DESIGN REPORT

Offline Conversational Home Inventory Locator

Field	Details
Client	Sabari Nathan Anbalagan (s.n.anbalagan@utwente.nl)
Project Manager	Ivan Andonov (i.n.andonov@student.utwente.nl)
Database Engineer	Giorgos Kyriakou (g.kyriakou-1@student.utwente.nl)
AI Specialist	Saksham Birla (s.s.birla@student.utwente.nl)
Lead Developer	Lyubomir Yosifov (l.yosifov@student.utwente.nl)
UI/UX Specialist	Leonardo Iara (l.r.endoiara@student.utwente.nl)
Institution	University of Twente
Date	April 2026
Version	1.0

Table of Contents

1. Introduction

1.1 Project Background & Rationale

Modern households accumulate large quantities of personal belongings stored across multiple rooms, floors, and compartments. Over time, the precise locations of infrequently used items become difficult to recall, resulting in significant time lost to searching. This problem is compounded in households with complex storage layouts spanning attics, basements, and multiple floors.

Existing digital inventory solutions fail to address this problem due to three main reasons. First, most rely on cloud services for storage and require active internet connectivity. Second, they require structured manual input through rigid forms which creates high friction that discourages consistent use. Third, they lack natural language capabilities forcing users to adapt to the system rather than the system adapting to the user.

This project directly addresses these gaps by developing a privacy-focused, fully offline Android application that allows users to log household items and their pinpoint storage locations and retrieve that exact information through a natural language conversational chatbot interface. All data processing and storage are handled locally on the device. No data is transmitted externally at any point.

1.2 Project Objectives

- Build a standalone Android app that runs entirely on-device, with no reliance on external services.
- Store items with enough location detail to be useful, down to specific drawers or compartments, not just room names.
- Design a conversational search interface: a chatbot-style UI for natural language queries (e.g. 'Where are the winter gloves?').
- Enable intuitive data entry: allow users to register items and map them to rooms, containers, or shelves.
- Ensure total data privacy: no data transmitted to the internet; all processing is local.

1.3 Scope & Limitations

In scope:

- Native Android application with offline SLM engine
- Local relational database with hierarchical location support
- Chatbot UI for item registration and retrieval
- Photo and description attachments for items
- Household inventory sharing between users on the same device

Out of scope:

- Cloud synchronization or any internet-based features
- iOS or cross-platform support
- Multi-user authentication systems

1.4 SMART Goals

Goal	Measurable	Attainable	Relevant	Time-bound
Standalone Android app for conversational home inventory	100% of core functional requirements complete	Native mobile frameworks optimized for local performance	Eliminates frustration of losing track of possessions	By 29.03.2026
100% offline NLP engine for item querying	Successful parsing of natural language queries	Lightweight on-device NLP, no external API calls	Ensures privacy and functionality without internet	By 15.03.2026
Multi-level local database for household storage mapping	Retrieve item locations down to specific drawers	Robust local relational database architecture	Solves storage amnesia through granular location tracking	By 22.02.2026
Modern Material Design UI with intuitive data entry	Positive client sign-off on high-fidelity Figma prototype	Iterative design reviews with stakeholders	Reduces friction and encourages consistent use	By 01.03.2026

2. Requirements Specification

2.1 Stakeholders & Intended Audience

Stakeholder	Role	Interest
Sabari Nathan Anbalagan	Client	Needs a reliable, offline home inventory tool for a multi-floor household
Project Team (5 students)	Developers	Deliver a working application meeting all defined requirements
Household members	End users	Use the app to log and find items through a conversational interface
Privacy-conscious homeowners	Secondary users	Require data sovereignty and offline-only operation

2.2 Functional Requirements

2.2.1 Location Management

- **[HIGH] FR-L1 -** The system shall allow the user to define, edit, and remove hierarchical storage locations (e.g. floors, rooms, cabinets, shelves).
- **[HIGH] FR-L2 -** The system shall not allow storage spaces at the same level to share the same name (e.g. two drawers must be named 'left' and 'right').
- **[HIGH] FR-L3 -** The system shall allow the user to specify detailed item locations (e.g. floor, room, cabinet, shelf).

2.2.2 Item Management

- **[HIGH] FR-I1 -** The system shall allow multiple items to have the same name.
- **[MEDIUM] FR-I2 -** The system shall allow the user to add a text description to an item.
- **[MEDIUM] FR-I3 -** The system shall allow the user to attach one or more photos to an item.
- **[MEDIUM] FR-I4 -** The system shall allow the user to update the location or description of an item.
- **[MEDIUM] FR-I5 -** The system shall allow the user to remove items from the inventory.
- **[LOW] FR-I6 -** The system shall allow the user to view all stored items in a list.

2.2.3 Conversational Interface

- **[HIGH] FR-C1 -** The system shall provide a chatbot interface for interacting with the inventory.
- **[HIGH] FR-C2 -** The system shall allow the user to register new items through the chatbot.
- **[HIGH] FR-C3 -** The system shall allow the user to find items by chatting with the system.
- **[HIGH] FR-C4 -** The system shall return multiple results when a search query matches more than one item, clearly displaying each item's location.
- **[HIGH] FR-C5 -** The system shall request clarification when a search query is ambiguous.
- **[MEDIUM] FR-C6 -** The system shall correctly interpret natural language search queries and return associated locations.

2.3 Non-Functional Requirements

- **[HIGH] NFR-1 -** The system shall work completely offline without requiring an internet connection.
- **[MEDIUM] NFR-2 -** The system shall respond to user queries within an acceptable time frame.
- **[MEDIUM] NFR-3 -** The system shall store inventory data reliably and prevent data loss during normal operation.
- **[LOW] NFR-4 -** The system shall use a modern user interface based on Material Design principles.
- **[LOW] NFR-5 -** The system shall be usable by first-time users without prior training.

2.4 Security Requirements

- **[HIGH] SR-1 -** The system shall store all inventory data locally on the user's phone.
- **[HIGH] SR-2 -** The system shall not require user login or separate usernames.
- **[HIGH] SR-3 -** The system shall not send any inventory data to external servers.
- **[MEDIUM] SR-4 -** The system shall require explicit user confirmation before deleting or modifying inventory items.
- **[MEDIUM] SR-5 -** The system shall allow secure sharing of the inventory using a specific technology.
- **[LOW] SR-6 -** The system shall protect locally stored photos and descriptions from unauthorized access by other applications.

