System Architecture

Project Name: KU Parking App
Team Members: Sabeen Ahmad, Sriya Annem, Tanushri Sakaray, Samantha Adorno, Kaden Huber, Anna Ross
Project Synopsis: An interactive campus parking tracker showing real-time lot capacity and permit restrictions through a responsive map interface.

1. Overview

The KU Smart Parking App is a mobile application designed to provide real-time parking availability for several parking lots on the University of Kansas campus. The system integrates frontend, backend, and data processing components to deliver up-to-date and historical parking analytics to users. While the current implementation relies on mocked parking data, future iterations will integrate machine learning and computer vision to automatically detect vehicle occupancy using cameras and sensors. The system's primary objective is to help students, faculty, and visitors make informed parking decisions by presenting intuitive visualizations of parking lot occupancy, permit restrictions, and time-based traffic patterns.

2. Architectural Style and Rationale

The architecture follows a client–server model combined with cloud-based backend services. The client, built with React Native using the Expo framework, provides an interactive user interface that runs seamlessly on both iOS and Android devices. The backend is powered by Firebase, which offers real-time database capabilities, authentication, and serverless cloud functions. This architecture was selected for its scalability, ease of deployment, and compatibility with mobile environments.

Firebase eliminates the need for managing physical servers, allowing developers to focus on application logic and data modeling. The use of Firestore, Firebase's NoSQL database, enables real-time updates so all connected clients are updated automatically without requiring manual refreshes. This architecture promotes modularity, real-time responsiveness, and maintainability, which are critical for a mobile application designed to evolve into a data-driven system supported by machine learning.

3. System Components

3.1 Frontend (Client Layer)

The frontend is implemented using React Native with the Expo toolkit, enabling cross-platform development and deployment. The primary responsibilities of the frontend include data visualization, user interaction, and navigation between views. It connects to Firebase via secure API calls to fetch parking data and user-submitted reports.

Key Components:

1. Landing Page (Map View):
   The entry point of the app, displaying a map of the KU campus using react-native-maps. Each parking lot is represented by an interactive marker. Users can tap on a marker to view details or use the search bar to locate lots based on filters such as permit type (red, yellow, green), occupancy rate, or distance.

2. Lot Details Page:
   Upon selecting a lot, users are navigated to a detailed view showing:

   - Total and available spaces
   - Last updated timestamp
   - A progress bar visualizing occupancy percentage
   - A bar chart of hourly activity resembling Google Maps' "busy hours" feature
   - Permit types allowed in that lot

3. The data is fetched in real time using Firestore's onSnapshot listener, which updates the view whenever the backend data changes.

4. Navigation Menu:
   A collapsible hamburger menu (or bottom navigation bar) provides quick access to:

   - Submit Bug Report: Form allowing users to report issues stored in the Firestore /reports collection.
   - Theme Toggle: Enables switching between light and dark modes.
   - Calendar of Events: Displays campus events pulled from the /events collection, with pop-up alerts if parking restrictions apply today.

The frontend also maintains a consistent global state using React's Context API or Redux for theming, user preferences, and data caching.

3.2 Backend (Cloud Layer)

The backend is built using Firebase Cloud Services, consisting primarily of Firestore, Cloud Functions, and Firebase Hosting (for potential admin dashboards).

Each parking lot document stores both static data (name, total spots) and dynamic data (available spots, last updated time). The **busy_hours** field is an aggregated record showing average lot usage per hour, enabling the display of historical trends.

Firebase Cloud Functions:

Cloud Functions are serverless scripts triggered by events such as new data uploads. In later iterations, when the ML model updates parking counts, these functions will:

1. Validate incoming data from the sensor/ML pipeline.
2. Update Firestore with the latest occupancy metrics and timestamps.
3. Trigger recalculations for the busy-hours histogram.

This architecture ensures that all updates are atomic, secure, and real-time, while offloading computational logic from the mobile client.

3.3 Machine Learning Integration (Future Component)

Future versions of the system will employ an ML model that detects vehicle movements using camera feeds. The sensor (e.g., Raspberry Pi with a mounted camera) will capture parking lot footage and run a lightweight object detection algorithm (e.g., TensorFlow Lite + OpenCV) to determine when vehicles enter or exit. This data will be sent to Firebase via a REST endpoint or MQTT message broker.

Each time a car is detected entering or leaving, the sensor node updates the Firestore record for that lot.

4. Data Flow

1. **User Interaction:** A user opens the app and selects a parking lot from the map.
2. **Data Retrieval:** The client fetches real-time lot data from Firestore via secure API calls.
3. **Rendering:** The React components render updated availability and visual charts.

