UNIVERSITY OF BUEA

FACULTY OF

ENGINEERING

AND TECHNOLOGY

REPUBLIC OF CAMEROON

PEACE-WORK-FATHERLAND

FACULTE D'INGINERIE ET

TECHGNOLOGIE

COURSE TITLE: Internet and Mobile Programming

COURSE CODE: CEF440

COURSE INSTRUCTOR: Dr. Nkemeni Valery

TASK 6: DATABASE DESIGN AND IMPLEMENTATION

Group Members:

1.	AGYINGI RACHEL MIFOR	FE22A140
2.	DIONE MELINA MAKOGE	FE22A188
3.	JANE AHONE ELOUNDOU	FE22A253
4.	TIMAH BERRY-NAURA ENYAUGH	FE22A318

PROPOSED BY GROUP 14

Contents

Vital-Signal QoE Mobile Application

Executive Summary

This report outlines the database architecture and implementation of the *Vital-Signal Quality of Experience (QoE)* mobile application. The platform is designed to collect, store, and analyze both user feedback and real-time network performance metrics, delivering actionable insights into mobile service quality for telecommunications providers.

The system is built on **Firebase Firestore**, a flexible, cloud-hosted NoSQL database, and is integrated with a custom **Express.js backend API**. While Firestore is inherently schemaless, a structured data model has been designed and documented to ensure consistency, facilitate data integrity, and support scalable application development. This design enables effective correlation between user experience and network behavior, forming the foundation for data-driven optimization of telecom services.

1. Data Elements

1.1 User-Provided Inputs

The application collects the following user-generated data:

- **Ratings**: Numerical ratings from 1-5 representing user satisfaction
- **Network Issues**: Categorized problems such as "Slow Internet", "No Signal", "Call Drops"
- **Text Comments**: Optional detailed feedback from users
- **User Preferences**: Language settings, data upload policies, and permission grants
- **Contextual Information**: Location, time, and situational context during feedback submission

1.2 Automatically Collected Metrics

The system automatically captures technical performance indicators:

- **Signal Strength**: Measured in dBm and ASU (Arbitrary Strength Units)
- **Network Technology**: Classification as 3G, 4G, 5G, or Wi-Fi
- **Network Operator**: Service provider identification and connectivity status
- **Location Data**: GPS coordinates with accuracy measurements

- **Device Information**: Operating system, device model, and SIM card switching events
- **Temporal Data**: Timestamps for all data collection events

1.3 System-Generated Elements

The application generates unique identifiers and metadata:

- **User Identifiers**: Unique user IDs for session management
- **Session Tracking**: Session IDs linking related activities

1.4 Output Data

The system produces analytical outputs:

- **Network Status Display**: Real-time operator, technology, and signal strength information
- **Feedback History**: Historical user interaction records
- **Application Settings**: User preferences and system configurations
- **Analytics Dashboard**: Signal trends, issue frequency analysis, and satisfaction scores

2. Conceptual Design

2.1 System Architecture Overview

The Vital-Signal application follows a modular architecture consisting of five primary components:

- **Data Collection Module**: Responsible for gathering QoE metrics from both user inputs and automated system monitoring. This module ensures data integrity and validates input parameters before storage.
- **User Feedback Module**: Manages the collection and processing of user ratings, comments, and issue categorization. It provides structured interfaces for feedback submission and handles user preference management.
- **Analytics Module**: Aggregates collected data to generate trends, satisfaction scores, and correlation analyses. This module performs statistical computations and prepares data for visualization.

- **Visualization Module**: Provides dashboard interfaces and geographic mapping capabilities for data presentation. It transforms raw analytics into user-friendly visual representations.
- **Export & Access Module**: Offers API endpoints for third-party data access and integration. This module handles data export functionality and external system integration.