2.5 Constraints

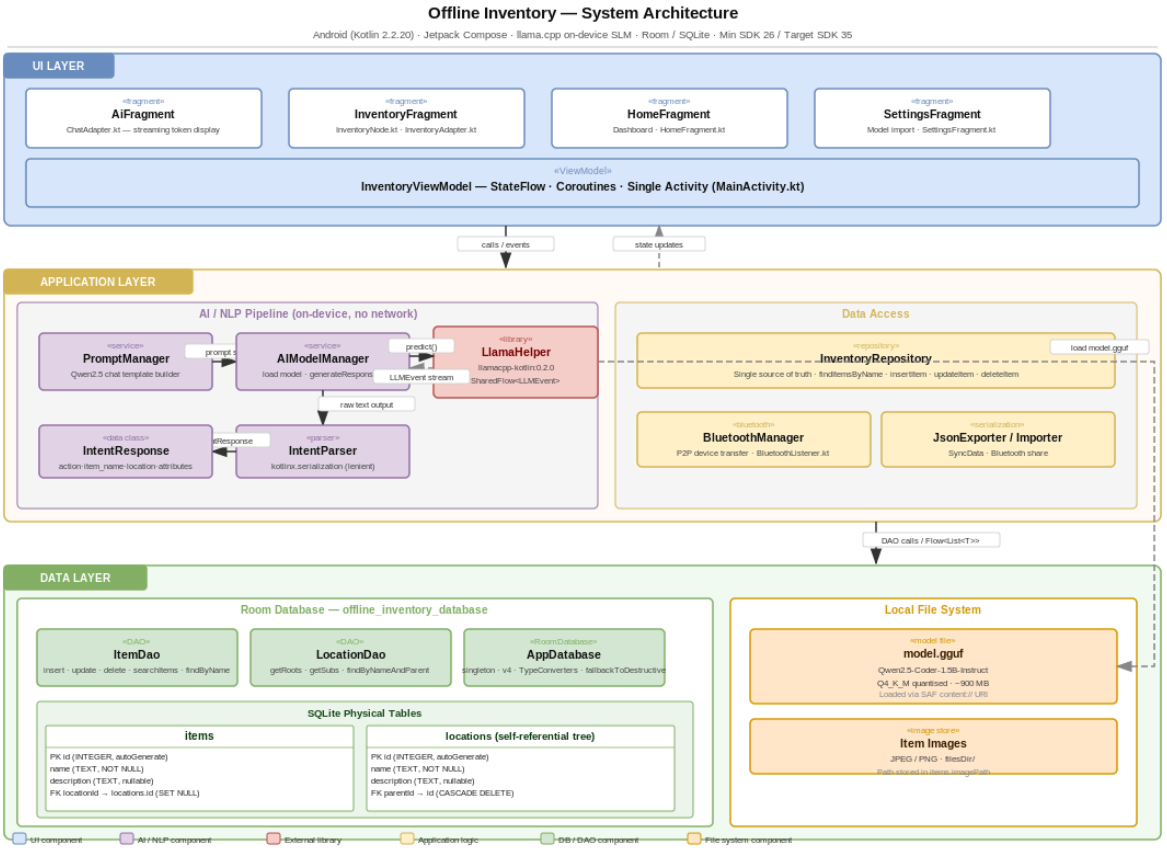
- The application must function entirely offline. No network calls permitted during normal operation.
- The target platform is Android
- The on-device SLM must be optimized for low-end mobile hardware.
- All data storage must use local mechanisms only.

3. Global Design

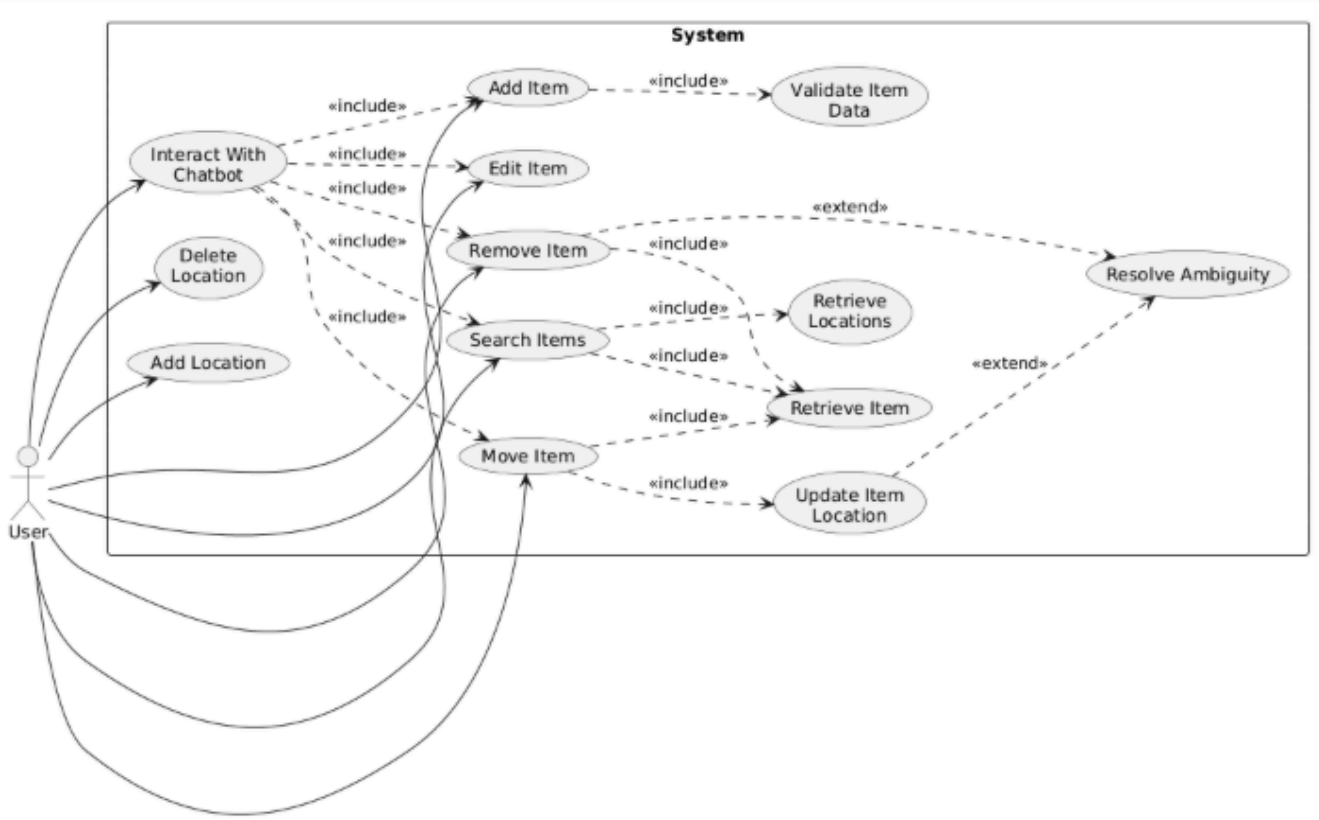
3.1 System Architecture Overview

The application follows a layered, local-first architecture consisting of three primary layers:

- Presentation Layer: Android UI built with Jetpack Compose following Material Design 3 principles. Includes the chat interface, item management screens, and location browser.
- Business Logic Layer: Handles NLP query parsing via an on-device Small Language Model (SLM), intent recognition, ambiguity resolution, and inventory operations.
- Data Layer: Local Room/SQLite database storing all item and location data. No network layer exists by design.



3.1 Use Case Diagram



The use case diagram shows how the user interacts with the chatbot-based inventory system. The system is designed to help users manage items in their home in an easy and structured way using natural language commands.

The main actor in this system is the User, who interacts with the chatbot to perform different actions. These actions include adding new items, removing items, updating item information, moving items between different storage locations, searching for items, and viewing items stored in a specific location. The user can also add or remove storage locations, such as rooms or boxes in the house.

Each use case represents a specific goal that the user wants to achieve. For example, the “Add item” use case allows the user to store a new item in the system, while the “Search item” use case helps the user find where an item is located. The “Move item” use case allows items to be transferred from one location to another, which helps keep the inventory up to date.

Some use cases are connected using relationships such as <<include>> and <<extend>>. The <<include>> relationship is used when a certain step is always required as part of a process, for example validating or saving item data. The <<extend>> relationship is used for optional behavior, such as suggesting locations or showing additional information when multiple results are found.

Overall, the use case diagram provides a clear overview of the system’s functionality and shows how the chatbot supports the user in managing their home inventory efficiently. It helps define the system scope and ensures that all main user interactions are properly identified before implementation.

3.2 Technology Stack

Component	Technology	Version	Justification
Language	Kotlin	2.2.20	Null safety, coroutine-native concurrency, full Jetpack ecosystem support, industry standard
UI Framework	Jetpack Compose + Material Design 3	Compose BOM 2024.09.03	Declarative UI, eliminates XML boilerplates, Material 3 design language
Local Database	Room (SQLite)	2.6.1	Type-safe DAOs, LiveData/Flow integration, Android native Jetpack library
SLM Engine	Qwen2.5-Coder-1.5B-Instruct (GGUF Q4_K_M) via llamacpp-kotlin	Llamacpp-kotlin 0.2.0	Runs fully on-device over ARM64, instruction-tuned for structured JSON output
AI Inference	llama.cpp (Android wrapper)	0.2.0 (Maven Central)	NEON-optimised CPU inference; no GPU, no network, no cloud dependency
Inventory Sharing	Bluetooth + JSON import/export	Android SDK 26+	Fully local P2P transfer. no Wi-Fi router or cloud intermediary required
Build System	Gradle + Android Gradle Plugin	AGP 8.9.1+	-
ABI Target	Arm64-v8a	-	Sole ABI supported by llamacpp-kotlin; targets Snapdragon 8 Gen 3 / Exynos 2400
Version Control	Git	-	Industry standard

3.3 Key Components

3.3.1 Language and Build Environment

The application is written entirely in Kotlin 2.2.20. Kotlin was selected mainly due to the fact that it's an industry standard. The developers also wanted to expand their knowledge and learn a new language. The language's null-safety type system also reduces defensive boilerplate significantly in many areas such as JSON parsing in which the model might omit optional fields in its output. The ABI filter is locked to arm64-v8a, the only architecture supported by llama.cpp Android wrapper, and the primary architecture of the Samsung phones used for development and testing which also is the target model for the app.

