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

Database Design and Implementation of a Mobile-Based Attendance System Using Geo-Fencing and Facial Recognition

By: Group 24

Option: Software Engineering

CEF440:Internet programming and Mobile programming

Course Master:

Dr. Nkemeni Valery

University of Buea

2024/2025 Academic Year

Name	Matricule Number
BILLA SOPHIA	FE22A176
EKANE METUGE AKAME FAVOUR	FE22A199
EYONG GODWILL NGANG	FE22A214
ONYA MARTHA	FE22A292
NEBOTA ISMAEL OWAMBA	FE22A256

CONTENTS

1. INTRODUCTION

1.1. OVERVIEW OF DATABASE DESIGN AND IMPLEMENTATION

The database is the foundational pillar of the AuraCheck mobile-based attendance management system. Its design and implementation are critical to the application's data integrity, security, and operational performance. Given that the system relies on real-time, sensitive data—including user biometrics (facial recognition) and geolocation—it is imperative to structure a database that is robust, scalable, and secure. This design employs a non-relational, document-based model that clearly defines data collections for users, courses, attendance sessions, and more. This approach ensures efficient handling of complex, semi-structured data, secure storage of sensitive information, and fast data retrieval for the system's various modules.

The implementation leverages Google Firebase, a modern backend-as-a-service (BaaS) platform. Specifically, Cloud Firestore is used for the database, Firebase Authentication for user management, and Firebase Storage for binary data like facial templates. This choice provides a highly scalable, real-time, and secure infrastructure that integrates seamlessly with both the Flutter frontend and the Python backend.

1.2. PURPOSE AND SCOPE OF THE TASK

The primary purpose of this task is to design and implement a database system that effectively supports all core functionalities of the AuraCheck application. These functionalities include authenticating user identity via facial recognition, verifying physical presence through GPS geofencing, and recording attendance in real-time. The system must guarantee high data reliability, support concurrent access from multiple user roles (students, instructors, and administrators), and maintain a secure and structured repository for all user and attendance-related data.

The scope of this task encompasses:

- Conceptual modeling to identify key data entities and their relationships.
- Translating the conceptual model into a physical database schema using Cloud Firestore collections.
- Defining data structures, types, and validation rules.
- Implementing the database and integrating it with the Flask backend server.

- Ensuring the design is scalable for future enhancements, such as advanced analytics and reporting.

2. DATA ELEMENTS

2.1. DESCRIPTION OF DATA COMPONENTS

The AuraCheck system is built around a set of core data components that enable the accurate recording, verification, and management of attendance. These components are modeled as collections in the Firestore database.

- **Users**: The central collection representing every individual in the system. It stores profile information, role, and status. This collection is the single source of truth for user identity.
- **Courses**: Contains information about academic courses, including their code, name, description, and the instructor responsible.
- **Enrollments**: A mapping collection that links students to the courses they are registered for, effectively managing the many-to-many relationship between them.
- **Sessions**: Represents a specific class or lecture instance for a course. Each session has a start time, end time, and is linked to a geofence for location-based validation.
- **Attendance**: Records each student's check-in for a specific session. It stores the check-in timestamp, verification status, and any manual overrides.
- **Geofences**: Defines the virtual geographic boundaries for attendance. Each geofence consists of latitude, longitude, and a radius, which outlines the permissible area for check-in.
- **Facial Templates**: While not a Firestore collection, this data element is crucial. It consists of encrypted binary data representing a user's facial features, stored securely in Firebase Storage.

2.2. USER ROLES AND DATA ACCESS

AuraCheck implements a robust Role-Based Access Control (RBAC) model to ensure data security and appropriate access levels for its three distinct user roles.

- **Student**:
  - **Can access**: Their personal profile, enrolled courses, and attendance history.

