gefeence; db.collection('Geofences') doc(geofenceID).gett)
Verify location: within geofence
Check-in susuccess response
Ifwithin getofence capture lack using(oogje ML KIL
POSP / 'checkin (sessionID- studentIbID; featureVector)
Extract facial feature vo-
Compare feature vectore@imijorify.sco-re.: threshold)
Confirmation
Check-in success response
Check-in success response
Student
MobileApp
BackendAPI
Firestore

FIGURE 4=SEQUENCE DIAGRAM

5.4 Explanation of AuraCheck Check-in Process Sequence Diagram

This Sequence Diagram details the step-by-step interactions for the "Check-in to Class" use case. Lifelines represent the participants: Student, MobileApp, BackendAPI, and Firestore. Messages are shown as arrows between these lifelines, ordered chronologically from top to bottom. The diagram clearly illustrates the process: the student initiates check-in via the MobileApp, which then acquires location, validates it against the geofence data fetched from the BackendAPI (which in turn queries Firestore), captures and processes facial data, sends it for verification to the BackendAPI (which again queries Firestore for the stored template), and finally records attendance if all checks pass. This visualizes the flow of control and data exchange over time for this critical function.

6. Class Diagram

6.1 Purpose of a Class Diagram

A Class Diagram is a static structure diagram that describes the structure of a system by showing its classes, their attributes, methods (or operations), and the relationships between those classes. It is a fundamental building block for object-oriented modeling.

6.2 Class Diagram for AuraCheck

Key Classes, Attributes, and Methods (based on "ClassDiagram1.jpg" and "Diagrams Description.pdf"):

- **User:**
  - Attributes: `userID: string`, `fullName: string`, `email: string`, `passwordHash: string`, `role: string` (enum: 'Student', 'Instructor', 'Admin'), `status: string` (enum: 'Active', 'Inactive'), `createdAt: timestamp`, `updatedAt: timestamp`.
  - Methods: `login(email, password)`, `logout()`, `updateProfile(fullName, email)`, `changePassword(oldPwt, newPwt)`, `deactivate()`, `getAttendanceHistory(...)`.
- **Course:**
  - Attributes: `courseID: string`, `courseCode: string`, `courseName: string`, `description: string`, `instructorID: string` (refers to User), `createdAt: timestamp`, `updatedAt: timestamp`.
  - Methods: `addSession(...)`, `removeSession(...)`, `assignInstructor(...)`, `getSessions()`, `generateReport(...)`.
- **Enrollment:**
  - Attributes: `enrollmentID: string`, `studentID: string` (refers to User), `courseID: string` (refers to Course), `enrollmentDate: timestamp`, `status: string`.
  - Methods: `enrollStudent(student, course)`, `cancelEnrollment()`, `updateStatus(newStatus)`.
- **Geofence:**
  - Attributes: `geofenceID: string`, `name: string`, `latitude: number`, `longitude: number`, `radius: number`, `status: string`, `createdAt: timestamp`, `updatedAt: timestamp`.
  - Methods: `containsLocation(lat, lon)`, `updateLocation(lat, lon, radius)`, `activate()`, `deactivate()`.
- **Session:**
  - Attributes: `sessionID: string`, `courseID: string` (refers to Course), `startTime: timestamp`, `endTime: timestamp`, `geofenceID: string` (refers to Geofence), `status: string` (enum: 'Scheduled', 'Active', 'Ended'), `createdAt: timestamp`, `updatedAt: timestamp`.
  - Methods: `start()`, `end()`, `isActive()`, `getAttendance()`, `overrideAttendance(...)`.

- **AttendanceRecord:**
    - Attributes: `attendanceID: string`, `studentID: string` (refers to User), `sessionID: string` (refers to Session), `status: string` (enum: 'Present', 'Absent'), `checkInTimestamp: timestamp`, `overrideJustification: string` (optional), `createdAt: timestamp`.
    - Methods: `markPresent()`, `markAbsent()`, `overrideStatus(newStatus, justification)`, `getDuration()`.
- **FacialTemplate:**
    - Attributes: `templateID: string`, `studentID: string` (refers to User), `featureVector: binary` (encrypted), `enrollmentDate: timestamp`.
    - Methods: `enrollFeatureVector(featureVector)`, `verify(featureVector)`, `updateTemplate(newFeatureVector)`.
- **AuditLog:**
    - Attributes: `logID: string`, `timestamp: timestamp`, `userID: string` (refers to User), `eventType: string`, `affectedResource: string`, `outcome: string`, `context: string`.
    - Methods: `logEvent(...)`, `queryLogs(...)`.