3.3.2 User Interface Framework

The UI is built with Jetpack Compose using the BOM 2024.09.03 pin, which ensures consistent versions across all Compose artifacts. Compose was chosen over the legacy View system mainly because Compose has first class support for Material Design 3 which satisfies NFR-4 easily and reliably.

3.3.3 Local Database Layer

The local database layer is implemented using SQLite through Android's Room persistence library. It stores the full inventory entirely on-device, which is essential for satisfying the system's offline and privacy requirements. The schema is centred around two relational entities: Location and Item. Location represents the hierarchical storage structure of the home, allowing paths such as floor → room → cabinet → shelf to be modelled through self-referential parent-child relationships. Item records are linked to specific location nodes, enabling the system to store not only item names but also associated metadata such as description, quantity, flexible attributes, and optional image data. Room was chosen because it provides type-safe database access through Kotlin entities, DAO interfaces, and repository abstraction, while still mapping directly onto a lightweight SQLite backend suitable for mobile deployment.

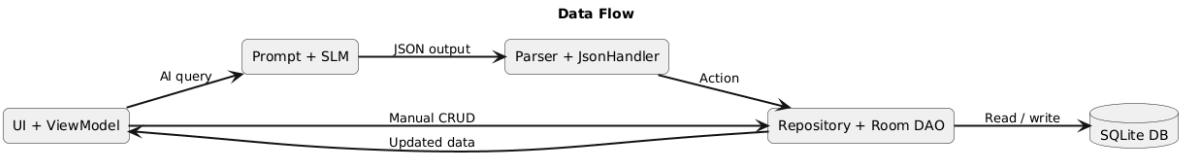
3.3.4 NLP / Small Language Model Engine

The SLM engine is the most significant component of the architecture. It is also directly tied to the project's offline constraint. After evaluating several quantized device models against the criteria defined in section 4.3.1, the team selected *Qwen2.5-Coder-1.5B-Instruct* in *Q4_K_M GGUF* quantized format. This model weighs approximately 900MB on disk and loads into under 1.5GB of RAM, which is decent for 8GB models but perfect for 12GB models. The "Coder" variant was chosen despite no actual coding task because its instruction-tuning capabilities produce reliable structured JSON outputs. This is crucial since the application's correctness depends on the model outputting a strict and accurate JSON object for every query. Alternatively, the general purpose 1B class models were tested but it outputted incorrect JSON at a higher rate than the 2B class models especially for short and/or ambiguous inputs.
The inference is performed by llama.cpp through the Maven Central Android wrapper. Llama.cpp performs quantized integer arithmetic and achieves CPU-only inference without requiring any GPU, NPU or any network connectivity. The LlamaHelper class is constructed with a MutableSharedFlow<LLMEvent> and a CoroutineScope. They output LLMEvent.Ongoing(word) tokens as they are generated and fire LLMEvent.Done on completion. The AIModelManager accumulates tokens in a StringBuilder before calling predict() and detects when the JSON object closes at which point it cancels the collection job instead of waiting for the model to finish generating past the useful output for efficiency. The inference context length is set to 2048 tokens which is enough for the prompt template plus the JSON response reducing both latency and memory pressure. The prompt itself is constructed by PromptManager using Qwen2.5 chat template format. The system prompt instructs the model to output exactly one JSON object with four keys: action, item_name, location (hierarchical array) and attributes.

3.3.5 Bluetooth Module

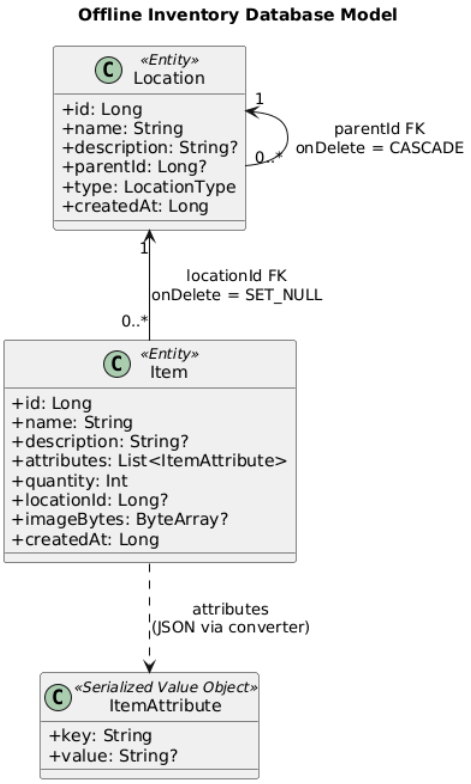
The Bluetooth module enables offline data synchronization between devices using Bluetooth Classic communication. It is based on the RFCOMM protocol, which provides a reliable stream-based connection similar to traditional sockets. A dedicated BluetoothManager class handles connection setup, data transfer, and lifecycle management, while a listener interface is used to propagate connection and data events to the UI layer. To avoid connection conflicts, a deterministic client/server selection is performed by comparing device MAC addresses. Data is exchanged in JSON format using a simple delimiter-based protocol to ensure correct message reconstruction over the stream.

3.4 Data Flow



4. Detailed Design

4.1 Database Schema



4.1.1 Entity-Relationship Overview

The database is built around two main entities: locations and items. The locations table is self-referential, allowing it to represent storage hierarchies of arbitrary depth, such as floor → room → cabinet → shelf → compartment. This structure enables the system to store and reconstruct precise location paths within the inventory.

The items table stores inventory objects and links each item to a specific entry in locations through the location_id foreign key. This ensures that every item is associated with a place in the hierarchy, while still allowing multiple items to share the same name.

To support richer item descriptions, the model also allows optional metadata such as a textual description, quantity, flexible attributes, and an optional image stored directly in the database as bytes. The attributes field is used to store flexible item-specific properties, such as colour, size, or condition, without requiring additional schema changes. This keeps all inventory data fully local and supports the project's offline-first design.

4.1.2 Key Tables

Table	Key Columns	Notes
locations	id, name, parent_id, created_at	Self-referential table used to represent the storage hierarchy. The parent_id references another row in locations, allowing nested structures such as room → cupboard → shelf. Sibling locations under the same parent must have unique names.
items	id, name, description, attributes, quantity, location_id, image_bytes, created_at	Stores inventory items linked to a location. Multiple items may share the same name. Optional fields such as description, attributes, and image support richer item representation without requiring separate lookup tables.

4.1.3 Uniqueness Constraints

A composite UNIQUE(name, parent_id) constraint on the locations table ensures that no two locations with the same parent can share the same name. This prevents ambiguous paths inside the

hierarchy, for example two shelves both called “Top Shelf” inside the same cupboard. However, identical location names are still allowed in different branches of the hierarchy.

The items table has no uniqueness constraint on name, since multiple items may legitimately share the same label, such as two “screwdriver” entries stored in different places. Each item is associated with a location through the foreign key location_id.

In addition, the database enforces referential integrity through:

- locations.parent_id → locations.id
- items.location_id → locations.id

This design supports hierarchical storage navigation while preserving flexibility for item naming and richer item metadata.