    - **Can perform**: Facial enrollment, check-in for active sessions (when within the geofence).
    - **Cannot access**: Other users' data, administrative panels, or course management features.
- **Instructor**:
    - **Can access**: Their profile, assigned courses, student enrollment lists for their courses, and real-time attendance data for their active sessions.
    - **Can perform**: Create and manage class sessions, monitor attendance, manually override attendance status with justification, and generate course-specific reports.
    - **Cannot access**: System-wide administrative settings, other instructors' courses, or user management functions.
- **Admin**:
    - **Can access**: Full system-wide data, including all user profiles, courses, attendance logs, and system audit trails.
    - **Can perform**: All CRUD (Create, Read, Update, Delete) operations on users, courses, and geofences. They can manage system settings, assign roles, and monitor overall system health.

This tiered access model is enforced at both the application level (in Flutter and Flask) and through Firebase Security Rules to provide defense-in-depth.

3. CONCEPTUAL DESIGN

This section outlines the conceptual data model for AuraCheck, which forms the blueprint for the physical database implementation in Firestore.

3.1. IDENTIFICATION OF ENTITIES

The conceptual model identifies the following primary entities (which map to Firestore collections):

- users
- courses

- sessions
- attendance
- geofences
- enrollments

3.2. KEY ATTRIBUTES

Each entity contains a set of key attributes that define its structure:

- **users**: userId (PK), fullName, email, matriculeOrStaffId, role, status, hasFacialTemplate, createdAt, updatedAt.
- **courses**: courseId (PK), courseCode, courseName, description, instructorId (FK to users), createdAt, updatedAt.
- **sessions**: sessionId (PK), courseId (FK to courses), startTime, endTime, geofenceId (FK to geofences), status, createdAt, updatedAt.
- **attendance**: attendanceId (PK), studentId (FK to users), sessionId (FK to sessions), status, checkInTimestamp, overrideJustification, overrideBy (FK to users).
- **geofences**: geofenceId (PK), name, latitude, longitude, radius, isActive.
- **enrollments**: enrollmentId (PK), studentId (FK to users), courseId (FK to courses), enrolledAt.