2.2 Data Flow Architecture

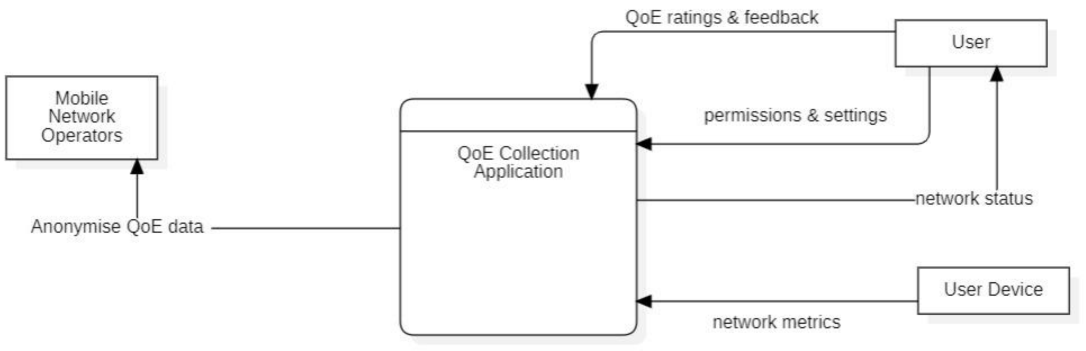

Figure 1: Data Flow Architecture

The data flow architecture illustrates the interaction between key system components in the Vital-Signal QoE application. Mobile Network Operators provide anonymized QoE data that feeds into the central QoE Collection Application. Users interact with the system by providing QoE ratings and feedback while also granting permissions and configuring settings. The User Device component captures network metrics automatically and transmits network status information to the collection system. This bidirectional flow ensures comprehensive data gathering from both automated network monitoring and direct user input, enabling the application to maintain a complete picture of network quality and user experience across different operators and devices.

3. Entity-Relationship (ER) Diagram

3.1 Primary Entities

The database schema consists of six core entities:

User Entity

- Attributes: user_id (Primary Key), device_info, language_preference, permissions
- Description: Stores user profile information and application preferences

Session Entity

- Attributes: session_id (Primary Key), user_id (Foreign Key), start_time, end_time
- Description: Tracks user interaction sessions and temporal boundaries

SignalMetric Entity

- Attributes: metric_id (Primary Key), session_id (Foreign Key), timestamp, signal_strength, network_type, operator, location
- Description: Records technical network performance measurements

Feedback Entity

- Attributes: feedback_id (Primary Key), user_id (Foreign Key), timestamp, rating, issue_type, comment
- Description: Stores user-provided quality assessments and feedback

LocationData Entity

- Attributes: location_id (Primary Key), latitude, longitude, accuracy, timestamp
- Description: Maintains geographic coordinate information with precision metadata