4.1.4 Hierarchical Location Modelling

The locations table is designed as a recursive structure, where each row may reference another row in the same table through parent_id. This makes it possible to represent storage hierarchies of arbitrary depth without requiring a fixed number of levels in the schema. As a result, the model can support both simple structures, such as room → shelf, and more detailed hierarchies, such as floor → room → cabinet → drawer → box.

This approach is more flexible than storing separate columns for each hierarchy level, because it avoids schema changes when users create deeper or differently structured storage paths. It also enables the application to reconstruct full breadcrumb-like location paths by traversing parent-child relationships from a leaf location back to the root.

4.1.5 Item-to-Location Association

Each item is linked to a single location through the location_id foreign key. This design ensures that every stored item has a concrete physical placement in the inventory hierarchy. Associating each item with exactly one location simplifies retrieval operations, since the system can always map an item directly to a specific node and reconstruct its full path for display.

At the same time, the schema allows multiple items to share the same name. This reflects real household usage, where identical or similarly named objects may exist in different locations, such as two chargers, multiple screwdrivers, or several storage boxes. Ambiguity is therefore resolved at the application layer by combining the item name with its resolved location path.

4.1.6 Support for Flexible Item Metadata

The items table includes optional fields such as description, quantity, attributes, and image_bytes to support richer item representation. This allows the system to store additional contextual information without forcing every item to have the same level of detail. For example, some items may only require a name and location, while others may benefit from custom attributes such as colour, size, condition, or category.

The use of an flexible attributes field avoids frequent schema modifications when new item properties are needed. This is particularly useful in a conversational inventory system, where the information provided by users may vary from one interaction to another. Storing images directly as bytes also keeps the data self-contained and consistent with the project’s offline-first privacy requirements.

4.1.7 Referential Integrity and Consistency

Referential integrity is maintained through foreign key relationships in both tables. The self-referential constraint locations.parent_id → locations.id ensures that every non-root location points to a valid parent, while items.location_id → locations.id ensures that each item is associated with a valid location in the hierarchy.

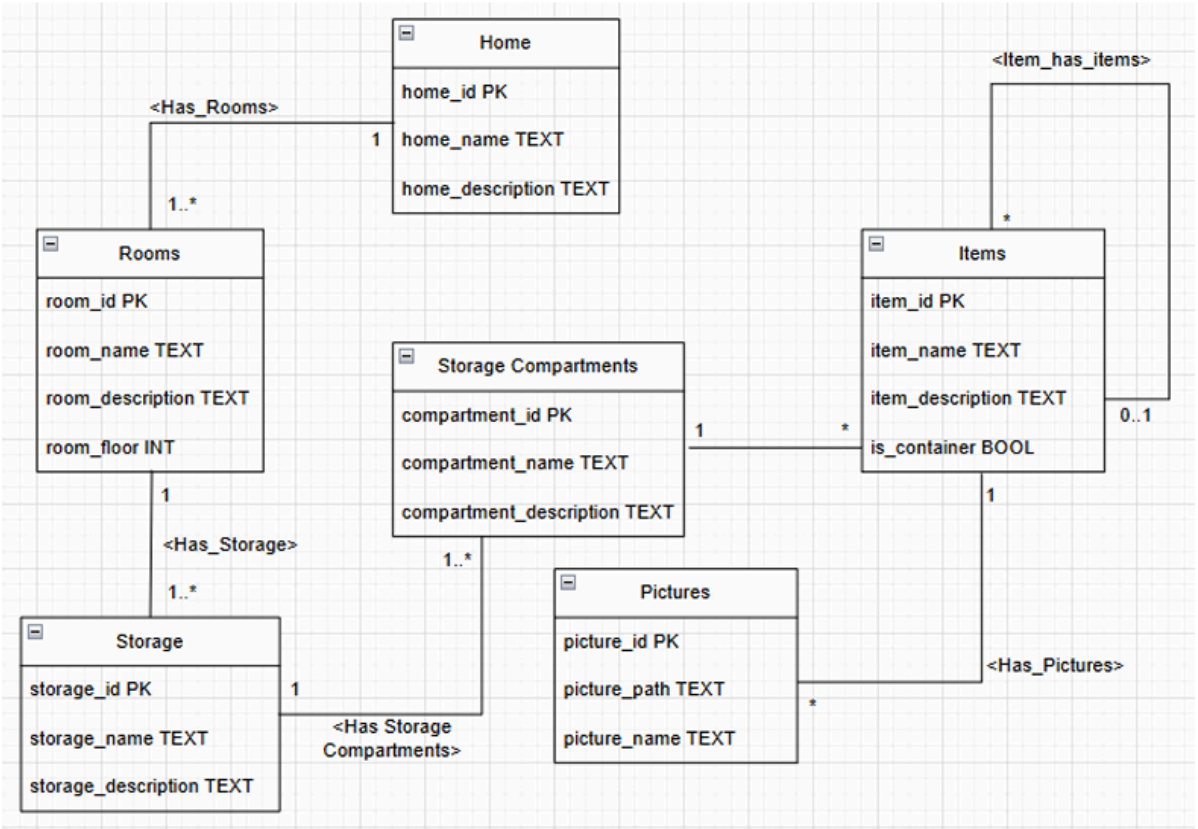
These constraints reduce the risk of orphaned records and help maintain consistency between the logical structure of the home inventory and its database representation.

4.1.8 Design Rationale

The schema was intentionally kept compact to match the project's offline and mobile-first setting. Instead of introducing many specialised tables, the design focuses on two central entities that capture the core relationship of the system: items are associated with locations. This makes the database easier to maintain, reduces query complexity, and supports efficient local execution on a mobile device.

The design also reflects the conversational nature of the application. Since users may describe locations in natural language and may refer to items with incomplete or ambiguous names, the schema must support both hierarchical lookup and duplicate item names. The chosen structure therefore prioritises flexibility in storage modelling while still enforcing enough constraints to preserve consistency.

4.1.9 Initial Database Draft



In the initial design phase, this was one of the proposed database structures. We found that this design was easy to understand and work with; however, it lacked flexibility when handling nested

locations.As shown in the design, entities such as Homes, Rooms, Storage, and Storage Compartments are organized in a structured and normalized manner, ensuring clear relationships and strong data integrity. This approach makes the system intuitive and prevents invalid relationships, such as assigning incorrect parent entities. The design also allows items to be nested within other items, supporting real-world scenarios where objects can be stored inside one another (e.g., a box within a container). This provides some level of flexibility and enables the system to handle multiple levels of depth without requiring additional tables. Overall, this hybrid design combines clarity and structure, making it efficient for standard queries, but less adaptable for more complex or deeply nested storage relationships.

4.2 Database Implementation

4.2.1 SQLite and Room Integration

The local database is implemented in SQLite and accessed through Android's Room persistence library. Room maps the schema to Kotlin entity classes, DAO interfaces, and type converters, allowing the persistence layer to integrate directly with the Android application code. The database configuration registers the Item and Location entities together with the Converters class used for non-primitive fields such as item attributes. A simplified version of the Room database configuration is provided in Appendix A.3.1.

4.2.2 Room Entities and Table Mapping

The schema described in Section 4.1 is implemented through two Room entity classes, Location and Item. These classes map the database tables directly to Kotlin data models used throughout the application. The Location entity implements the hierarchy and constraint structure defined in the schema, while the Item entity implements item storage together with optional metadata such as description, quantity, attributes, and image data. The corresponding entity definitions are provided in Appendix A.3.2 and Appendix A.3.3. At implementation level, the Item entity includes fields such as attributes, quantity, and imageBytes, which support richer item representation without requiring additional tables. The Location entity includes the fields needed for hierarchical organisation and timestamp tracking. Together, these entities provide the application with a direct and consistent mapping between the conceptual schema and the persistence layer.