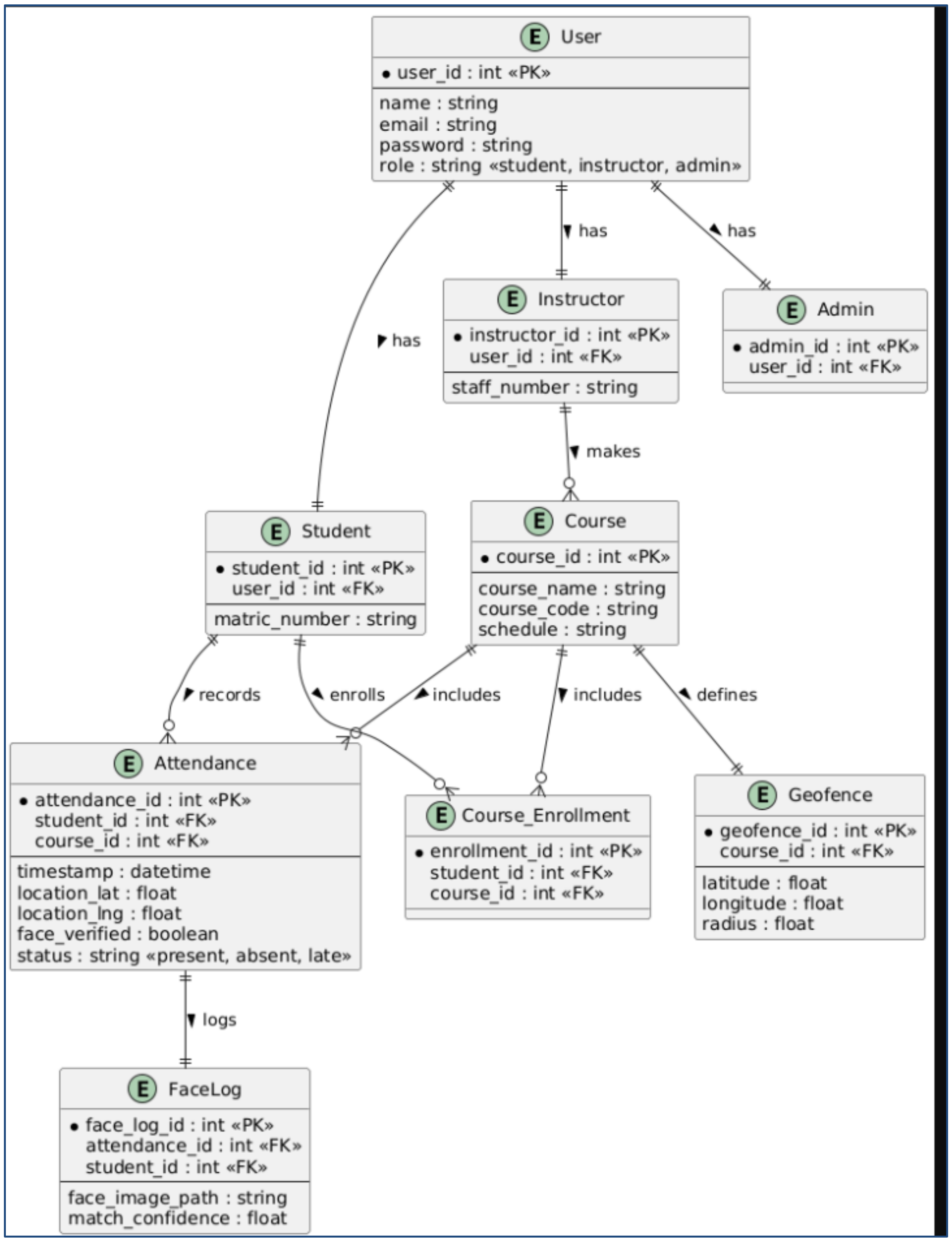
3.3. RELATIONSHIP MAPPING

The relationships between entities are managed through document references (foreign keys):

- **One-to-Many**: An instructor (a user with role 'instructor') can be linked to many courses via the instructorId field in the courses collection. A course can have many sessions.
- **Many-to-Many**: The relationship between users (students) and courses is resolved by the enrollments collection. A student can enroll in many courses, and a course can have many students.
- **Direct Links**:
    - Each attendance record is directly linked to one student and one session.

- Each session is linked to one course and one geofence.
- A student's facial template in Firebase Storage is linked via their userId.

