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 4: SYSTEM MODELLING AND DESIGN

Group Members:

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

Contents

1. Introduction

Following the comprehensive requirement analysis conducted in Task 3, this document presents the system design and modeling phase of the QoE Collection Application development process. The previous requirement analysis phase established a clear understanding of stakeholder needs, functional and non-functional requirements, and technical constraints. This system design documentation builds directly upon those findings to translate requirements into concrete architectural specifications.

System modeling serves as the critical bridge between requirement analysis and implementation, providing a structured approach to visualize, validate, and communicate the proposed solution architecture. Through systematic modeling, we ensure that all identified requirements are properly addressed in the system design while establishing a clear roadmap for the development team.

2. Importance of System Modeling

System modeling is essential in software engineering for several key reasons that directly apply to our QoE Collection Application:

Requirements Validation: System models serve as a mechanism to verify that all functional and non-functional requirements identified in Task 3 are properly incorporated into the system architecture. Each diagram provides a different lens through which to examine requirement coverage and identify potential gaps.

Communication and Alignment: Visual models create a shared understanding among team members, stakeholders, and future maintainers. By representing complex system interactions through standardized UML notation, we ensure consistent interpretation of system behavior across different audiences.

Design Verification: Before committing to implementation, system models allow us to evaluate architectural decisions, identify potential bottlenecks, and assess the feasibility of proposed solutions. This is particularly critical given the resource constraints and performance requirements established in our MVP scope.

Risk Mitigation: Early identification of system complexities, dependencies, and potential failure points through modeling reduces development risks and prevents costly redesign efforts during implementation phases.

Documentation and Maintenance: Well-structured system models serve as living documentation that guides not only initial development but also future enhancements and maintenance activities outlined in our post-MVP roadmap.

3. Modeling Approach and Methodology

This documentation employs multiple complementary UML modeling perspectives to ensure comprehensive system specification:

Behavioral Modeling: Use Case and Sequence Diagrams capture dynamic system behavior, illustrating how users interact with the system and how system components collaborate to fulfill requirements. These models directly validate the functional requirements (FR1-FR10) established in our requirement analysis.

Structural Modeling: Class Diagrams define the static architecture of system entities, their relationships, and responsibilities. This modeling approach ensures proper encapsulation and maintainability as specified in our non-functional requirements (NFR8).

Data Flow Modeling: Data Flow Diagrams visualize information movement through the system, validating our data management and transmission requirements (FR5) while ensuring privacy and security considerations (NFR5) are properly addressed.

Deployment Modeling: Deployment Diagrams specify the physical distribution of system components, directly addressing scalability (NFR4), compatibility (NFR7), and availability (NFR9) requirements identified in our analysis.

4. Requirements Traceability

Each modeling artifact in this documentation maintains explicit traceability to the requirements established in Task 3. The models collectively address:

- **Must-Have Requirements:** All critical MVP functionality identified in our prioritization framework is represented across the various diagrams
- **Should-Have Requirements:** Important features that enhance user experience are incorporated into the design with clear extension points
- **Could-Have Requirements:** Deferred features are considered in the architecture to ensure future extensibility

5. Technical Foundation

The system models reflect the technical stack decisions made during requirement analysis, specifically the choice of React Native for cross-platform development and Firebase for backend infrastructure. These architectural decisions ensure that our models represent implementable solutions within the identified technical constraints and resource limitations.

The following sections present detailed UML diagrams that collectively define the complete system architecture, from high-level user interactions to low-level component relationships and deployment strategies.

1.0 Use Case Diagram

The Use Case Diagram shows interactions between User, ISP, and Server.

1. **Users can:**
    - Register/Log in
    - Submit feedback (with acknowledgment),
    - Modify app settings, and
    - View feedback.
2. **The ISP** manages feedback info and accesses user feedback.