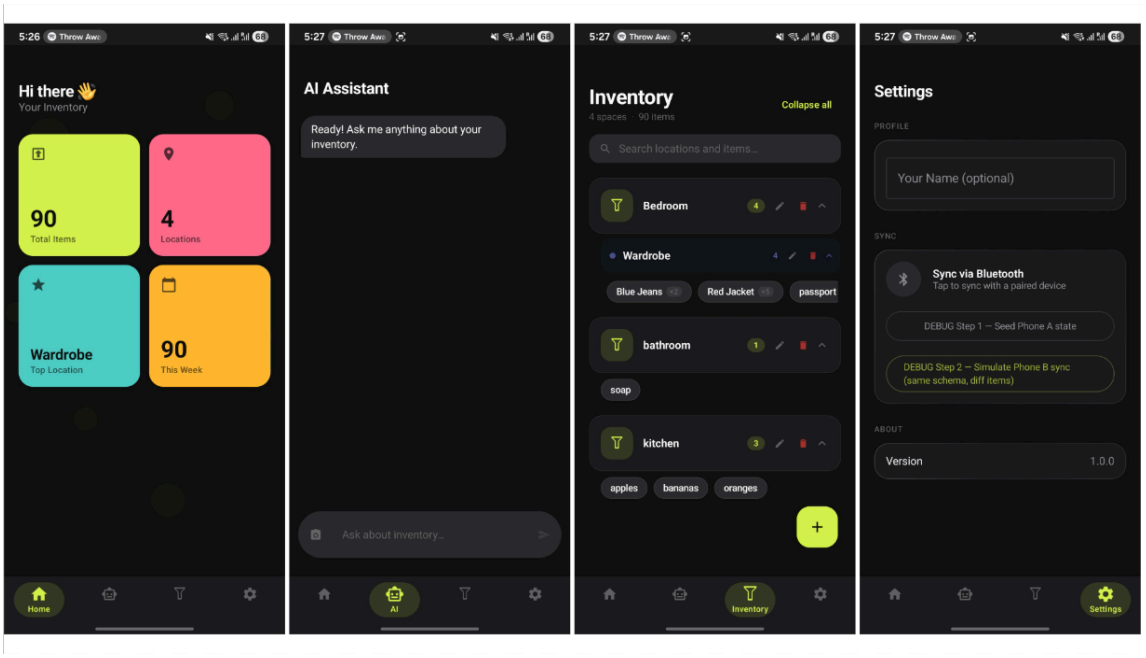
4.2.3 DAO Layer

Database access is implemented through Room DAO interfaces. LocationDao provides methods for inserting, updating, deleting, and querying locations, including methods used for parent-child traversal and name-and-parent lookups. ItemDao provides methods for item insertion, update, deletion, name-based search, and location-scoped retrieval. These DAO interfaces are wrapped by the InventoryRepository, which exposes higher-level methods to the rest of the application. This repository layer separates persistence concerns from business logic and allows components such as the JsonHandler and InventoryViewModel to interact with the database through a cleaner abstraction. Relevant DAO and repository snippets are provided in Appendix A.3.4, with the LocationDao, ItemDao, and InventoryRepository shown in Appendix A.3.4.1, Appendix A.3.4.2, and Appendix A.3.4.3, respectively.

4.2.4 Type Converters and Flexible Attributes

To support flexible item metadata, the implementation uses a Room type converter for the attributes field. This converter serialises a List<ItemAttribute> into JSON text for storage and deserialises it back into Kotlin objects when reading from the database. This allows variable item-specific properties to be stored without introducing additional schema complexity. The converter also includes defensive handling for null, blank, and malformed values by returning an empty list instead of failing. This makes the persistence layer more robust when handling incomplete or irregular data. The converter implementation is provided in Appendix A.3.5.

4.3 Conversational Interface Design



4.3.1 Chat UI Flow

The SLM screen is implemented as a full screen Jetpack Compose layout and serves as the primary interaction surface of the application. It is organized into three vertically stacked regions: a fixed title bar, a scrollable message history and a persistent input row.

Message History. Conversation turns are rendered in a LazyColumn that automatically scrolls to the most recent message after each exchange (see Appendix A.4.5). User messages are displayed on the right as chat bubbles with accent colored backgrounds. The system responses are on the left with a surface-variant background. Each bubble enters the screen with a fade-in and slide-in animation with the directions relative to the message origin (User messages slide from the right and system from the left). This provides a natural visual distinction between the two parties. Users can attach photos to the messages using the small camera action (see Appendix A.4.6) and the label “Photo attached” is prepended to the user bubble when the message is sent. While the model is processing a request, a TypingIndicator component is shown, signalling to the user that inference is in progress (see Appendix A.4.3). The underlying message data model is defined in Appendix A.4.1.

Input Row. The input pill is fixed to the bottom of the screen and remains visible at all times. It is positioned relative to the keyboard and the navigation bar. It contains three interactive elements: a camera icon button that invokes the camera to capture a picture and attach it to the pending message (see Appendix A.4.6), a multi line OutlinedTextField for typing the input, and a circular send button (see Appendix A.4.5). The send button and camera button animate their opacity and scale with respect to the interaction state. The send button is fully opaque and active only when the model is ready and when the text field is non-empty. This prevents accidental submissions and double sends.

Model Loading State. If the model file (GGUF) has not yet been imported or fails to load, the message list is replaced by a centered card that informs the user that the SLM model is not loaded and provides a “Select a model file” button. This button invokes the Android Storage file picker, allowing the user to select a .gguf file from the device’s storage. The file is then copied into the application’s private internal storage directory and loaded into the inference engine. Progress is reported as successive text messages in the chat history, including a percentage and cumulative megabyte count

during the copy phase. The three loading states are modelled as a sealed class detailed in Appendix A.4.2

4.3.2 Initial UI Design + Implementation

During the initial stages of the project, we came up with some figma wireframes as a proof of concept to present to the client and get some formal feedback on how we should implement the UI of the application. As you can see in Appendix C, these wireframes were very barebones and had no color. However, these helped us communicate with our client how we imagined the chatbot and storage interaction could look like.
We had two versions, Appendix C.1 and C.2 show one of the potential designs where the application would be completely focused on the chatbot interface. It had minimal UI elements, only the text bar and the storage display elements. Once the user focused the text bar, the background would blur and only show the messages between the user and the chatbot, otherwise those messages would not be displayed. On the other hand, the second version we came up with had the same interface as the first version but with the addition of a button next to the text bar which once clicked, would open a modal to add an item or a storage. This gave the user the option to manually insert items and storages in case the chatbot was not a preferred or viable option.
After presenting these designs to the client, he gave us overall positive remarks about the design as a whole. There were suggestions for adding colors as the grey shades were not completely appealing. And a combination of both designs was agreed upon. He claimed that the app should be chatbot focused, however, giving the choice for users to manually insert items and storages was a good idea to increase user accessibility.
Once we began with development, a minimal version of the figma wireframes was implemented in an isolated environment. Then once the base for the UI was made, all the other components were connected to it.
Later during the development process, the UI was redesigned to the current version of the finished application due to the need of more colors, and due to the new features we implemented needing specific UI elements.

4.4 SLM Module Design

4.4.1 Model Selection

The SLM engine is implemented using Qwen2.5-Coder-1.5B-Instruct, a 1.5 billion parameter instruction tuned language model developed by Alibaba Cloud, distributed in GGUF format and quantized to 4-bit int precision with the Q4_K_M quantization scheme.The resulting model is 900 MB approximately and is loaded at runtime from device's internal storage via the Android Storage Access Framework. Inference is performed on-device using the llamacpp-kotlin library which is a kotlin wrapper for the lamma.cpp inference engine. The model was selected using the following criteria:

- **Model size:** 900MB after quantization is decent. The model fits within the RAM budget of a modern mid-tier to flagship android devices.
- **Structured Output reliability:** Qwen2.5-Coder models are trained with a strong bias towards structured and code-like output, making them considerably more reliable at outputting well-formed JSON when compared to other variants. This property is important given that the application's entire intent pipeline depends on valid JSON outputs.
- **Chat templates:** The model supports the <|im_start|> / <|im_end|> chat templates natively which allows for precise role-separated prompting without any additional fine tuning needed
- **Context length:** A context window of 2048 tokens is sufficient for the whole system prompt, a single user prompt and the expected JSON output.