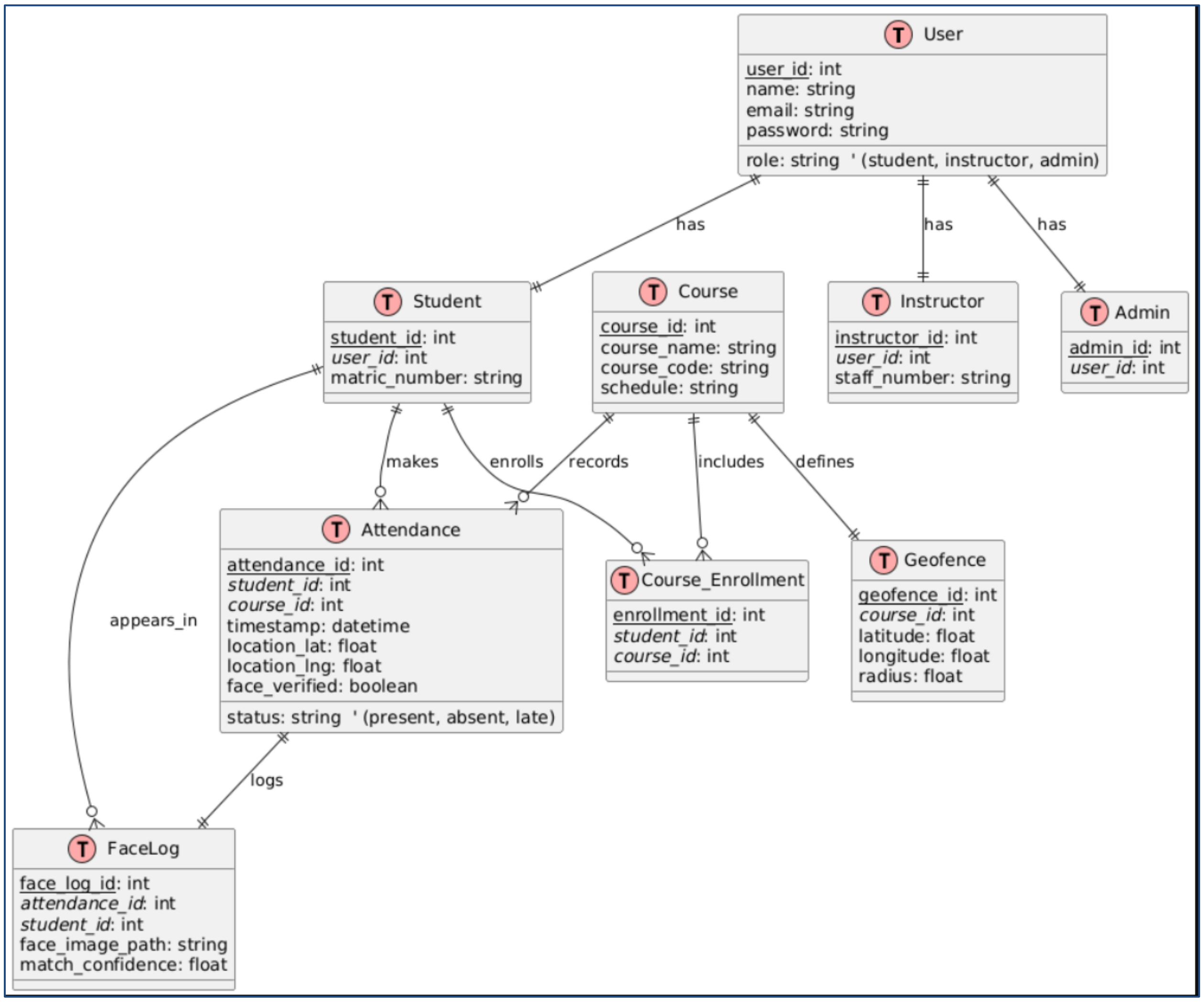
This NoSQL structure is optimized for the access patterns of the application, allowing for efficient fetching of related data in a single query where possible.



4. ER DIAGRAM

4.1. ENTITY-RELATIONSHIP DIAGRAM

While ER diagrams are traditionally associated with relational databases, a conceptual diagram can still effectively visualize the relationships between collections in a NoSQL database like Firestore. The diagram below represents the logical structure of the AuraCheck data model.



4.2. EXPLANATION OF ENTITIES AND RELATIONSHIPS

- **Users**: The central entity. It holds common data for all user types, distinguished by the role field.
    - *Relationship*: A User document's ID is used as a foreign key in courses (for instructors), enrollments (for students), and attendance records.
- **Courses**: Represents an academic course.
    - *Relationship*: It is linked to an instructor (a User) via instructorId. It has a one-to-many relationship with sessions and enrollments.
- **Enrollments**: This is a linking entity that resolves the many-to-many relationship between students and courses.
    - *Relationship*: It contains a studentId and a courseId, connecting a specific user to a specific course.
- **Geofences**: Defines a valid geographical area for check-ins.
    - *Relationship*: It is linked to one or more sessions, defining where attendance can be taken.
- **Sessions**: Represents a single scheduled class.
    - *Relationship*: Each session belongs to one course, is monitored by the course's instructor, and has many associated attendance records.
- **Attendance**: The record of a student's check-in.
    - *Relationship*: It is uniquely linked to one student and one session, creating a timestamped record of presence. It is the core transactional entity of the system.

5. DATABASE IMPLEMENTATION

5.1. CHOICE OF DATABASE MANAGEMENT SYSTEM (DBMS)

The primary database management system chosen for AuraCheck is **Google Cloud Firestore**, supplemented by **Firebase Storage** and a local mobile database, **Hive**.

