M-Hike Mobile Application Development Report Outline

Coursework: COMP1786 Mobile Application Design and Development **Project:** M-Hike: Hiker Management App

This outline is strategically designed to maximize marks by directly addressing all required sections (Part B) and leveraging the complexity of your system's advanced features. It highlights the evolution from your **Figma Prototype** (Basic) to the **Final Android Implementation** (Improved), showcasing the technical depth of Firebase, Semantic Search, and Security.

Report Structure

1. Title Page and Table of Contents (Unmarked, but essential for professionalism)

- **Title Page:** Course Name, Module Code, Project Title (M-Hike), Student Name/ID, Date.
- **Table of Contents:** Clear listing of all sections and subsections.

2. Introduction

- **Goal:** Set the stage by defining the application's scope and sophisticated technical architecture immediately.
- **Content to Include:**
  - **Overview:** Introduce M-Hike as a comprehensive hiking management application designed with a robust **Offline-First Architecture**.
  - **Core Purpose:** Explain that it allows users to track hiking activities and observations seamlessly, regardless of internet connectivity, by utilizing a local **Room Database** as the primary source of truth and **Firebase Cloud Firestore** for backup and synchronization.
  - **Advanced Capabilities:** Highlight the integration of **Artificial Intelligence** for data discovery. Specifically, mention the use of **Gemini 2.5 Flash** for generating vector embeddings and a **Python FastAPI backend** for performing **Cosine Similarity** searches, allowing users to find hikes based on semantic meaning rather than just keywords.
  - **Tech Stack:** Briefly list the high-level tools: Android (Java 11), Room (SQLite), Firebase (Auth & Firestore), and Python (FastAPI + NumPy).

Section 1. Feature Implementation Checklist (2%)

- **Goal:** Provide the required checklist table, confirming the completion status of all core features (A-F) and the advanced feature (G).
- **Strategy for High Marks:** For the 'Comments' column, I have expanded this to specifically name-drop the algorithms and extra functions found in your documentation (Fuzzy Search, Selective Sync, etc.).

Feature	Status	Your Comments
A) Enter hike details	☑ Fully completed	Implemented input validation and standard controls. **Enhanced with "Active Hike Tracking"**: Added specific logic (`isActive` boolean, `startTime` , `endTime`) to track ongoing hikes and automatically calculate duration. Includes custom fields for `Purchase Parking Pass` and user linkage.
B) Store, view, edit, delete hikes/database	☑ Fully completed	Data stored locally using **Room Persistence Library (SQLite)**. Implemented a **"Selective Sync" Strategy**: The system tracks a `synced` boolean flag on every entity. Only modified or new items (`synced = false`) are uploaded to Firebase to save bandwidth, rather than overwriting the entire database.
C) Add observations to a hike	☑ Fully completed	Observations linked to a `hikeID` via Foreign Keys with `CASCADE` delete. Supports full CRUD. Time defaults to current date/time using custom `DateTypeConverters` .
D) Search	☑ Fully completed	Implemented a dual-layer search system. 1) **Local Fuzzy Search**: Uses a custom `SearchHelper` algorithm to handle typos (e.g., "Snowden" matches "Snowdon"). 2) **Advanced Filters**: Location, Date Range, Length, Difficulty.
E) Xamarin/MAUI prototype (Feature A)	☑ Fully completed	Cross-platform prototype developed using Xamarin/MAUI for data entry interface, demonstrating knowledge of hybrid frameworks.
F) Xamarin/MAUI persistence (Feature B)	☑ Fully completed	Implemented persistence/storage for the cross-platform prototype.
G) Additional features	☑ Fully completed	**1. AI-Powered Semantic Search:** Integrated Gemini 2.5 Flash embeddings via a Python FastAPI backend to allow natural language queries. **2. Offline-First Sync:** Robust `FirebaseSyncManager` that queues data when offline. **3. Home Dashboard:** Weather API integration (OpenWeatherMap) and hiking statistics (Total KM). **4. Security:** "Wipe-on-Logout" feature to ensure data privacy on shared devices.