4.4.2 Query Processing Pipeline

The query processing pipeline transforms a raw natural language message into a confirmed database operation through six sequential stages

- **Prompt Construction:** The user's raw input string is passed to PromptManager.buildPrompt() which wraps it in the Qwen2.5 chat template. The system turn contains a detailed instruction prompt that defines the output schema, action vocabulary, location formatting rules, together with eight worked examples covering all four action types. The assistant turn is opened but left empty, prompting the model to complete it with the JSON response
- **On-Device Inference:** The formatted prompt is sent to AIModelManager.generateResponse() which calls the inference engine with LlamaPredictor.predict(). The inference engine is configured with sampling parameters chosen to favor deterministic, low creativity output which is appropriate for structured data extraction. Temperature 0.2, top-K 20, top-P 0.9, min-P 0.05 and a repetition penalty of 1.1 Output generation is capped at 128 tokens. Generation automatically stops early on any of the stop sequences. Tokens are emitted incrementally through a SharedFlow as ongoing events, which allows the UI to display a typing indicator while the inference runs (see Appendix A.4.3).
- **JSON Extraction and Parsing:** Once generation completes the IntentParse receives the raw output string. It locates the first "{" and the last "}" characters and extracts the substring between them. This process ensures that any surrounding conversational text that the model may have generated is discarded. The extracted JSON is deserialized into an IntentResponse data class. A custom FlexibleLocationSerializer handles the case where the model emits location as a bare string instead of a JSON array. This basically normalizes the representations into a List<String> before the object is returned
- **Intent Dispatch:** The parsed IntentResponse carries three fields: action (add, update, remove, retrieve), location (ordered array of hierarchy segments from broadest to most specific) and attributes (JSON object of key-value modifiers such as color, size etc). The JsonHandler.process() method inspects the action field and routes the execution to the corresponding private handler.
- **Location Resolution:** For any operation that includes a storage location, the model's location output, represented as a string array, must be translated into a concrete Location entry in the Room database before the requested action can be executed. This resolution is handled internally by the JsonHandler through a multi-stage matching process. First, the returned path is normalized by trimming whitespace and removing empty segments. The system then attempts an exact hierarchical lookup, matching each path segment against the database in parent-to-child order. If this fails, a secondary lookup is performed using the fully reversed path order to recover from cases where the model returns the hierarchy backwards. When no exact match is found, the resolver applies a hierarchy-aware fallback strategy: it searches the stored location tree for valid root-to-leaf chains whose normalized names correspond to the same set of path elements, even if the original ordering was inconsistent. If multiple valid matches are found, a tie-breaker selects the chain whose ordering most closely matches the model's original output. This design improves robustness against structurally imperfect model responses while ensuring that only valid, existing database locations are used. The corresponding Kotlin implementation of this multi-stage location resolution mechanism is provided in Appendix A.1.
- **Response Generation:** The handler returns the action value. The UI then maps each variant to a human readable confirmation string that is appended to the chat message list and rendered.

4.4.3 Ambiguity Resolution

Ambiguity resolution is applied only for operations on existing items, such as update and remove actions. Before executing such an action, the JsonHandler queries the database for all items whose name matches the item name. If there are multiple items with the same name, the full list of items

with that name are rendered in the UI and the original IntentResponse is re-ran once the user makes a selection. The UI renders each item as an inline ChoiceButton displaying the item name and its resolved location path. Selecting a button applies the original action to the chosen item specifically. A cancel button is also provided. While the choice prompt is active, the input field is disabled preventing the user from submitting a new query before the ambiguity is resolved. An example of this case can be seen in Appendix F.

4.4.4 SLM Evaluation

To verify the reliability of the SLM, a structured evaluation pipeline was implemented. The SLM's task is to perform a structured information extraction. 57 test cases were written across 10 difficulty categories, covering basic inputs, noisy and spelling error inputs, multi-attribute extraction, deep location hierarchies and even hallucination tests designed to catch cases where the model makes up information that wasn't in the input.

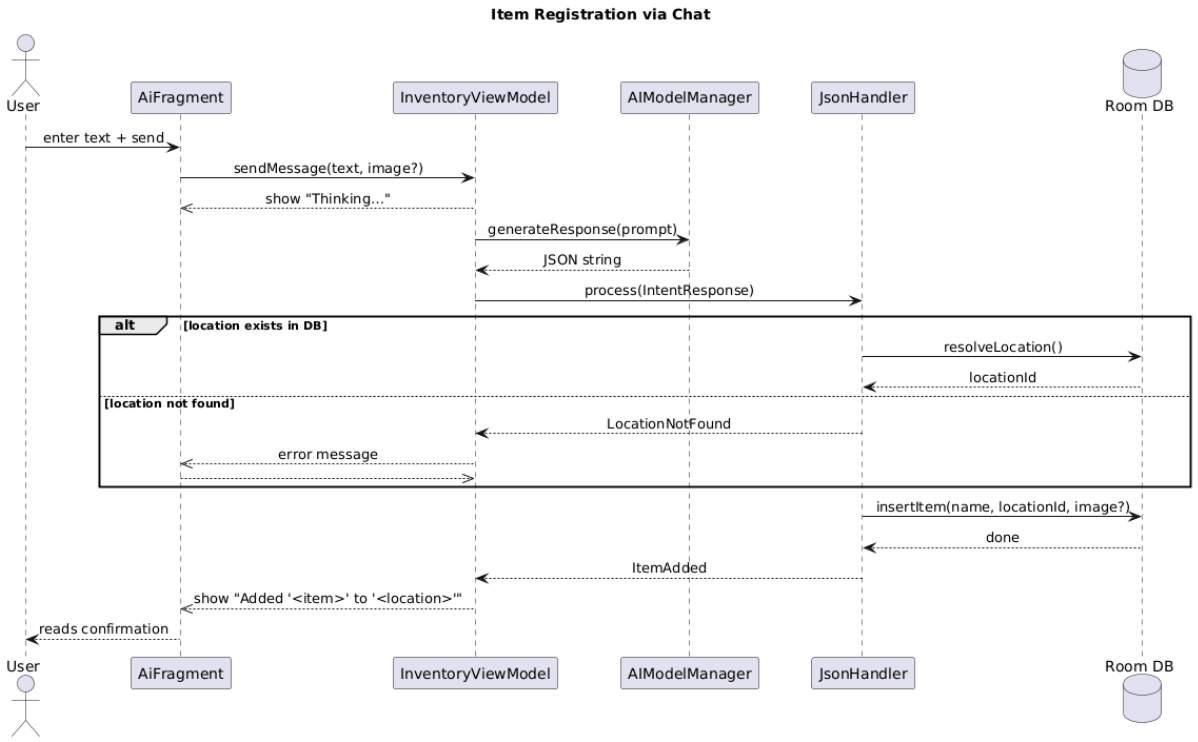
Two scoring methods were applied. Strict matching which requires every field to be exactly correct. This serves as the high bar or best case scenario. The other method is partial scoring which is more forgiving and awards points according to the field level accuracy. For this we used the following formula: score = 0.40 × action + 0.25 × itemF1 + 0.20 × locationLCS + 0.15 × attributes - penalty. The action field is scored as binary correct or incorrect. Item name scoring uses the token overlap F1 to give partial credits for lexical similarity instead of strict matching with ground truth. Location scoring uses the Longest Common Subsequence to credit partially correct hierarchies. Attribute scoring measures key coverage. A hallucination penalty of -0.20 is applied when the model hallucinates over a location and -0.15 when it hallucinates over attributes. A case is considered a partial pass if the score is >= 0.75

4.5 Data Entry Flow

4.5.1 Chat Based Registration

To register an item, the user simply types a natural language statement into the chat describing what the item is and where it is stored (eg: I put my red jacket in the bedroom closet). The app responds within seconds with a confirmation message showing the item name and the resolved location path. No menus, dropdowns or form fields are involved. If the location the user described does not exist in their inventory yet, the app tells them to add the location using the Inventory screen.