Relationships (Examples):

- A `User` (Instructor) can teach multiple `Course`s (1 to many).
- A `User` (Student) can have multiple `Enrollment`s (1 to many).
- An `Enrollment` links one `User` (Student) to one `Course` (Many-to-many effectively through Enrollment).
- A `Course` can have multiple `Session`s (1 to many).
- A `Session` uses one `Geofence` (1 to 1 or Many to 1 if geofences can be reused).
- A `Session` can have multiple `AttendanceRecord`s (1 to many).
- An `AttendanceRecord` is associated with one `User` (Student) and one `Session`.
- A `User` (Student) has one `FacialTemplate` (1 to 1).
- An `AuditLog` entry is associated with one `User`.

6.3 Class Diagram

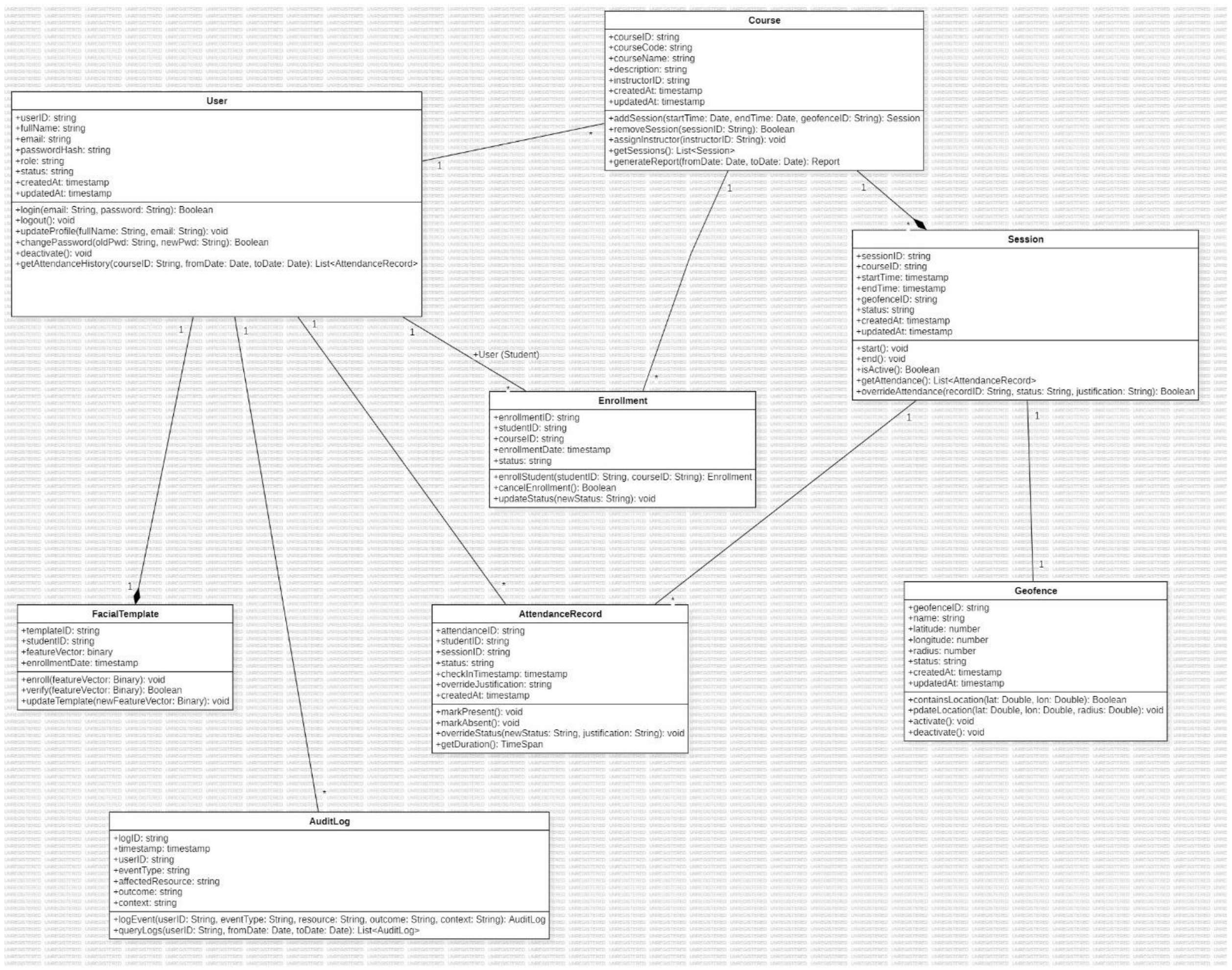
Course
+courseID: string
+courseCode: string
+courseName: string
+description: string
+instructorID: string
+createdAt: timestamp
+updatedAt: timestamp
+addSession(startTime: Date, endTime: Date, geofenceID: String): Session
+removeSession(sessionID: String): Boolean
+assignInstructor(instructorID: String): void
+getSessions(): List<Session>
+generateReport(fromDate: Date, toDate: Date): Report
User
+userID: string
+fullName: string
+email: string
+passwordHash: string
+role: string
+status: string
+createdAt: timestamp
+updatedAt: timestamp
+login(email: String, password: String): Boolean
+logout(): void
+updateProfile(fullName: String, email: String): void
+changePassword(oldPwd: String, newPwd: String): Boolean
+deactivate(): void
+getAttendanceHistory(courseID: String, fromDate: Date, toDate: Date): List<AttendanceRecord>
Session
+sessionID: string
+courseID: string
+startTime: timestamp
+endTime: timestamp
+geofenceID: string
+status: string
+createdAt: timestamp
+updatedAt: timestamp
+start(): void
+end(): void
+isActive(): Boolean
+getAttendance(): List<AttendanceRecord>
+overrideAttendance(recordID: String, status: String, justification: String): Boolean
+User (Student)
Enrollment
+enrollmentID: string
+studentID: string
+courseID: string
+enrollmentDate: timestamp
+status: string
+enrollStudent(studentID: String, courseID: String): Enrollment
+cancelEnrollment(): Boolean
+updateStatus(newStatus: String): void
FacialTemplate
+templateID: string
+studentID: string
+featureVector: binary
+enrollmentDate: timestamp
+enroll(featureVector: Binary): void
+verify(featureVector: Binary): Boolean
+updateTemplate(newFeatureVector: Binary): void
AttendanceRecord
+attendanceID: string
+studentID: string
+sessionID: string
+status: string
+checkInTimestamp: timestamp
+overrideJustification: string
+createdAt: timestamp
+markPresent(): void
+markAbsent(): void
+overrideStatus(newStatus: String, justification: String): void
+getDuration(): TimeSpan
Geofence
+geofenceID: string
+name: string
+latitude: number
+longitude: number
+radius: number
+status: string
+createdAt: timestamp
+updatedAt: timestamp
+containsLocation(lat: Double, lon: Double): Boolean
+pdateLocation(lat: Double, lon: Double, radius: Double): void
+activate(): void
+deactivate(): void
AuditLog
+logID: string
+timestamp: timestamp
+userID: string
+eventType: string
+affectedResource: string
+outcome: string
+context: string
+logEvent(userID: String, eventType: String, resource: String, outcome: String, context: String): AuditLog
+queryLogs(userID: String, fromDate: Date, toDate: Date): List<AuditLog>