4. **Sensor Update (Future):** The camera sensor detects car activity and posts data to Firebase.

5. **Database Update:** Firestore updates trigger Cloud Functions to recalculate statistics.

6. **Real-Time Sync:** Updated data automatically propagates to all connected clients through Firestore's real-time listener mechanism.

5. Non-Functional Requirements

- **Scalability:** Firebase's serverless architecture allows for handling additional parking lots without architectural refactoring.

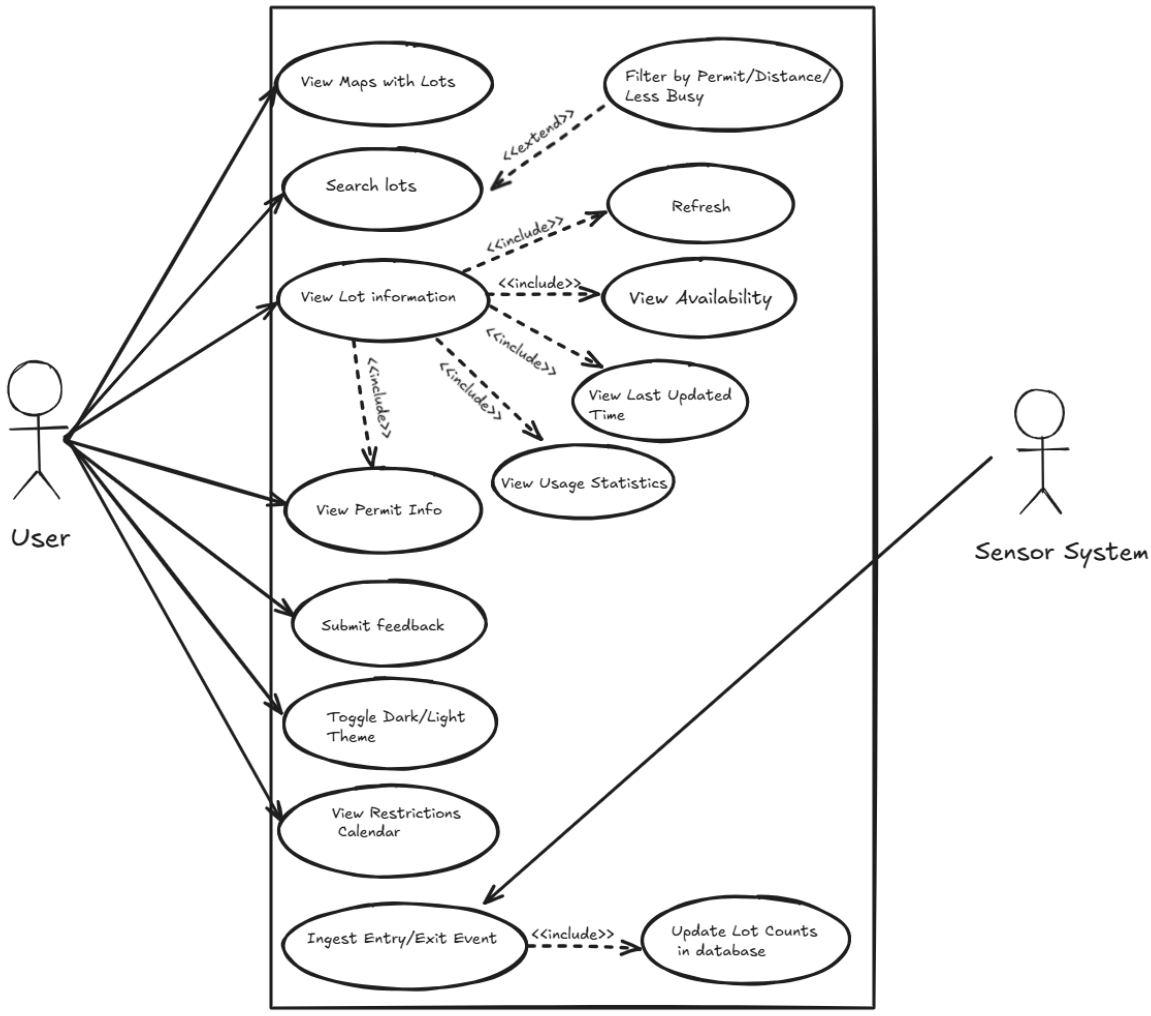
- **Reliability:** Real-time database replication ensures minimal downtime and consistent data delivery.