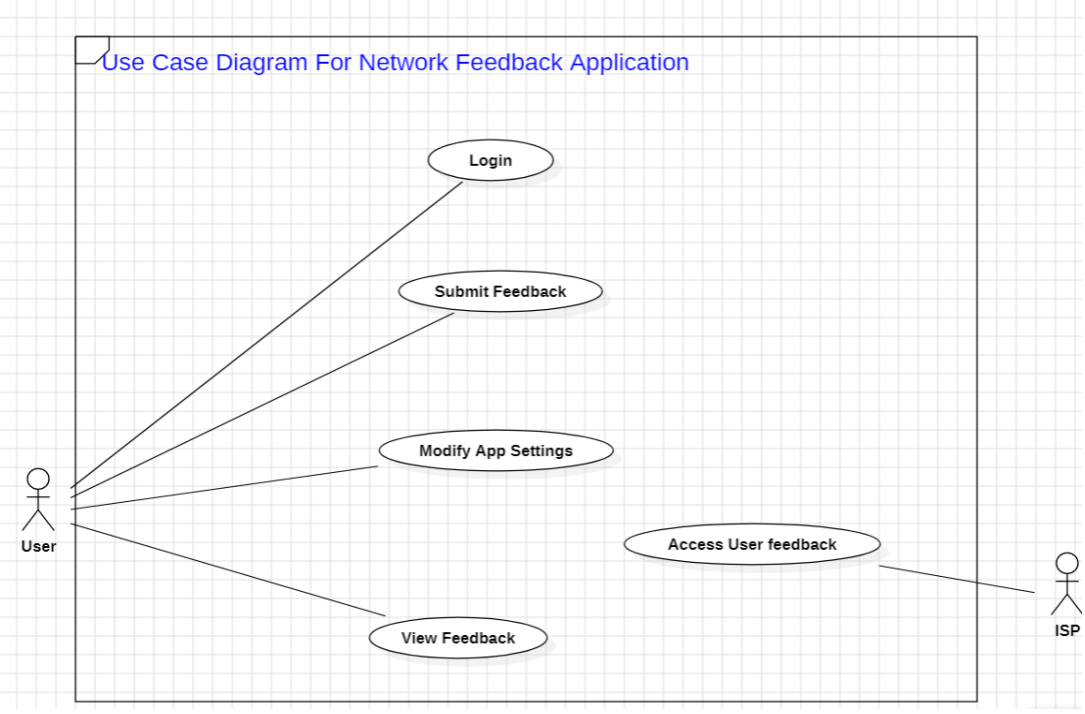


Figure 1: Use Case Diagram of Network Feedback System

2.0 Sequence Diagrams

2.1 Login/Register Sequence

SD0 outlines the login process.

- The User enters credentials, the app validates locally, and the Server queries the Database.
- If credentials are invalid, a 401 Unauthorized message is returned; otherwise, a 200 OK response is sent, and the user is redirected.

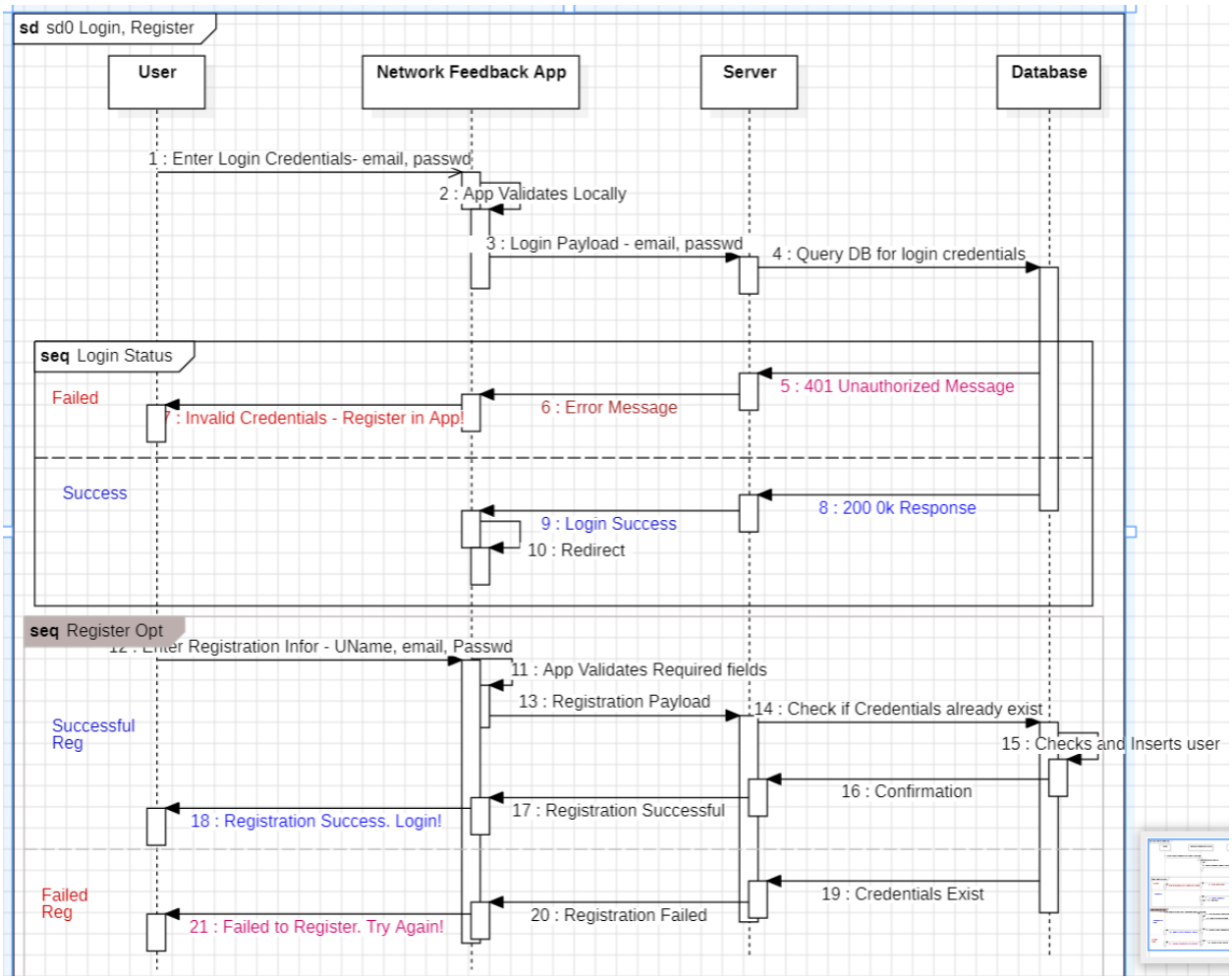


Figure 2: Register/ Login Sequence

2.2 Feedback Submission

SD1 details feedback and background data submission.