FIGURE 5-CLASS DIAGRAM

6.4 Explanation of AuraCheck Class Diagram

The Class Diagram for AuraCheck outlines the static structure of the system. Key classes include `User`, `Course`, `Session`, `AttendanceRecord`, `FacialTemplate`, `Geofence`, and `Enrollment`. Each class box displays its name, attributes (data members), and methods (operations). Lines connecting the classes represent relationships such as associations (e.g., a `User` is associated with `AttendanceRecord`s), and multiplicities indicate the number of instances involved (e.g., one `Course` can have many `Session`s). This diagram is essential for understanding the data entities, their properties, behaviors, and how they relate to each other, forming the blueprint for the database schema and object-oriented code structure.

7. Deployment Diagram

7.1 Purpose of a Deployment Diagram

A Deployment Diagram visualizes the physical deployment of software artifacts on hardware nodes. It shows the runtime configuration of processing nodes (e.g., servers, devices) and the components (software artifacts) that reside on them. It's crucial for understanding the system's physical architecture and how its parts are distributed.

7.2 Deployment Diagram for AuraCheck

Nodes (Hardware/Execution Environments):

- **Mobile Device:** Represents the students' and instructors' smartphones.
  - Operating Systems: Android 6.0+, iOS 12.0+.
- **Azure Container Instance (Execution Environment):** Hosts the backend application.
- **Azure Container Registry (Execution Environment):** Stores Docker images for deployment.
- **Firebase Firestore (Execution Environment):** Provides the NoSQL cloud database.