3.2 Relationship Mapping

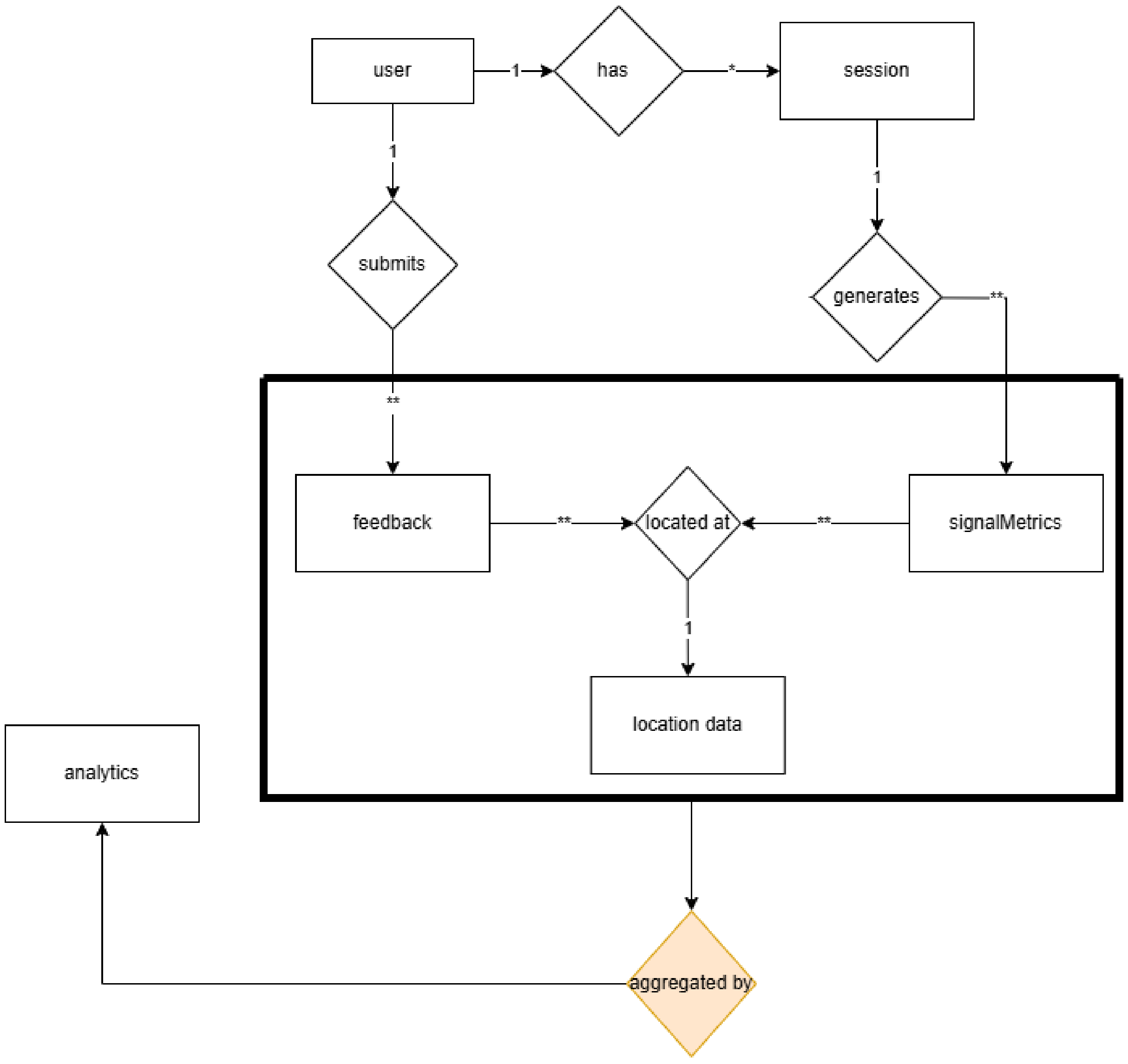

Figure 2: ER Diagram for QOE application

- One User maintains Many Sessions (1:*)
- One Session generates Many SignalMetrics (1:*)
- One User submits Many Feedback entries (1:*)
- SignalMetrics and Feedback entities reference LocationData (*:1)

- The feedback collected, signal metrics and location data are then aggregated to give the analytics

4. Database Implementation

4.1 Technology Selection

Firebase Firestore was selected as the primary database solution due to:

- **Scalability**: Automatic scaling capabilities for growing user bases
- **Real-time Synchronization**: Live data updates across client applications
- **NoSQL Flexibility**: Schema-less design accommodating evolving data structures
- **Integrated Authentication**: Seamless user management and security
- **Global Distribution**: Multi-region data replication for performance optimization

4.2 Collection Structure

The Firestore implementation utilizes the following collection organization:

```
vital-signal (Project)

├── users/

│   └── {user_id}/

├── sessions/

│   └── {session_id}/

├── feedback/

│   └── {feedback_id}/

├── signalMetrics/

│   └── {metric_id}/

├── locationData/

│   └── {location_id}/

└── analytics/

    └── networkFeedback/
```

4.3 Data Normalization Strategy

The implementation follows normalized database principles:

- **First Normal Form (1NF)**: All attributes contain atomic values
- **Second Normal Form (2NF)**: Elimination of partial functional dependencies
- **Third Normal Form (3NF)**: Removal of transitive dependencies through reference-based relationships

4.4 Indexing and Performance Optimization

Although Firebase Firestore is inherently a schemaless NoSQL database, this diagram represents a **conceptual data model** used to organize and structure the application's data consistently.