- The User enters feedback, the app collects metrics, and the Server validates the payload.
- If invalid, a 404 message is returned; if valid, data is uploaded to the Database, background collection stops, and a success response is sent.

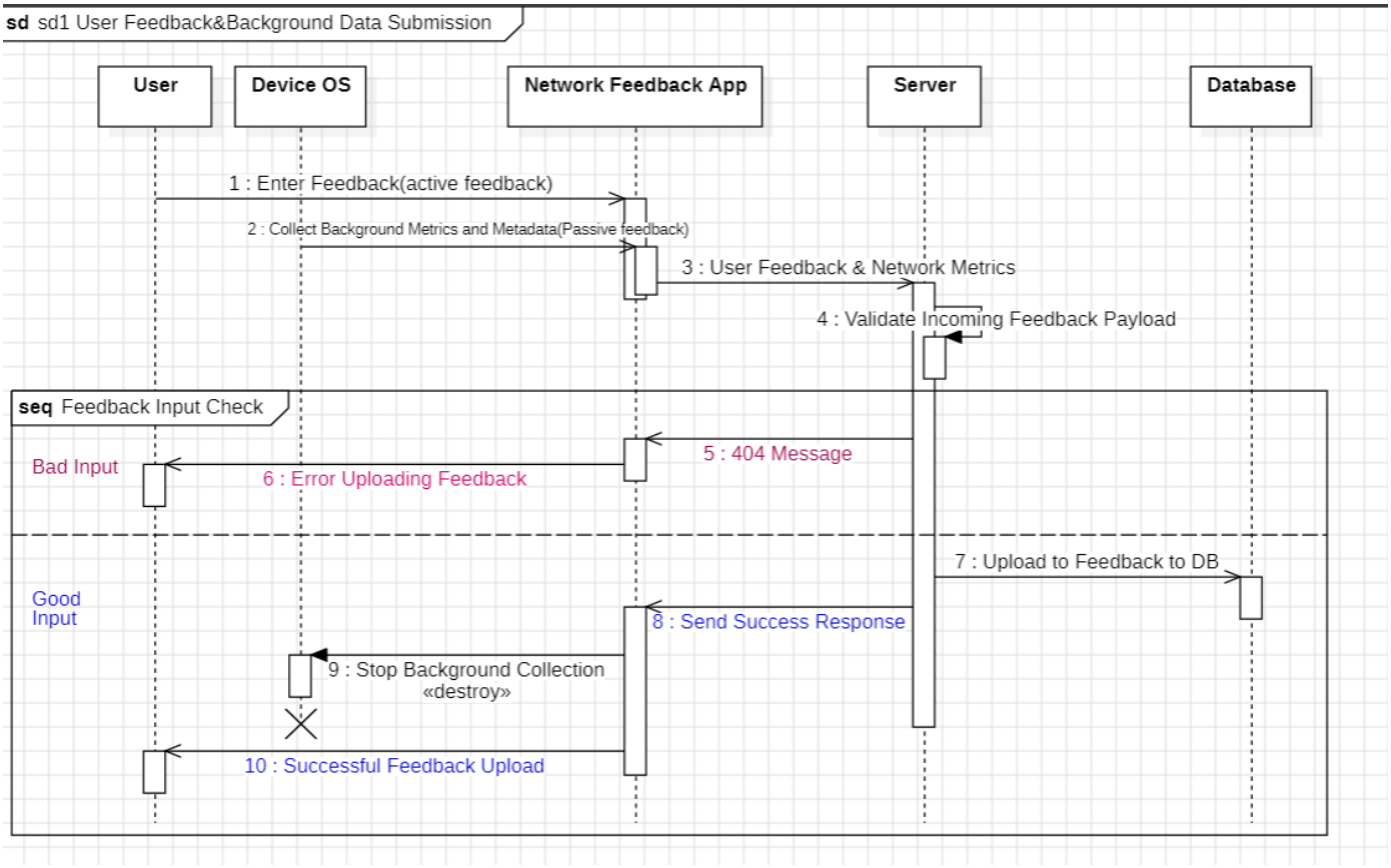


Figure 3: Feedback Submission Sequence

2.3 Offline Data Sync

SD2 covers offline data synchronization.

- When disconnected, feedback is saved locally.
- Upon reconnection, the app retrieves unsynced data, sends it to the Server, which stores it in the Database and sends a success response.

