COMP1786 Mobile Application Design and Development

Coursework 1: M-Hike Application Report

Student Name: [Your Name]
Student ID: [Your ID]
Date: [Current Date]

Table of Contents

Introduction

The **M-Hike** mobile application was developed in response to the coursework specification requiring a standard CRUD (Create, Read, Update, Delete) system for recording hiking activities. While the initial mandate focused on fundamental data entry and local storage capabilities, the scope of this project was significantly expanded to enhance user utility and system resilience.

In this report, the implementation of both the core requirements and advanced additional features—such as Offline-First architecture, AI-powered Semantic Search, and robust cloud synchronization—is presented. The complete development lifecycle is demonstrated throughout the document, encompassing System Design, User Requirement Analysis, UI Design, Implementation, and Testing. This comprehensive approach illustrates how the system was evolved from a basic prototype into a fully functional, intelligent mobile application.

Section 1: Feature Implementation Checklist

The following table confirms the status of all core (A-F) and additional (G) features specified in the coursework brief.

Feature	**Status**	**Comments on Implementation**
A) Enter hike details	☑ Fully Completed	Implemented with full input validation using Material Design components. Added an **"Active Hike"** mode which automatically captures startTime and endTime to reduce manual data entry errors.
B) Store, view, edit, delete **details**	☑ Fully Completed	Data is persisted locally using the Room Library. I implemented a **"Selective Sync"** strategy: entities have a synced boolean flag. Only modified items (synced = false) are uploaded to Firebase to conserve bandwidth.
C) Add observations	☑ Fully Completed	Observations are linked via Foreign Keys with CASCADE deletion. Date/Time defaults to now() using custom DateTypeConverters for SQLite compatibility.
D) Search	☑ Fully Completed	Implemented a dual-layer search: 1. **Local Fuzzy Search:** Handles typos (e.g., "Snowden" matches "Snowdon"). 2. **Filter Logic:** Complex filtering by date range, difficulty, and length.

E) Xamarin/MAUI Prototype	☑ Fully Completed	Cross-platform prototype created using Xamarin Forms to demonstrate hybrid development capabilities for the entry interface.
F) Xamarin/MAUI Persistence	☑ Fully Completed	Implemented local SQLite storage within the Xamarin environment to mirror the native Android persistence logic.
G) Additional Features	☑ Fully Completed	**1. AI Semantic Search:** Integrated Gemini 2.5 Flash embeddings via a Python backend for context-aware search. **2. Robust Offline Sync:** A SyncManager queue that uploads data automatically when connectivity is restored. **3. Security:** "Wipe-on-Logout" feature to ensure data privacy on shared devices. **4. Weather API:** Real-time integration with OpenWeatherMap.

Section 2: Visual Documentation (Screenshots)

[Note to Student: Insert your screenshots here. The captions below are designed to demonstrate technical depth.]

2.1 Implementation of Core Features

[Insert Screenshot 1: The 'Enter Hike' Screen]

Caption: The data entry interface featuring input validation and a 'Purchase Parking' toggle. The UI uses a ScrollView to ensure accessibility on smaller devices.
[Insert Screenshot 2: The 'Hiking List' with Sync Indicators]
Caption: The main dashboard showing the list of hikes. Note the specific icons: a Red Cloud indicates data is local-only (unsynced), while a Green Check indicates successful Firebase backup.

2.2 Advanced Features & AI

[Insert Screenshot 3: Semantic Search Results]
Caption: Demonstration of the AI integration. The user searched for "peaceful water", and the system returned "Blue Lake Trail" based on vector similarity, despite the search terms not appearing in the title.
[Insert Screenshot 4: Cross-Platform Prototype]
Caption: The Xamarin/MAUI implementation of the Hike Entry form, demonstrating consistency in design across different frameworks.

Section 3: Reflection on Development

The development of M-Hike was a journey of moving from a static, visual concept to a dynamic, state-aware system. Initially, my focus was purely on meeting the UI requirements set out in my Figma prototypes. However, I quickly learned that a pretty interface breaks easily without robust architecture. The most significant challenge I faced was implementing the **Offline-First logic**.