Artifacts (Software Components Deployed on Nodes):

- **On Mobile Device:**
  - `Mobile App (Flutter)`: The client application used by students and instructors.
- **On Azure Container Instance:**
  - `Docker Container` containing:
    - `Backend API (Python - FastAPI, RESTful)`: Handles business logic, API requests.
    - `Web Server (serving Admin Interface static files, React SPA)`: Provides the administrative web interface. (Note: The description mentions a React SPA for Admin, which is a common pattern).
- **On Firebase Firestore:**
  - `Firestore Database`: Stores all application data (Users, Courses, Attendance, etc.).

Communication Paths and Protocols:

- **Mobile App (on Mobile Device) <-> Backend API (on Azure Container Instance):**
  - Protocol: HTTPS (TLS 1.2+)
  - Communication: RESTful API calls.
- **Backend API (on Azure Container Instance) <-> Firestore Database (on Firebase):**
  - Protocol: Firebase Admin SDK (which typically uses gRPC over HTTPS).
- **Azure Container Instance <-> Azure Container Registry:**
  - Deployment relationship (pulling Docker images).

Notes from Description:

- The Admin Interface is a React single-page application served by the FastAPI backend.
- Deployment is managed via GitHub Actions to Azure Container Instances.
- Authentication uses JWT tokens.
- All communications are secured with HTTPS.
-

7.3 Deployment Diagram

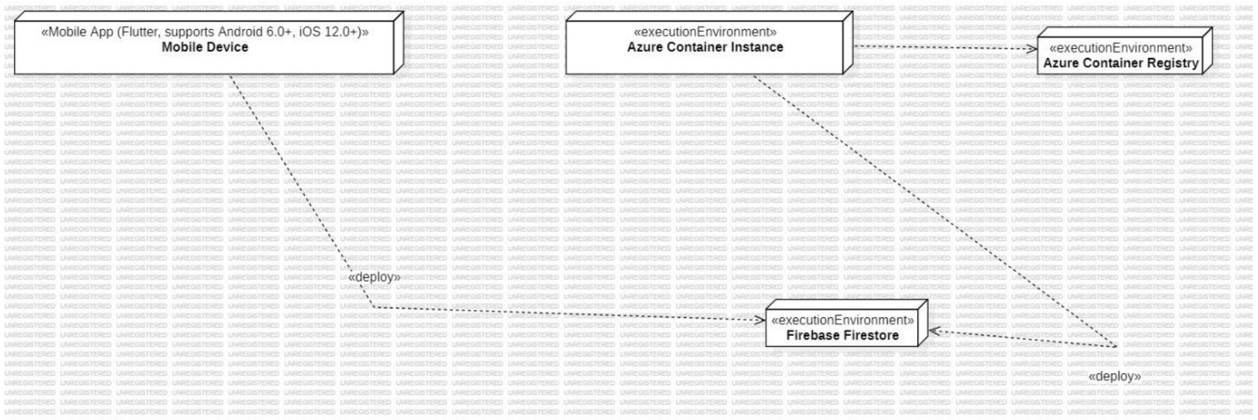

FIGURE 6-DEPLOYMENT DIAGRAM

7.4 Explanation of AuraCheck Deployment Diagram

The Deployment Diagram for AuraCheck shows the physical architecture. The `Mobile Device` node runs the `Mobile App (Flutter)`. The backend, consisting of the `Backend API` and `Web Server` for the admin interface, is deployed as a `Docker Container` within an `Azure Container Instance`. This container image is sourced from `Azure Container Registry`. The `Firebase Firestore` node hosts the application's database. Communication paths are depicted, such as the Mobile App connecting to the Backend API via HTTPS, and the Backend API communicating with Firestore using the Firebase Admin SDK. This diagram clarifies how software components are distributed across physical or virtual hardware, providing insight into the operational environment.

8. Conclusion

The system modelling and design phase for the AuraCheck Mobile-Based Attendance Management System has been detailed through various diagrams. The Context Diagram established the system's boundaries and external interactions. The Level 1 Dataflow Diagram provided insight into the primary internal processes and data stores. The Use Case Diagram defined the system's functionality from the perspective of its different users. The Sequence Diagram for the check-in process illustrated the dynamic interactions between components over time for a critical scenario. The Class Diagram outlined the static structure, attributes, and relationships of the core entities within the system. Finally, the Deployment Diagram visualized the physical architecture and distribution of software components.

Collectively, these models provide a comprehensive understanding of AuraCheck's scope, functionality, structure, behavior, and physical deployment. They serve as a vital foundation for the subsequent development, testing, and deployment phases, ensuring clarity and alignment among the project team and stakeholders. This structured approach to modelling helps in mitigating risks, managing complexity, and ultimately delivering a robust and effective attendance management solution for the University of Buea.