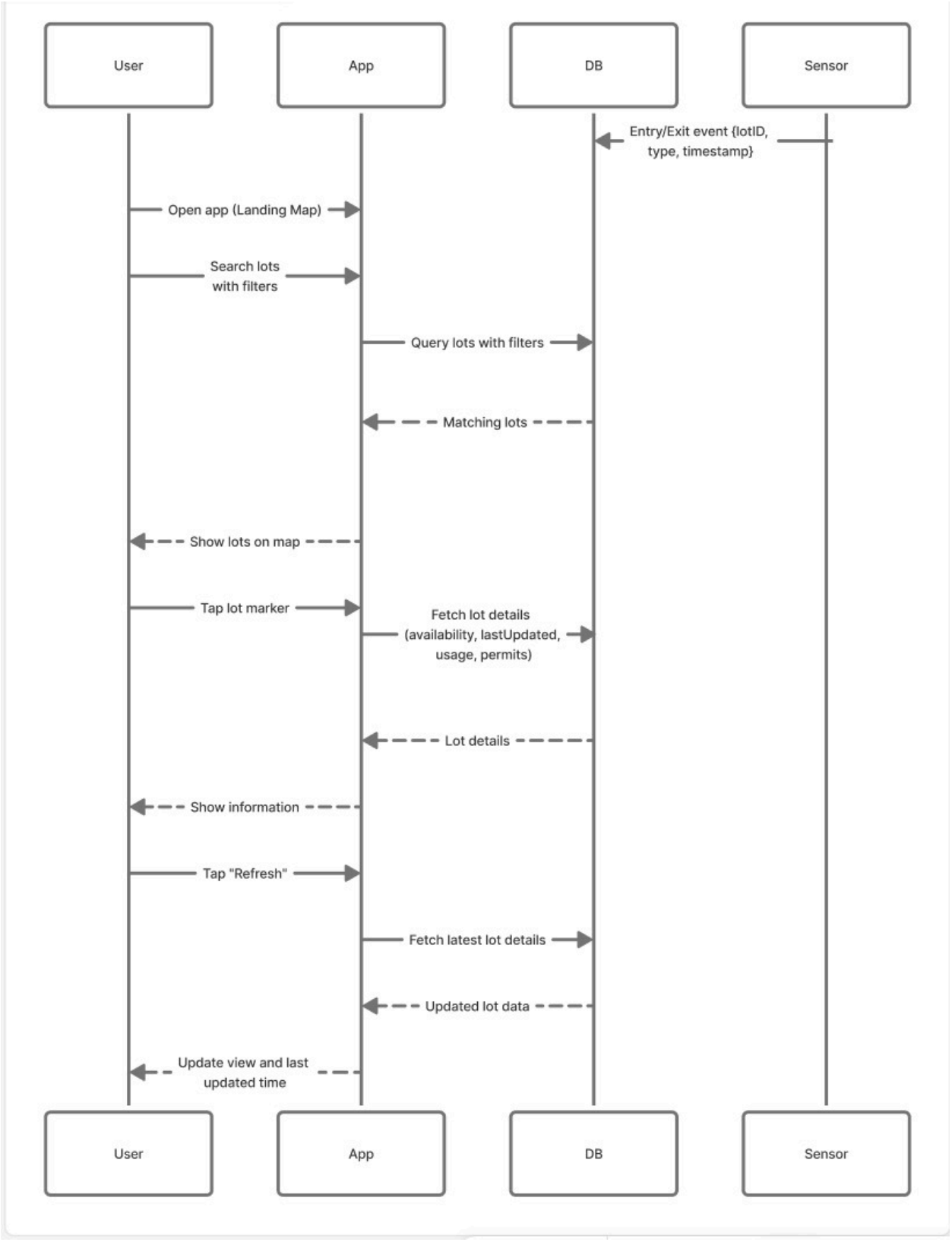
- **Performance:** React Native ensures smooth UI transitions and efficient rendering even with live data updates.