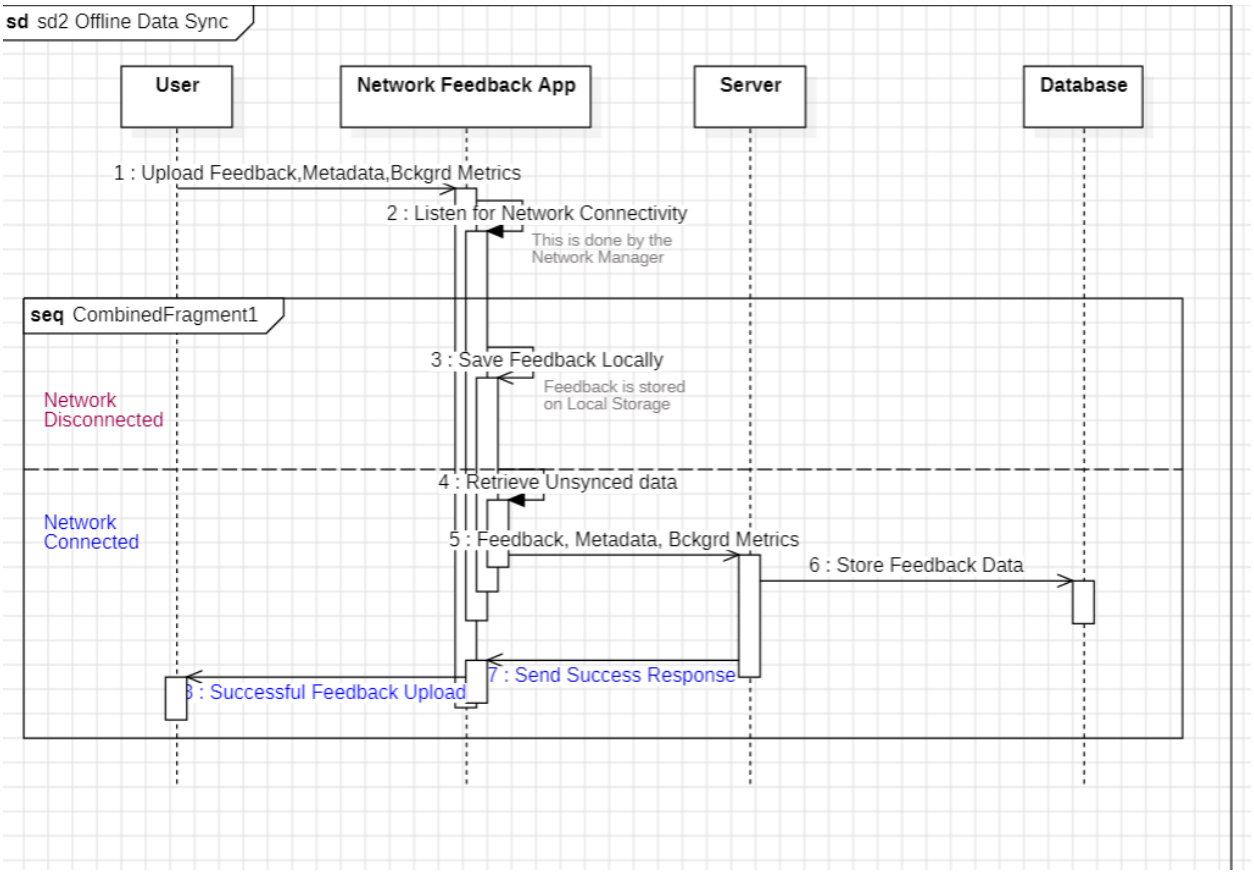
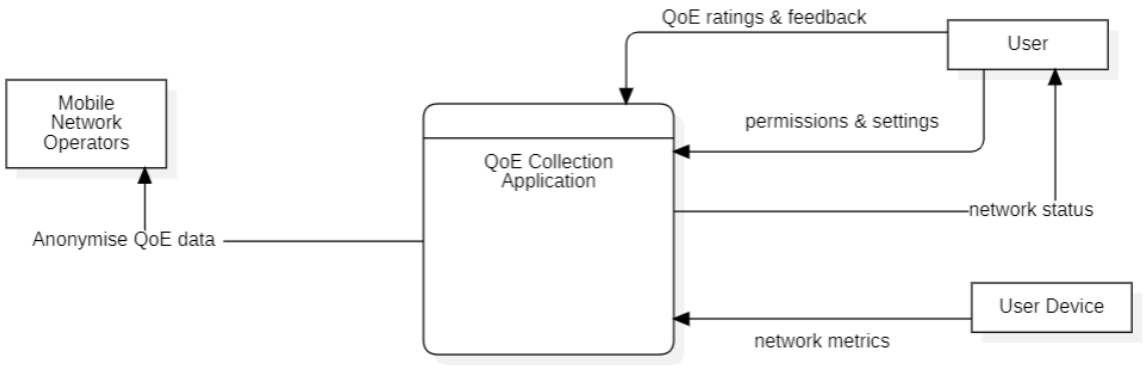


Figure 4: Offline Data Sync Sequence

3.0 Data Flow Diagram

A Data Flow Diagram (DFD) is a graphical representation of the "flow" of data through an information system, modeling its process aspects. DFDs are used to visualize how data is processed by a system in terms of inputs and outputs.

Figure 5: Context Diagram



This context diagram illustrates the data flows for a QoE (Quality of Experience) Collection Application system. Here's how the system works:

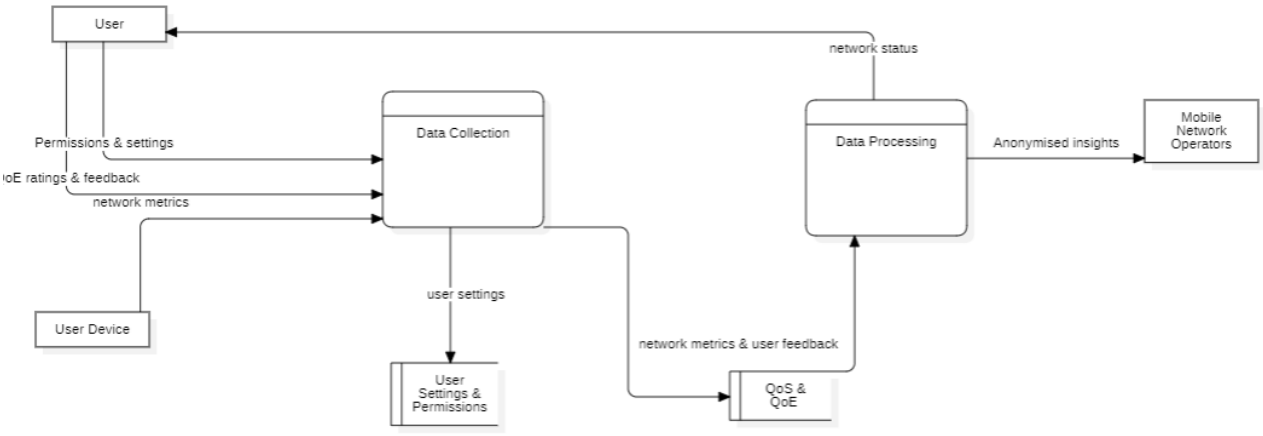
Central System: The QoE Collection Application serves as the main processing hub that gathers and manages network quality data.

Key Data Flows:

1. **From Mobile Network Operators**: The system receives anonymized QoE data from telecom providers, which likely includes network performance metrics, service quality measurements, and user experience indicators that have been stripped of personally identifiable information.

2. **To/From Users**: There's a bidirectional relationship where:
    - Users provide QoE ratings and feedback about their network experience
    - Users can configure permissions and settings for data collection
    - The system reports network status back to users
3. **From User Devices**: The application collects network metrics directly from user devices, such as connection speeds, latency, signal strength, and other technical performance indicators.

Figure 6: Data Flow - Level 1



This diagram shows a more detailed view of the QoE (Quality of Experience) system's internal processes and data flows. Here's how it works:

Core Process Flow:

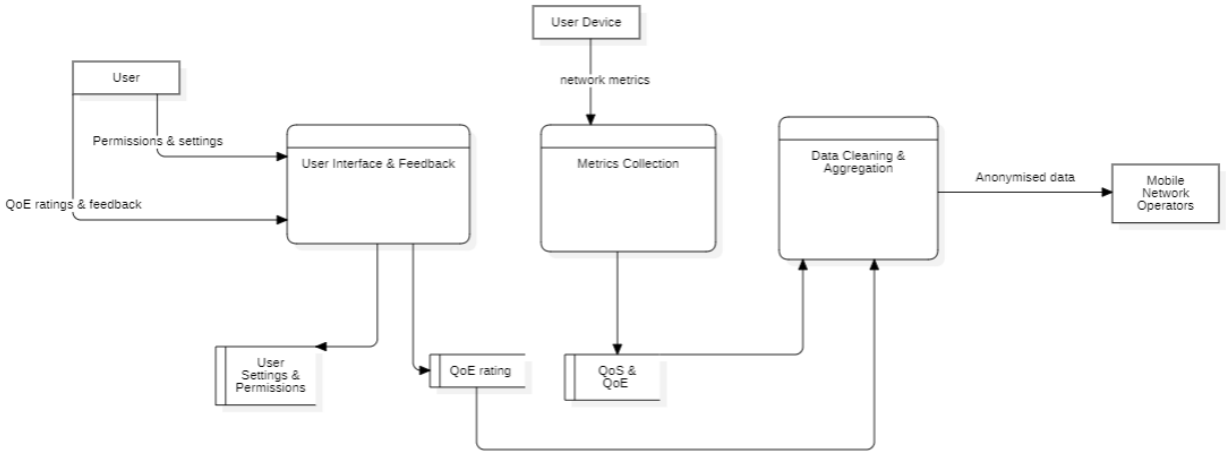
1. **Data Collection Phase**: The system gathers multiple types of input:
    - User permissions and settings configuration
    - QoE ratings and feedback from users

    - Network metrics from user devices
    - User settings are stored in a dedicated "User Settings & Permissions" component
2. **Data Processing Phase**: All collected data (network metrics and user feedback) flows into a central "Data Processing" component that:
    - Analyzes and processes the raw data
    - Generates insights from the combined information
    - Produces anonymized insights that are shared with Mobile Network Operators
3. **QoS & QoE Component**: This appears to be a quality assessment module that evaluates both:
    - QoS (Quality of Service) - technical network performance metrics
    - QoE (Quality of Experience) - user-perceived service quality

Key Features:

- **User Control**: Users maintain control over their settings and permissions throughout the process
- **Feedback Loop**: Network status information flows back to users, creating transparency
- **Privacy Protection**: Data is anonymized before being shared with network operators
- **Comprehensive Analysis**: The system combines technical metrics with subjective user feedback for a complete quality assessment

Figure 7: Data Flow Level 2



This diagram shows a more granular view of the QoE system's data processing pipeline with specific functional components. Here's the workflow:

Data Input Sources:

- **User Device**: Provides network metrics (technical measurements)
- **User**: Provides permissions/settings and QoE ratings/feedback through the interface

Core Processing Components:

1. **User Interface & Feedback**: Acts as the primary user interaction point, handling:
    - User permissions and settings configuration
    - Collection of QoE ratings and feedback
    - Management of user settings and permissions (stored separately)
2. **Metrics Collection**: Processes the technical network metrics from user devices and generates QoS (Quality of Service) and QoE (Quality of Experience) assessments
3. **Data Cleaning & Aggregation**: The central processing hub that:
    - Receives QoE ratings from the user interface
    - Receives QoS & QoE data from metrics collection
    - Cleans, processes, and aggregates all data