- **Cloud Firestore (Primary Database)**:

    - **Rationale**: As a fully managed NoSQL document database, Firestore offers immense scalability, powerful querying, and real-time data synchronization. Its real-time listeners are perfect for features like live attendance monitoring. The flexible data model is ideal for the evolving requirements of the application. Its seamless integration with Firebase Authentication and robust security rules makes it a secure and efficient choice.

- **Firebase Storage**:
    - **Rationale**: Used for storing binary files, specifically the encrypted facial recognition templates. Firestore is not optimized for large binary data, so Storage is the appropriate service for this task.
- **Hive (Local Mobile Database)**:
    - **Rationale**: Used on the Flutter client for its offline-first architecture. Hive is a lightweight and fast key-value database for Dart. It caches essential data (like user profiles, course lists) locally, allowing the app to function even with intermittent or no network connectivity. Data is then synchronized with Firestore when a connection is available.

5.2. DATABASE SCHEMA

The physical database schema in Cloud Firestore is structured as follows:

- users/{userId}
- courses/{courseId}
- sessions/{sessionId}
- attendance/{attendanceId}
- geofences/{geofenceId}
- enrollments/{enrollmentId}

5.3. DATA TYPES AND CONSTRAINTS

Firestore supports a range of data types. The key types used in AuraCheck are:

- **String**: For fields like fullName, email, courseCode, role, and IDs.
- **Number**: For latitude, longitude, and radius.

- **Boolean**: For flags like hasFacialTemplate and isActive.
- **Timestamp**: For all date and time fields like createdAt, startTime, and checkInTimestamp. This is crucial for chronological sorting and queries.
- **Reference**: Although not used explicitly here, relationships are maintained by storing the document ID of a related document as a string.

Constraints are primarily enforced through **Firebase Security Rules**. These rules define who can read, write, and query data in the database. For example, a rule can state that a user can only write to their own attendance record (request.auth.uid == resource.data.studentId).

5.4. SAMPLE DATA RECORDS

Below are JSON examples of documents within each major collection.

users/FE22A256

```
{
  "fullName": "Nebota Ismael Owamba",
  "email": "nebota.ismael@example.com",
  "matriculeOrStaffId": "FE22A256",
  "role": "student",
  "status": "active",
  "hasFacialTemplate": true,
  "createdAt": "2024-10-26T10:00:00Z",
  "updatedAt": "2024-11-15T11:30:00Z"
}
```

courses/CEF440

```
{
  "courseCode": "CEF440",
  "courseName": "Internet Programming and Mobile Programming",
  "description": "Advanced topics in web and mobile development.",
  "instructorId": "INS001",
  "createdAt": "2024-09-01T09:00:00Z",
  "updatedAt": "2024-09-01T09:00:00Z"
}
```

sessions/{sessionId}

```
{
  "courseId": "CEF440",
  "startTime": "2025-06-09T14:00:00Z",
  "endTime": "2025-06-09T16:00:00Z",
  "geofenceId": "Amphi_600",
  "status": "active",
  "createdAt": "2025-06-09T13:55:00Z",
  "updatedAt": "2025-06-09T13:55:00Z"
}
```

6. BACKEND IMPLEMENTATION

The backend is a critical component that handles the intensive processing for facial recognition and provides a secure API for the mobile client.

6.1. TECHNOLOGY STACK

- **Language**: Python 3.x
- **Framework**: Flask
- **Facial Recognition**: DeepFace library with the VGG-Face model.
- **Database**: Firebase Admin SDK for Python to interact with Firestore.
- **Authentication**: JSON Web Tokens (JWT) for securing API endpoints.

6.2. API ENDPOINTS FOR ATTENDANCE MANAGEMENT

The Flask application exposes several RESTful API endpoints:

- POST /api/login: Authenticates a user and returns a JWT token.
- POST /api/verify_attendance: The core endpoint. It receives an image, student ID, and session ID. It performs:

1. Liveness detection to prevent spoofing.
2. Facial recognition against the stored template.
3. If successful, it creates or updates the attendance record in Firestore.