The design includes top-level collections such as users, feedback, sessions, and signalMetrics, each with clearly defined fields and expected data types. Relationships such as userId references between feedback and users, and sessionId references from signalMetrics are modeled explicitly to reflect how entities interact within the app.

While Firestore does not enforce these structures natively, maintaining a defined schema through design documentation and code validation helps improve **data integrity**, **query performance**, and **collaboration** across development teams.

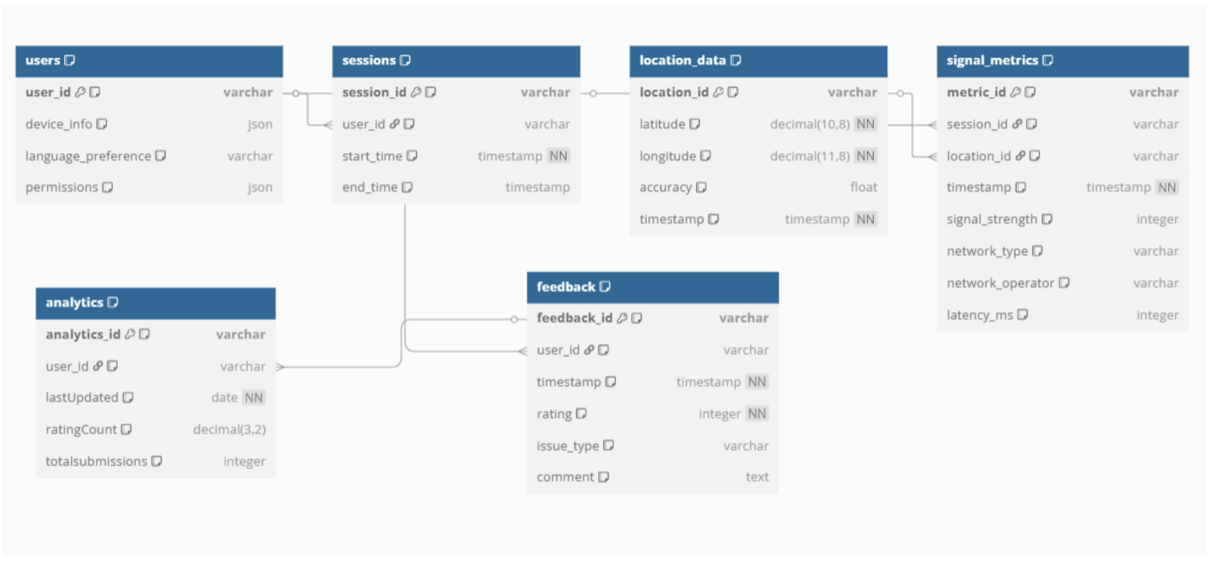

Figure 3: Schema representing QOE databse

Key performance optimizations include:

- Composite indexes on frequently queried field combinations
- Timestamp-based partitioning for temporal data analysis
- Geographic indexing for location-based queries
- Automatic garbage collection for expired session data

5. Backend Implementation

5.1 Technology Stack

The backend infrastructure utilizes:

- **Runtime Environment**: Node.js with Express.js framework
- **Database Integration**: Firebase Admin SDK
- **Security**: CORS middleware and rate limiting
- **Data Validation**: Custom middleware for input sanitization

5.2 API Architecture

The RESTful API implements the following endpoint structure:

5.2.1 Core Endpoints

Feedback Submission Endpoint

POST /api/network-feedback

- Purpose: Accepts user feedback and technical metrics
- Validation: Rating range verification, required field validation
- Processing: Multi-collection data normalization and storage

Analytics Retrieval Endpoint

GET /api/network-feedback/analytics

- Purpose: Provides aggregated feedback statistics
- Parameters: Date range, location filters, network type filters
- Output: Rating distributions, issue frequency, trend analysis

Health Check Endpoint

GET /api/health

- Purpose: System status verification
- Response: Service availability and timestamp information

5.2.2 Utility Endpoints

Network Connectivity Test

GET /ping-google

- Purpose: External connectivity verification
- Implementation: HTTP HEAD request to Google services

5.3 Data Processing Pipeline

The data processing follows this sequence:

1. **Input Validation**: Middleware-based request validation
2. **User Management**: Automatic user creation for new submissions
3. **Session Tracking**: Session initialization and management
4. **Multi-Collection Storage**: Normalized data distribution across collections
5. **Analytics Update**: Asynchronous metrics calculation
6. **Response Generation**: Success confirmation with generated identifiers

5.4 Error Handling and Logging

The system implements comprehensive error management:

- **Validation Errors**: 400 status codes with descriptive messages
- **Server Errors**: 500 status codes with sanitized error information
- **Logging**: Console-based error tracking for debugging

6. Connecting Database to Backend

6.1 Firebase Integration

The connection between the Express.js backend and Firestore database utilizes the Firebase Admin SDK:

```
// Firebase Configuration

const serviceAccount = {

  type: "service_account",

  project_id: process.env.FIREBASE_PROJECT_ID,

  private_key:     process.env.FIREBASE_PRIVATE_KEY?.replace(/\\n/g,
'\n'),

  client_email: process.env.FIREBASE_CLIENT_EMAIL,

  // Additional configuration parameters

};

// Initialize Firebase Admin

admin.initializeApp({

  credential: admin.credential.cert(serviceAccount),

});

// Initialize Firestore

const db = admin.firestore();
```

6.2 Data Access Patterns

The application implements several data access patterns:

Document Creation

```
await db.collection("collectionName").doc(documentId).set(data);
```

Document Retrieval with Filtering (Pseudo code)

```
let query = db.collection("feedback")

  .where("timestamp", ">=", startDate)

  .where("timestamp", "<=", endDate)

  .limit(1000);
```

Transaction Management

```
await db.runTransaction(async (transaction) => {

  // Atomic operations for analytics updates

});
```

6.3 Security Implementation

Security measures include:

- **Environment Variable Protection**: Sensitive credentials stored securely
- **Rate Limiting**: Request throttling to prevent abuse
- **Input Sanitization**: Validation middleware for all endpoints
- **CORS Configuration**: Cross-origin request security

6.4 Performance Considerations

Optimization strategies implemented:

- **Connection Pooling**: Reuse of database connections
- **Asynchronous Processing**: Non-blocking operations for analytics
- **Batch Operations**: Grouped database writes for efficiency
- **Caching Strategy**: In-memory analytics caching for frequent queries

7. Data Validation and Quality Assurance

7.1 Input Validation Strategy

The system implements multi-layer validation:

- **Client-Side Validation**: Initial data format checking
- **Middleware Validation**: Server-side verification before processing
- **Database Constraints**: Firestore security rules for data integrity

7.2 Data Quality Metrics

Quality assurance measures include:

- **Completeness**: Required field verification
- **Consistency**: Cross-reference validation between related entities
- **Accuracy**: Range and format validation for numerical data
- **Timeliness**: Timestamp validation and synchronization

8. Analytics and Reporting Capabilities

8.1 Real-time Analytics

The system provides real-time analytical capabilities:

- **Feedback Aggregation**: Live calculation of average ratings and distributions
- **Issue Frequency Analysis**: Dynamic tracking of common network problems
- **Geographic Analysis**: Location-based service quality mapping
- **Temporal Trends**: Time-series analysis of network performance

9. Scalability

9.1 Horizontal Scaling Strategy

The architecture supports growth through:

- **Firestore Auto-scaling**: Automatic capacity adjustment
- **Load Balancing**: Distribution of API requests across instances
- **Data Partitioning**: Geographic and temporal data segmentation
- **Microservices Migration**: Modular service decomposition potential

10. Conclusion

The Vital-Signal QoE application database implementation successfully addresses the requirements for comprehensive network quality monitoring and user feedback collection. The normalized Firestore schema provides flexibility for future enhancements while maintaining data integrity and performance optimization.

The Express.js backend API offers robust data processing capabilities with comprehensive error handling and security measures. The modular architecture ensures maintainability and scalability as the application grows.

References

Link to backend Repository: LINK TO BACKEND SOURCE CODE
Link to hosted backend: Hosted Backend