Users can optionally attach a photo before sending their message by tapping the camera button in the input row. The photo is captured in-app and sent attached to the text. If the message is recognized as an add action, the image is stored with the item automatically.



4.5.2 Form-Based Entry

A dedicated 'Add Item' screen provides structured form fields as an alternative to chat-based entry, catering to users who prefer traditional input methods.

4.6 Inventory Sharing Mechanism using Bluetooth

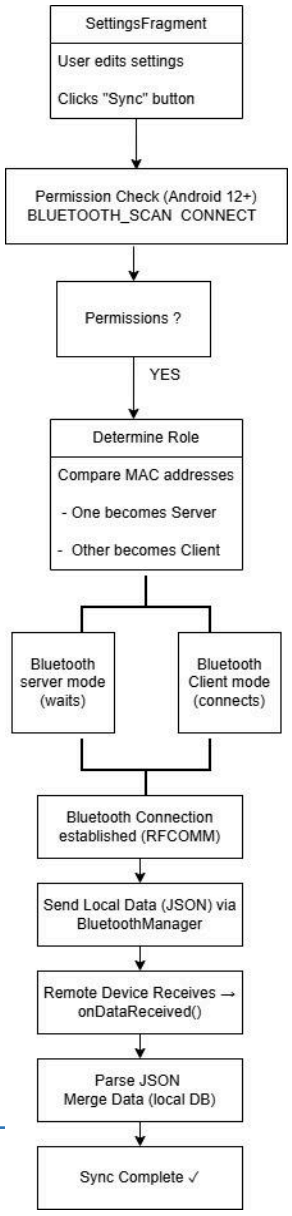
In our chatbot application, Bluetooth is used to exchange inventories (databases) across multiple household members. This functionality is implemented through the BluetoothManager class, which is responsible for setting up both server and client connections, as well as configuring the input and output streams required for data exchange. Additionally, coroutines are used to delegate specific operations to background Android threads, ensuring that the main UI thread remains responsive.

The BluetoothManager class requires a context and a listener object upon initialization. The listener facilitates communication between the manager and the SettingsFragment, which is part of the user interface. This interaction is illustrated in **Appendix A.2.1**. The class provides several functions, all of which—except for the close() function—are executed on background threads.

The connect() and startServer() functions utilize an RFCOMM socket, which acts as a serial communication channel for Bluetooth data exchange. Unlike traditional networking sockets that rely on standardized port numbers (e.g., port 80 for HTTP or port 21 for FTP), Bluetooth communication uses UUIDs to identify services. In this implementation, a UUID is used to ensure that only intended devices and services participate in the data exchange.

Bluetooth communication is handled primarily by two functions: sendData() and listenForData(). Together, these functions implement a simple protocol to reliably transmit complete messages over Bluetooth, which inherently operates as a continuous byte stream rather than a message-based system.

The sendData(data: String) function is responsible for transmitting data. It appends a special delimiter string (<<END>>) to each message to indicate its end. This is necessary because Bluetooth does not preserve message boundaries. The resulting message is then divided into smaller chunks of 512 characters to prevent buffer overflow and ensure stable transmission. Each chunk is converted into bytes and written sequentially to the Bluetooth



output stream, with a short delay between writes to avoid overwhelming the receiving device.
The listenForData() function continuously monitors the Bluetooth input stream for incoming data. It reads raw bytes into a buffer, converts them into strings, and appends them to a StringBuilder to reconstruct the full message over time. Since data may arrive in partial segments, the function accumulates these segments until the <<END>> delimiter is detected. Once detected, it signifies that a complete message has been received. The delimiter is then removed, and the reconstructed message is passed to the application through a callback (listener.onDataReceived). The buffer is subsequently cleared to prepare for the next message. The implementation of these functions is provided in **Appendix A.2.2.**
The SettingsFragment is responsible for managing user settings and coordinating Bluetooth-based synchronization between devices in the offline inventory system. It uses the BluetoothManager in combination with a listener interface to handle connection events, incoming data, errors, and disconnections. When a connection is successfully established, the application automatically transmits the local inventory data in JSON format to the connected device.
Incoming data is deserialized and merged into the local database using a custom synchronization algorithm. This algorithm compares locations and items, updates existing records, and inserts any missing entries. Furthermore, the fragment manages runtime Bluetooth permissions, handles device selection, and determines which device should act as the server or client to avoid connection conflicts. Overall, it serves as the central controller for user configuration and bidirectional offline data synchronization.
A flow diagram illustrating the complete synchronization process—from initiating the sync action to merging the databases—is presented below.

4.7 Justification of Design Choices

Decision	Alternatives Considered	Justification
Offline-first, no cloud	Cloud sync (Firebase, AWS)	Core privacy requirement; ensures functionality in signal-poor areas (basements, attics)
Hierarchical location model (self-referential table)	Flat location tags, fixed 3-level structure	Supports arbitrary depth without schema changes; mirrors real household structures
SLM over keyword/regex search	Keyword matching, structured search forms	Enables natural language queries; reduces interaction friction for non-technical users
Android native (Kotlin + Compose)	Flutter, React Native	Best offline performance; full access to Android ML APIs; no cross-platform overhead
Room over raw SQLite	Raw SQLite, Realm, ObjectBox	Type safety, LiveData integration, well-maintained Jetpack library; reduces boilerplate

5. Individual Contributions

The table below maps each team member to their primary responsibilities and report sections. Each student is expected to clearly indicate their contribution in the final submitted document.

Student	Primary Responsibilities	Report Sections
Ivan Andonov	Unit Testing, Manual Testing	Sections 1,6,7
Giorgos Kyriakou	Database schema design, fallback logic in the backend, integrating SLM with database	Sections 4.1,4.2,6.2 parts of 3.2, 4.4.3, 4.4.2
Saksham Birla	SLM pipeline, New and Modern UI, Integrating SLM with database, SLM evaluations	Initial report draft, Section 3.1, Section 3.3.4, Section 4.3.1, Section 4.4, Section 4.5, Section 7.4,
Lyubomir Yosifov	Bluetooth connection	Sections 3, 4.1.9
Leonardo Iara	Initial project and environment setup, UI design + implementation, Poster design	Sections 4.1.9, 4.2,4.4.2,4.4,7.3,8

During the middle of the development of this project, once the ai model was integrated in the system, it caused issues in running the apk of the application. This made it so that running the application on any android emulators was impossible to run. Therefore only members with android devices were able to properly run and test the application. This meant that members without android native devices had a much harder time to run and test the application, so less technical tasks had to be delegated to such team members.

6. Test Plan

6.1 Test Strategy & Scope

Testing will be conducted across five categories: unit testing (individual component testing), functional testing (requirement coverage), performance testing (SLM latency on target hardware), security/privacy testing (network isolation), and user acceptance testing (UAT with the client).

6.2 Continuous Integration Pipeline

To improve reliability during development, the project uses a Continuous Integration (CI) pipeline that automatically runs on each code push and pull request. The pipeline performs build verification, dependency resolution, and execution of automated tests, including unit tests for core logic and database-related behavior. This helps detect faults early, before changes are merged into the main branch.

Beyond test execution, the CI pipeline serves as a quality gate by preventing unverified changes from being integrated. This is particularly important in this project, where the conversational AI layer relies on deterministic backend logic and database consistency. Automated validation therefore helps maintain stability even as prompts, parsing logic, and intent handling evolve. In addition, the pipeline generates build and test artifacts that support debugging by providing developers with logs, reports, and related outputs, making failures easier to trace and resolve. (An example pipeline run and its generated artifacts are shown in Appendix E.1.)