Section 2. Visual Documentation & System Design (2%)

- **Goal:** Visually demonstrate the "Whole System Setup," including formal UML diagrams, User Navigation, and the evolution from Figma (Basic) to Android (Improved).
- **Strategy for High Marks:** Structure this section to tell a story: **Requirements (Use Case) -> Structure (Class) -> Design (Figma) -> Final App**.

2.1 Use Case Modeling & User Flow (The "What" and "How")

- **Use Case Diagram:**
  - *Caption:* "High-level functional requirements. Illustrates the Hiker's interaction with the Local System (CRUD operations, Active Hike Tracking) and the Cloud System (Syncing, Semantic Search via External API)."
  - *Key Actors:* Hiker, Firebase Auth, External Weather API, Gemini AI Service.
- **User Flow Diagram:** * A diagram illustrating the navigation path: `Login -> Home Dashboard -> Hiking List -> Hike Detail -> Observation`. Captions should explain the fluid navigation structure and how the "Bottom Navigation" connects the core modules.

2.2 Technical Architecture & Class Design (The "Technical Blueprint")

- **Class Diagram:**
  - *Caption:* "Entity Relationship structure. Demonstrates the One-to-Many relationships between `User` (1) → `Hike` (*) *and* `Hike` *(1)* → `Observation` (*). Highlights the Room Entity annotations (`@Entity` , `@ForeignKey`) and the hybrid ID strategy (`local_id` for internal logic vs `firebase_uid` for cloud sync)."
- **System Architecture Diagram:** * A high-level visual showing the connection between: `Android App (Room) <-> Sync Manager <-> Firebase Firestore <-> FastAPI Backend <-> Gemini API`.

2.3 Design Evolution: Figma vs. Final Implementation

- **Comparative Screenshot:** Place your **Figma Prototype** (Basic Design) side-by-side with the **Final Android Screenshot** (Improved).
  - *Caption:* "Evolution of the 'Add Hike' screen. The final version improves on the Figma prototype by adding Material Design input validation, 'Purchase Parking' toggle, and real-time syncing indicators (Red/Green status icons)."

2.4 Advanced Feature Evidence

- **Semantic Search & API Integration:**
  - Screenshot showing the "Semantic Search" toggle in `SearchActivity`.
  - Screenshot showing a result that matched by *meaning* (e.g., query "waterfall" finding a hike with "cascading water" description), proving the API integration works.
- **Syncing & Offline Status:**
  - Screenshot showing the app in "Offline Mode" (data stored locally).

- Screenshot showing the Firestore Console populated with the synced data, proving the cloud backup works.

2.5 Cross-Platform Prototype

- Screenshots of the Xamarin/MAUI implementation (Features E & F).

Section 3. Reflection on Development (Approx. 350 Words) (4%)

- **Goal:** Discuss the journey from the basic Figma concept to the complex, secure, synchronized system.

1. From Prototype to Production (Basic vs. Improved):

- Reflect on how the **Figma Prototype** served as the visual blueprint, but the actual development required significant enhancements for **HCI compliance**.
- Discuss the challenge of matching the Figma UI in Android Studio using XML and Material Components. Mention how you moved from static screens (Figma) to dynamic, state-aware screens (Android) that handle Loading, Success, and Error states (e.g., the "Syncing..." snackbars).

2. Architectural Challenges (The Hard Stuff):

- **Offline-First Logic:** Discuss the implementation of the **Selective Sync** strategy. Explain the challenge of maintaining a `synced` flag for every single Hike and Observation. The logic—"If `synced == false` AND `online == true`, then upload"—was complex to orchestrate, especially ensuring that Observations weren't uploaded before their parent Hike.
- **Design Patterns:** Highlight the specific patterns used to manage this complexity:
  - **Repository Pattern:** Used `HikeDao` and `ObservationDao` to abstract raw database queries from the UI.
  - **Singleton Pattern:** Used for `FirebaseSyncManager` to ensure only one sync process runs at a time.
  - **Observer Pattern:** Used in `RecyclerView` adapters to update the UI instantly when data changes.
- **AI Integration:** Reflect on the decision to build a separate **Python FastAPI** backend. Explain that while Firebase is great for storage, calculating **Cosine Similarity** for vector search required a dedicated server-side environment with NumPy, which drove the decision to separate the AI logic from the mobile app.