- Produces anonymized data for external sharing

Output:

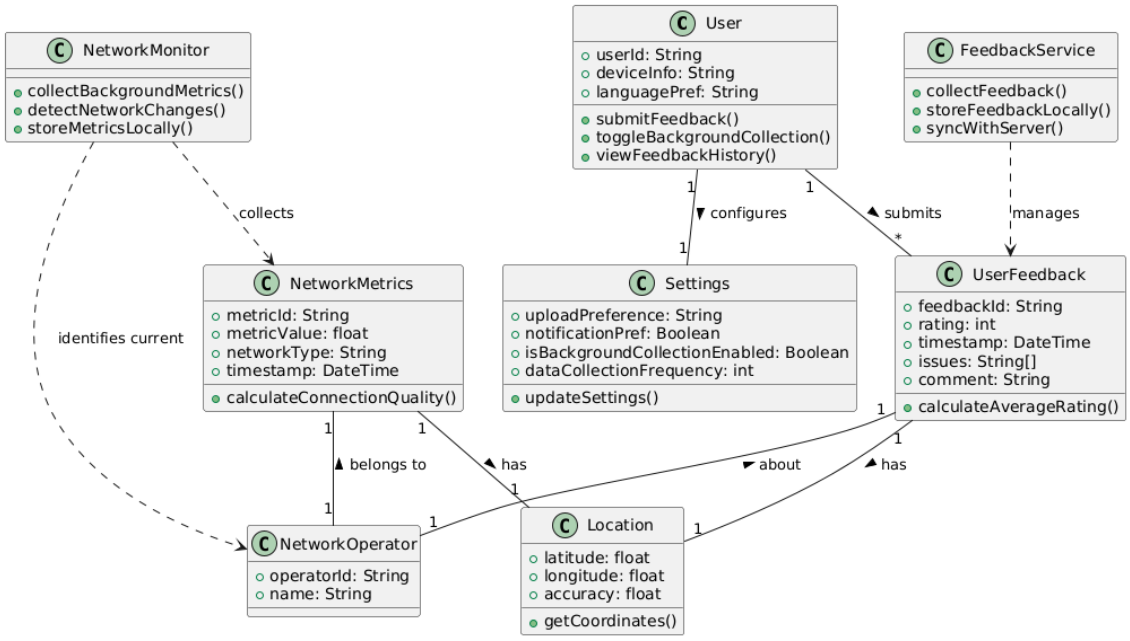
- **Mobile Network Operators** receive anonymized insights derived from the processed data

4.0 Class Diagram

4.1 Class Diagram Explanation: QoE Collection Application

This class diagram illustrates the static structure of the QoE Collection Application system. It defines key entities (classes), their responsibilities, attributes, methods, and the relationships between them. This blueprint clarifies how the system organizes and handles user interactions, data collection, and feedback management.

Figure 8: Class Diagram



1. User

- **Purpose**: Represents an individual using the application.
- **Attributes**:
    - userId: Unique identifier for the user.
    - deviceInfo: Details about the user's device.
    - languagePref: User's preferred language.
    - isBackgroundCollectionEnabled: Indicates if background data collection is allowed.

- **Methods**:
    - submitFeedback()
    - toggleBackgroundCollection()
    - viewFeedbackHistory()
- **Relationship**:
    - One User can submit multiple UserFeedback entries.
    - One User configures one Settings instance.

2. Settings

- **Purpose**: Stores user-defined application behavior settings.
- **Attributes**:
    - uploadPreference
    - notificationPref
    - dataCollectionFrequency
- **Method**:
    - updateSettings()
- **Relationship**:
    - Each User is linked to one Settings object.

3. UserFeedback

- **Purpose**: Captures subjective feedback from users about their network experience.
- **Attributes**:
    - feedbackId, rating, timestamp, issues, comment
- **Method**:
    - calculateAverageRating()
- **Relationships**:
    - Each UserFeedback:
        - Is submitted by one User
        - Is associated with one Location

        - Refers to one NetworkOperator

4. NetworkMetrics

- **Purpose**: Stores objective network performance data collected from the user's device.
- **Attributes**:
    - metricId, signalStrength, networkType, timestamp
- **Method**:
    - calculateConnectionQuality()
- **Relationships**:
    - Each NetworkMetrics:
        - Is tied to one Location
        - Belongs to one NetworkOperator

5. Location

- **Purpose**: Represents the geographical location tied to feedback or metrics.
- **Attributes**:
    - latitude, longitude, accuracy
- **Method**:
    - getCoordinates()
- **Relationship**:
    - Shared by both UserFeedback and NetworkMetrics for spatial context.

6. NetworkOperator

- **Purpose**: Models the telecom provider under analysis.
- **Attributes**:
    - operatorId, name, country
- **Relationships**:
    - Referenced by both UserFeedback and NetworkMetrics

7. FeedbackService

- **Purpose**: Service class managing user feedback lifecycle.
- **Methods**:
    - collectFeedback()
    - storeFeedbackLocally()
    - syncWithServer()
- **Dependency**:
    - Uses UserFeedback objects to perform its operations.