- GET /api/user/{userId}: Retrieves user profile information.

6.3. AUTHENTICATION AND VALIDATION

- **Authentication**: API endpoints are protected using JWTs. The mobile client includes the token in the Authorization header of its requests. The Flask backend validates this token before processing the request.
- **Validation**: The backend performs multi-factor validation:
  1. **Identity**: Confirmed via facial recognition.
  2. **Liveness**: Confirmed via an anti-spoofing model to ensure the subject is a live person.
  3. **Location**: The mobile app first confirms the user is within the geofence before even allowing a call to the verification API.

6.4. ERROR HANDLING

The backend implements comprehensive error handling, returning clear JSON responses with appropriate HTTP status codes:

- 401 Unauthorized: Invalid or missing JWT.
- 403 Forbidden: User does not have permission.
- 400 Bad Request: Missing data or invalid image.
- 404 Not Found: User or session not found.
- 500 Internal Server Error: For unexpected server-side issues.

7. CONNECTING DATABASE TO BACKEND

7.1. CONNECTION SETUP

The Flask backend connects to Firestore using the **Firebase Admin SDK**.

1. **Initialization**: The SDK is initialized at application startup using a service account key file (firebase.json). This gives the backend server privileged access to the Firebase project.

```
import firebase_admin
from firebase_admin import credentials, firestore

cred = credentials.Certificate("path/to/firebase.json")
firebase_admin.initialize_app(cred)
db = firestore.client()
```

2. **Connection Pooling**: The db client object manages connections to Firestore efficiently, handling pooling and reuse automatically.

7.2. CRUD OPERATIONS

The backend performs CRUD operations on Firestore collections to manage data.

- **Create**: Adding a new attendance record after successful verification.

```
attendance_ref = db.collection('attendance').document()
attendance_ref.set({
    'studentId': 'FE22A256',
    'sessionId': 'session_123',
    'status': 'present',
    'checkInTimestamp': firestore.SERVER_TIMESTAMP
})
```

- **Read**: Fetching a user's facial template path or profile data.

```
user_doc = db.collection('users').document('FE22A256').get()
if user_doc.exists:
    user_data = user_doc.to_dict()
```

- **Update**: Modifying an existing record, such as an instructor overriding an attendance status.
- **Delete**: Removing records, typically handled by an admin for data management.

7.3. REAL-TIME DATA SYNCHRONIZATION

While the Python backend primarily uses direct queries, the **Flutter frontend** makes extensive use of Firestore's real-time capabilities. The client subscribes to onSnapshot listeners on collections and documents. Whenever data changes in the backend (e.g., a new student checks in), Firestore pushes the update to all subscribed clients in real-time. This is how the instructor's dashboard can display live attendance updates without needing to constantly poll the server.

7.4. TESTING AND DEBUGGING DATABASE CONNECTIVITY

Connectivity is tested through:

- **Unit Tests**: Mocking the Firestore client to test business logic without making actual database calls.
- **Integration Tests**: Using a dedicated test Firebase project to run API tests (e.g., via Postman or pytest) that perform real CRUD operations and verify the data is correctly stored and retrieved.
- **Logging**: The backend includes detailed logging for all database interactions, which helps in debugging issues in the production environment.

8. CONCLUSION

The database architecture for AuraCheck, centered around Cloud Firestore, provides a modern, secure, and highly effective solution for a complex attendance management system. The design successfully decouples data storage from the client and backend, allowing each component to operate efficiently. By leveraging a NoSQL document model, the system gains flexibility and scalability. The integration of Firestore's real-time features, robust security rules, and the processing power of a Python backend for biometrics results in a system that is not only functional but also reliable and secure. This comprehensive design and implementation strategy effectively meets all the core requirements of the project and establishes a solid foundation for future development and expansion.