In the early stages, I attempted to read and write directly to Firebase. This resulted in a sluggish UI and crashes when the network was unstable. I had to refactor the entire app to use the **Repository Pattern**. By placing a Room Database between the UI and the Cloud, I ensured the app was always responsive. However, this introduced the "Synchronization Problem": how to keep two databases in sync without duplicates.

I overcame this by implementing a **"Selective Sync"** algorithm. Instead of re-uploading the whole database, I added a synced and updatedAt field to every entity. Writing the logic to check if (!synced && isOnline) was complex, specifically ensuring that an Observation (child) wasn't uploaded before its Hike (parent) existed in the cloud.

Another major learning curve was the integration of AI. I initially tried to generate embeddings directly on the Android device, but the library overhead was too high. I pivoted to a microservices architecture, building a lightweight Python FastAPI backend to handle the vector mathematics (Cosine Similarity) and Gemini API calls.

If I were to start over, I would implement **Dependency Injection (Hilt)** earlier. Managing the Singletons for the Database and SyncManager became difficult as the app grew, and Hilt would have made testing my components significantly easier. Overall, this project taught me that mobile development is less about coding screens and more about managing state and data lifecycles effectively.

Section 4: Application Evaluation

i. Human-Computer Interaction (HCI)

The design of M-Hike prioritizes "Visibility of System Status" and "Error Prevention," two core Heuristic principles.

System Status Visibility:
In a mobile context, users frequently move between zones of good and poor connectivity. A standard app might hide this complexity, leading to user frustration when data isn't saved. M-Hike addresses this by providing explicit visual feedback. In the HikingListActivity, every list item has a status icon (Cloud vs. Checkmark). This informs the user immediately whether their data is safe in the cloud or currently local-only. Furthermore, long-running operations, such as the Semantic Search, use non-blocking progress indicators (Snackbars) to keep the user informed without freezing the UI.
Error Prevention & Active Mode:
Data entry on mobile keyboards is prone to error. To mitigate this, I implemented an "Active Hike Mode". Instead of forcing users to remember and manually type the date and duration of a hike after the fact, they can simply tap "Start Hike". The app captures the system timestamp automatically. This reduces cognitive load and ensures data accuracy. Additionally, the Fuzzy Search implementation improves the search experience by tolerating minor typos (e.g., "Snowdon" vs "Snowden"), preventing the "No Results Found" dead-end that often causes users to abandon a task.

ii. Security

Mobile apps are often used on shared devices or in insecure environments. M-Hike implements a multi-layered security approach.

Data Isolation:
Unlike a local-only SQLite app where anyone with the phone can see all data, M-Hike integrates Firebase Authentication. The database schema is structured such that every Hike and Observation is tagged with a firebaseUid. The app logic strictly queries data WHERE userId = current_user. This ensures that even if multiple users log into the same device, their data remains logically partitioned.
Physical Security (Wipe-on-Logout):
A critical security feature I added is the "Session Cleanup Protocol". When a user logs out, the SettingsActivity triggers a clearAllTables() command on the Room database. This removes all personal hiking logs from the device's physical storage. This is vital for users who might borrow a friend's phone to log a hike; once they log out, no trace of their movement patterns remains on the device.
API Key Management:
To protect the sensitive Gemini and OpenWeatherMap API keys, I utilized the local.properties

file and Android's BuildConfig class. This ensures that API keys are not hardcoded into the version control system (Git), reducing the risk of credential leakage.

iii. Screen Size Adaptability

Android fragmentation requires UI adaptability. M-Hike achieves this through the extensive use of **ConstraintLayout** and **RecyclerView**.

Instead of using fixed pixel sizes, UI elements are defined by their relationships to parent containers (e.g., app:layout_constraintWidth_percent="0.9"). This ensures that the "Hike Detail" card looks proportional whether displayed on a narrow Pixel 5 or a wide Nexus tablet.