Section 4. Application Evaluation (700-1000 Words) (8%)

- **Goal:** Provide a deep, justified evaluation. Use the "Basic vs. Improved" angle to justify your high marks.

i. Human-Computer Interaction (HCI)

- **Evaluation (Active Tracking & Feedback):**

- **Active Hike Tracking:** "To enhance the user experience beyond simple data entry, I implemented an 'Active Hike' mode. Users can 'Start' and 'End' a hike, and the system automatically captures timestamps. This reduces manual input errors and provides immediate utility on the trail."
- **Fuzzy Search:** "To improve usability beyond exact matching, I implemented a **Fuzzy Search** algorithm. This ensures that if a user types 'Snowden' instead of 'Snowdon', the app still retrieves the correct hike, preventing user frustration."
- **Visual Feedback:** Evaluate the use of color-coded indicators (Red for unsynced, Green for synced) in the `HikingListActivity`. This gives users immediate confidence in the state of their data.

ii. Security (Role & API Key Enhancement)

- **Evaluation (Role Management & Data Privacy):**
  - **Role Enhancement:** Critically evaluate how the system isolates user data. "Unlike a basic local app, M-Hike uses Firebase Authentication UIDs to strictly partition data in the cloud (`users/{userId}/hikes`). A user cannot query or sync another user's private hiking logs."
  - **Single-User Cache:** Discuss the implementation of the **Logout** feature. "To secure data on shared devices, logging out triggers a `clearAllTables()` command on the Room database. This ensures no personal hiking data remains on the physical device after the session ends."
- **Evaluation (API Security):**
  - **API Key Management:** "To enhance security beyond the basic requirements, sensitive API keys (Gemini, OpenWeather) are not hardcoded in the UI layer but managed via `local.properties`."
  - **Recommendation:** "For a production environment, I recommend moving the Python Backend to HTTPS (Certificate Pinning) and hashing passwords using bcrypt before storing them locally in Room, as they are currently stored in plain text."

iii. Ability to Run on a Range of Screen Sizes

- **Evaluation:**
  - Discuss how `ConstraintLayout` and `RecyclerView` were used to ensure the UI matches the Figma design across different screen densities (Pixel 5 vs. Nexus Tablet).
  - Mention that the **Bottom Navigation** bar provides a consistent anchor point regardless of screen height.

iv. Changes for Live Deployment

- **System Setup & Scalability:**
  - **Batching:** "Currently, the sync manager uploads items one by one. For live deployment with thousands of users, this should be upgraded to **Firebase Batch Writes** to reduce network

overhead."

- **Offline Firestore:** "Enable Firestore's native offline persistence to allow direct reading from the cloud cache when the local Room database is empty."
- **Feature Roadmap:** Mention the "Coming Soon" features identified in the documentation: **GPS Tracking** (real-time location), **Social Features** (sharing hikes), and **Image Support** (uploading photos to Firebase Storage).

Section 5. Code Listing (2%)

- **Goal:** Select key code snippets.

1. `FirebaseSyncManager.java`: Show the `syncNow()` method that checks `getUnsyncedHikes()` and pushes them to Firestore.
2. `HikeDao.java`: Show the complex query for `getUnsyncedHikes()` and the `updateSyncedStatus()` method.
3. `GeminiEmbeddingService.java`: Show the code that constructs the JSON payload for the Gemini API and parses the vector response.
4. `User.java` **(Entity):** Show the class definition with the `@Entity` annotation, demonstrating the `userId` (Local PK) and `firebaseUid` (Cloud ID) duality.

Conclusion

- Summarize the project's success in bridging the gap between a visual prototype and a robust,