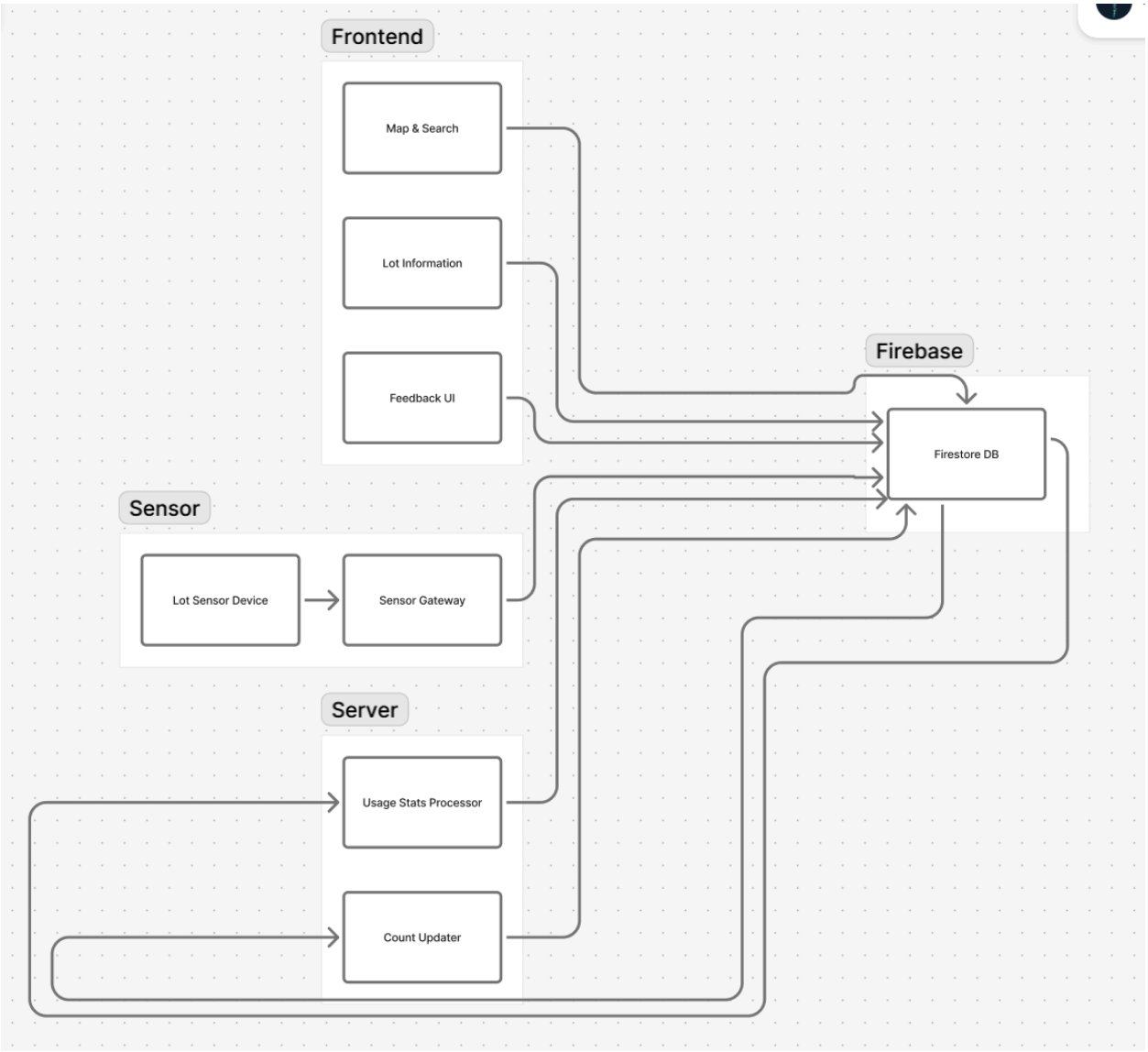
- **Security:** All Firebase endpoints are secured with authentication and Firestore security rules, ensuring only verified sensors and users can modify data.

- **Maintainability:** Modular structure with clear separation of concerns between UI, backend, and ML components simplifies updates and testing.

UML Diagrams


User
View Maps with Lots
Filter by Permit/Distance/ Less Busy
<<extend>>
Search lots
Refresh
<<include>>
View Lot information
<<include>>
View Availability
<<include>>
<<include>>
<<include>>
View Last Updated Time
View Usage Statistics
View Permit Info
Submit feedback
Toggle Dark/Light Theme
View Restrictions Calendar
Ingest Entry/Exit Event
<<include>>
Update Lot Counts in database
Sensor System


User
App
DB
Sensor
Entry/Exit event {lotID, type, timestamp}
Open app (Landing Map)
Search lots with filters
Query lots with filters
Matching lots
Show lots on map
Tap lot marker
Fetch lot details (availability, lastUpdated, usage, permits)
Lot details
Show information
Tap "Refresh"
Fetch latest lot details
Updated lot data
Update view and last updated time
User
App
DB
Sensor


Frontend
Map & Search
Lot Information
Feedback UI
Firebase
Firestore DB
Sensor
Lot Sensor Device
Sensor Gateway
Server
Usage Stats Processor
Count Updater