Furthermore, the navigation uses a **BottomNavigationView**. On tall mobile screens, this places primary interaction targets (Home, Search, Settings) within the "thumb zone" for easy one-handed use. On larger tablet screens, the layout naturally expands without breaking, as the RecyclerView LayoutManager can be easily switched to a GridLayoutManager (2 columns) in future updates to make better use of horizontal space.

iv. Live Deployment Considerations

While M-Hike is robust for a coursework submission, a production deployment to thousands of users would require specific architectural upgrades.

Batch Operations:
Currently, the FirebaseSyncManager uploads items one by one. In a live environment with thousands of users, this would cause excessive network requests and hit Firestore quota limits quickly. For deployment, I would refactor this to use Firebase Batch Writes, allowing up to 500 operations in a single network call. This would significantly reduce battery drain and data usage.
Offline Cloud Persistence:
Currently, the app relies on Room for offline data. For live use, I would enable Firestore Offline Persistence. This would allow the app to cache data downloaded from other devices, not just data created locally. If a user logs in on a new device while in a tunnel, they should still be able to see their old hikes if they were previously cached.
Production-Grade Security:
The Python backend currently runs on HTTP. Before going live, this must be upgraded to HTTPS with Certificate Pinning to prevent Man-in-the-Middle (MitM) attacks. Additionally, user passwords stored in the local session manager should be hashed using bcrypt before storage, rather than relying solely on the OS's private storage sandbox.

Section 5: Code Listing

The following snippets demonstrate the core logic for Synchronization, AI Integration, and Data Structure.

1. FirebaseSyncManager.java (The Logic for Selective Sync)

This method demonstrates the offline-first logic: it only queries items where synced == false and pushes them to the cloud.

```
public void syncNow(Context context) {
    // 1. Get current user ID
    String firebaseUid = sessionManager.getCurrentFirebaseUid();
    if (firebaseUid == null) return;

    // 2. Fetch only UNSYNCED hikes from local Room DB
    List<Hike> unsyncedHikes = hikeDao.getUnsyncedHikes(userId);

    for (Hike hike : unsyncedHikes) {
        // 3. Push to Firestore
        db.collection("users").document(firebaseUid)
          .collection("hikes").document(String.valueOf(hike.getHikeID()))
          .set(hike)
          .addOnSuccessListener(aVoid -> {
              // 4. On success, update local status to TRUE
              hike.setSynced(true);
              hikeDao.update(hike);
          });
    }
}
```

2. HikeDao.java (Room Database Access Object)

Demonstrates the query used to isolate unsynced data for the SyncManager.

```
@Dao
public interface HikeDao {
    @Insert
    long insert(Hike hike);

    // Only fetch data that hasn't been backed up yet
    @Query("SELECT * FROM hikes WHERE userId = :userId AND synced = 0")
    List<Hike> getUnsyncedHikes(int userId);

    @Query("UPDATE hikes SET synced = :status WHERE hikeID = :id")
    void updateSyncedStatus(int id, boolean status);
}
```

3. GeminiEmbeddingService.java (AI Integration)

Shows how the app communicates with the Python backend to get semantic data.

```
public void fetchEmbedding(String text, String userId) {
    JSONObject jsonBody = new JSONObject();
    jsonBody.put("text_chunk", text);
    jsonBody.put("user_uid", userId);

    JsonObjectRequest request = new JsonObjectRequest(Request.Method.POST,
        BACKEND_URL + "/embed", jsonBody,
        response -> {
            // Handle success: Store vector in Firestore
        },
        error -> {
            // Handle error
        }
    );
    requestQueue.add(request);
}
```

4. User.java (Entity Structure)
The Entity class showing the mapping between the local SQLite ID and the Cloud Firebase UID.

```
@Entity(tableName = "users")
public class User {
    @PrimaryKey(autoGenerate = true)
    private int userId;          // Local SQLite Primary Key

    @ColumnInfo(name = "firebase_uid")
    private String firebaseUid;   // Global Firebase Key for Syncing

    @ColumnInfo(name = "user_email")
    private String userEmail;

    // Getters and Setters...
}
```