8. NetworkMonitor

- **Purpose**: Handles background monitoring and metric collection.
- **Methods**:
    - collectBackgroundMetrics()
    - detectNetworkChanges()
    - storeMetricsLocally()
- **Dependencies**:
    - Uses NetworkMetrics
    - Detects NetworkOperator changes

Design Highlights

- **Encapsulation**: Each class defines its data and behavior clearly.
- **Modularity**: The system is organized into manageable, testable units.
- **Relationship Clarity**: Cardinality and associations are well-defined.
- **User-Centric**: User is central, controlling feedback and settings.
- **Contextual Awareness**: Location ties user actions and metrics to real-world context.
- **Service-Oriented**: Operational logic is encapsulated in dedicated service classes.

This structure aligns well with the application's dataflow and real-world functionality, bridging the user interface and backend logic while maintaining clarity and scalability.

5.0 Deployment Diagram

A **Deployment Diagram** is a type of **UML (Unified Modeling Language)** diagram used to model the **physical deployment** of software artifacts on hardware (nodes). It shows **how software components are deployed** across various physical devices, servers, or cloud platforms.

Purpose:

- To show **where** the system's **software components** run.
- To model the **hardware environment** (devices, servers, cloud).
- To describe the **connections** and **communication protocols** between devices.
- Useful in **system architecture**, **networking**, and **mobile/cloud-based applications**.

Elements of a Deployment Diagram

1. Node

- Represents a **physical or virtual hardware device** or execution environment.
- **Examples:**
    - Mobile Phone
    - Cloud Server
    - Database Server
    - Web Browser Environment
- Nodes can be **devices** (hardware) or **execution environments** (like containers or JVMs).

2. Artifact

- A **software unit** (e.g., a file, executable, library, or app) deployed to a node.
- Represents what is **installed or running** on a node.
- Examples:
    - App.apk on a phone
    - server.js on a cloud node
    - database.db on a database server

3. Association / Communication Path

- Shows **how nodes are connected**, typically via **networks** like the internet, LAN, Bluetooth.
- Can include **protocols** like HTTPS, TCP/IP.

4. Dependency

- Shows that one artifact **depends** on another.
- Example: Mobile app depends on Firebase SDK, or Analytics depends on Cloud Firestore.

5. Execution Environment (Optional)

- A type of node that **executes a software component**.
- Example: JVM, Android OS, Node.js environment.
- Typically shown **nested inside a device node**.

Deployment Diagram for our Quality of Experience Mobile App

Nodes and their artifact

1. Node: Mobile device

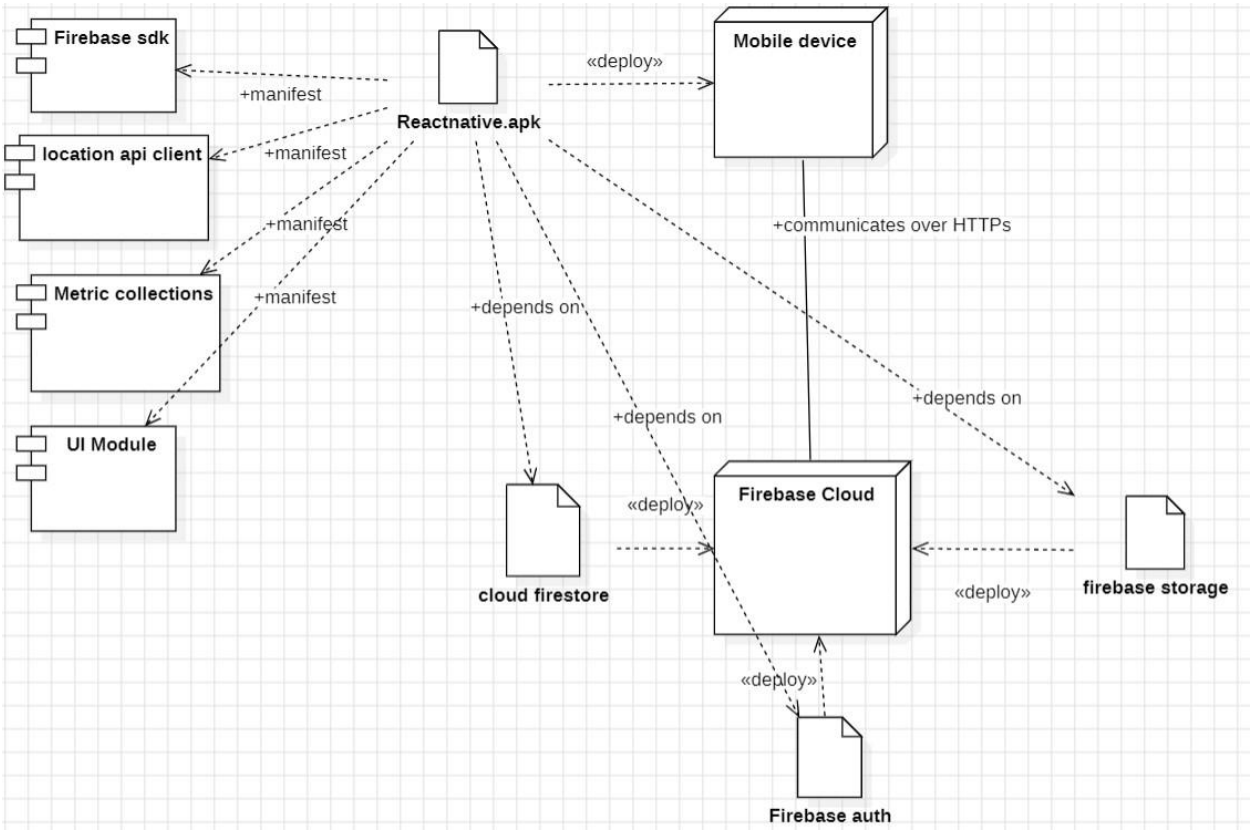
   Artifacts:

   - Reactnative.apk
   - Firebase SDK
   - Location Api Client
2. Node: Firebase cloud

   Artifacts:

   - Firebase Authentication
   - Cloud Firestore
   - Firebase storage

Figure 9: Deployment Diagram



1. Mobile Device Deployment

- **Artifact Deployed**: reactnative.apk is deployed to the Mobile Device node.
- This APK includes several internal modules and components:
    - **UI Module**: Handles user interface and user interactions.
    - **Metric Collections Module**: Gathers background metrics such as performance data, network speed, and device behavior.
    - **Location API Client**: Interfaces with the mobile device's GPS and location services.
    - **Firebase SDK**: Provides integration with Firebase services such as Authentication, Cloud Firestore, Storage, and Cloud Functions.

The Mobile Device communicates with Firebase Cloud over **secure HTTPS protocols**, ensuring that all user data and communication are encrypted.

2. Firebase Cloud Deployment

- The Firebase Cloud node hosts multiple Firebase services that the mobile application depends on:
    - **Firebase Auth**: Used for managing user authentication.
    - **Cloud Firestore**: Serves as the real-time database solution, storing and syncing app data.
    - **Firebase Storage**: Manages file uploads (e.g., user images, documents).

These services are deployed as artifacts under the Firebase Cloud node, and they **serve requests** initiated from the mobile device via the Firebase SDK.

3. Dependencies and Modularity

- The internal structure of reactnative.apk is modular:
    - Each module (UI, metric collection, location, etc.) is loosely coupled but interconnected.
    - This modular design enhances **maintainability** and **scalability**, allowing components to be independently upgraded or reused.
- The app **depends on** external services like Firebase, but encapsulates core logic within its local modules.

4. Architectural Qualities Highlighted

- **Separation of Concerns**: Functional boundaries are clearly defined between UI, metrics, and service interaction layers.
- **Cloud-Centric Architecture**: The use of Firebase reduces backend complexity and speeds up development by offloading authentication, storage, and real-time data sync to managed services.
- **Extensibility**: Additional features such as push notifications or performance monitoring can easily be added via Firebase or other SDKs.

This deployment diagram effectively demonstrates how a React Native mobile application leverages modular design and cloud services to deliver a robust, scalable, and maintainable

system. It shows not only deployment and communication but also the architectural intent behind component integration and service dependency.

Conclusion

This system design documentation successfully translates the requirements established in Task 3 into a comprehensive architectural blueprint for the QoE Collection Application. Through systematic application of UML modeling techniques, we have created a robust foundation that addresses all identified functional and non-functional requirements while maintaining clear traceability to our MVP objectives.

The modeling process has validated several critical aspects of our proposed solution:

Architectural Soundness: The Use Case and Sequence Diagrams demonstrate that our system design effectively supports all primary user workflows, from registration and feedback submission to offline synchronization. The behavioral models confirm that our architecture can handle the complex interactions between users, mobile devices, and backend services while maintaining data integrity and user privacy.

Technical Feasibility: The Class and Deployment Diagrams validate our technical stack decisions, confirming that the React Native and Firebase architecture can support the modular, scalable system required by our requirements. The clear separation of concerns evident in our class structure ensures maintainability and extensibility for future enhancements identified in our post-MVP roadmap.

Data Flow Integrity: The comprehensive Data Flow Diagrams at multiple abstraction levels verify that our system properly handles the critical path from data collection through processing to anonymized insights delivery. This modeling has confirmed that our privacy-first approach and offline-capable design align with both user needs and regulatory requirements.

Deployment Readiness: The Deployment Diagrams demonstrate that our proposed architecture can scale to handle the projected user base of 50,000-100,000+ users while maintaining the performance characteristics required by our non-functional requirements.

The systematic modeling approach has also revealed design strengths that will benefit long-term project success. The modular architecture supports incremental development and testing,

enabling our team to deliver a functional MVP while building toward the comprehensive feature set outlined in our future work. The clear separation between client-side and server-side responsibilities facilitates parallel development and reduces integration risks.

Most importantly, this design documentation establishes a solid foundation for the implementation phase. Each diagram provides specific guidance for development teams, from detailed component interfaces in the Class Diagrams to precise data flow specifications that inform API design. The models collectively ensure that implementation efforts remain aligned with user needs and business objectives identified in our requirement analysis.

Moving forward, these models will serve as living documentation that guides not only initial development but also quality assurance, user acceptance testing, and future maintenance activities. The comprehensive nature of this system design positions the QoE Collection Application for successful delivery and long-term evolution in response to changing user needs and technological opportunities.

The transition from requirement analysis to system design represents a critical milestone in our development process. With these architectural specifications in place, the project is well-positioned to proceed to the implementation phase with confidence in the technical approach and clear understanding of the system's intended behavior and structure.