6.3 Unit Test Cases Plan

Test ID	Class tested	Test name	Test Description	Input	Expected Output
UT-01	JsonHandler	addItem succeeds with simple single location	This test verifies that the system correctly processes an AI-generated request to add an item when a single location name is provided	JSON-style object: {action: "add_item", item: "Hammer", location: ["Shed"]}	AiActionResult.ItemAdded. This result object contains: itemName: "Hammer" locationPath: ["Shed"]
UT-02	JsonHandler	addItem succeeds with nested location path	Verify that the JsonHandler can correctly resolve a multi-level location hierarchy and link a new item to the deepest (leaf) location ID in the database	JSON-style object: { item: "Bike", location: ["Home", "Garage"] }	AiActionResult.ItemAdded. This result object contains: itemName: "Bike" locationPath: ["Home", "Garage"]
UT-03	JsonHandler	addItem returns LocationNotFound when path is broken	Verify that the JsonHandler correctly identifies and reports a failure when the AI specifies a location path that does not exist in the local database	JSON-style object:{ action: "add_item", item_name: "Drill", location: ["Tools", "Missing"]	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.LocationNotFound

Test ID	Class tested	Test name	Test Description	Input	Expected Output
				}	
UT-04	JsonHandler	addItem creates separate entries for items with duplicate names	This test verifies the idempotency and uniqueness logic of the inventory system. It ensures that if a user adds two items with the exact same name to the same location, the system treats them as distinct physical entities rather than overwriting the first entry or ignoring the second.	Two identical requests for "Screwdriver" in "Toolbox" JSON-style object:{ action: "add_item", item_name: "Screwdriver ", location: [Toolbox]}	For both inputs, the function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemAdded The system performs a lookup for the "Toolbox" location. The system executes the insertItem command twice.
UT-05	JsonHandler	addItem adds two different items to the same location	This test verifies that the system can correctly process multiple sequential requests for different items destined for the same physical location.	JSON-style object: {action: "add_item", item_name: "Wrench", location: ["Garage"]} JSON-style object: {action: "add_item", item_name: "Pliers", location: ["Garage"]}	For both inputs, the function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemAdded
UT-06	JsonHandler	remove item items	This test verifies that the system can successfully process an AI-generated request to remove a specific item from the inventory	JSON-style object: {action: "remove_item", item_name: "Old Toy"}	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemRemoved
UT-07	JsonHandler	removeItem removes an item that has a nested location	This test verifies that the system can correctly remove an item that is stored within a nested location hierarchy.	JSON-style object: {action: "remove_item", item_name: "Specific Drill"}	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemRemoved
UT-08	JsonHandler	retrieveItem builds path for match in	This test verifies that the system can successfully retrieve an item and correctly identify its location	JSON-style object: {action: "retrieve_item",	The function jsonHandler.process(response) returns a sealed class object of type

Test ID	Class tested	Test name	Test Description	Input	Expected Output
		single location	when it is stored in a single, non-nested location.	item_name: "Hammer"}	AiActionResult.ItemFound
UT-09	JsonHandler	retrieveItem nested location path for matches	This test verifies that the system can successfully retrieve an item stored within a nested location hierarchy and correctly reconstruct its location path	JSON-style object: {action: "retrieve_item", item_name: "Hammer"}	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemFound
UT-10	JsonHandler	updateItem test	This test verifies that the system can successfully update an existing item's location within the inventory	JSON-style object: {action: "update_item", item_name: "Hammer", location: ["Shed"]}	The function jsonHandler.processChosenItem(chosenId, action) returns a sealed class object of type AiActionResult.ItemUpdated
UT-11	JsonHandler	resolveLocationPath succeeds using hierarchy when the order is different	This test verifies the system's ability to auto-resolve a location hierarchy even when the input list is provided in an unconventional order (e.g., child before parent)	JSON-style object: {action: "add_item", item_name: "Bolt", location: ["Drawer", "Workshop"]}	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemAdded
UT-12	JsonHandler	resolveLocationPath uses tie-breaker for ambiguous hierarchy matches	This test verifies the system's ambiguity resolution and tie-breaking logic. It addresses a scenario where multiple locations share the same name (e.g., two different "Drawers") and ensures the handler uses the sequence of the provided input to correctly identify the specific parent-child relationship intended	JSON-style object: {action: "add_item", item_name: "Fork", location: ["Kitchen", "Drawer"]}	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemAdded
UT-13	JsonHandler	process returns Error for unknown action	This test verifies the system's robustness and error handling when receiving an invalid or unsupported command	JSON-style object: {action: "dance_with_item", item_name: "Robot"}	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.Error
UT-14	JsonHandler	process returns Error when item_na	This test verifies the system's input validation logic for mandatory fields.	JSON-style object: {action: "add_item", item_name: null}	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.Error

Test ID	Class tested	Test name	Test Description	Input	Expected Output
		me is missing			
UT-15	ItemDaoDatabase	insertAndReadItem_persistsAttributes	This test verifies the data persistence and integrity of the database when handling complex objects. It ensures that custom lists of attributes (metadata) are correctly serialized into the database and deserialized back into objects without data loss, specifically checking that null values and multiple list entries remain intact.	JSON-style object: {name: "Jacket", description: "Winter jacket", quantity: 1, attributes: [{key: "size", value: "L"}, {key: "color", value: "red"}, {key: "cold", value: null}]}	The name of the loaded object must be "Jacket" The attributes list must be identical to the input list
UT-16	ItemDaoDatabase	searchItems_findsInsertedItem	This test verifies the search functionality of the database DAO. It ensures that the query logic correctly filters the inventory based on a string match, returning only the items that satisfy the search criteria while ignoring those that do not.	Two distinct item objects are inserted into the database: JSON-style object: {name: "Red Shirt", quantity: 2} JSON-style object: {name: "Blue Shirt", quantity: 1}	The name of the item in the list is "Red Shirt"
UT-17	ItemDaoDatabase	findByName_returnsMatchingItems	This test verifies the exact and partial name matching capability of the database DAO. It ensures that the query logic can identify and return multiple records that share a common naming component, confirming that the search is inclusive of all items containing the specified string.	Two distinct item objects are inserted into the database: JSON-style object: {name: "Socks", quantity: 5} JSON-style object: {name: "Wool Socks", quantity: 2}	Every item in the resulting list contains the substring "Socks" in its name
UT-18	LocationDaoDatabase	insertAndGetLocation	This test verifies the basic lifecycle of a location record within the database	JSON-style object: {name: "Bedroom", type: "ROOM"}	The function db.locationDao().getById(id) returns a Location object.
UT-19	LocationDaoDatabase	testHierarchy_parentAndSubLocations	This test verifies the hierarchical relationship logic between parent and child locations within the database. It ensures that when sub-locations are linked to a parent via a foreign key (parentId), the database correctly organizes this structure and can retrieve all	JSON-style object: {name: "Bedroom", parentId: null} (The Root) JSON-style object: {name: "Closet", parentId: 1} (Child of Bedroom)	The size of the returned list is exactly 2. The names of the objects in the list are "Closet" and "Under Bed".

Test ID	Class tested	Test name	Test Description	Input	Expected Output
			children associated with a specific parent container.	JSON-style object: {name: "Under Bed", parentId: 1} (Child of Bedroom)	
UT-20	LocationDao Database	getRootLocations_returnsOnlyNullParents	This test verifies the system's ability to identify and retrieve top-level (root) locations. It ensures that the database query correctly filters the location tree to return only those entries that do not have a parent, effectively distinguishing between primary zones (like rooms) and their nested sub-locations (like closets or containers).	JSON-style object: {name: "Bedroom", parentId: null} JSON-style object: {name: "Closet", parentId: 1} (Child of Bedroom) JSON-style object: {name: "Kitchen", parentId: null}	The list contains the objects "Bedroom" and "Kitchen"
UT-21	LocationDao Database	deleteLocation_cascadesToSubLocations	This test verifies the cascading delete behavior within the location hierarchy. It ensures that the database maintains referential integrity by automatically removing child locations when their parent location is deleted	JSON-style object: {name: "Box", parentId: null} JSON-style object: {name: "Inside Box", parentId: 1}	A lookup for the parent ID (1) returns null. A lookup for the sub-location ID (2) also returns null
UT-22	Converters	itemAttributes_roundTrip_preservesData	This test verifies the serialization and deserialization logic of the TypeConverters. Since SQLite cannot natively store lists or complex objects, this test ensures that the conversion from a list of ItemAttribute objects to a raw String (usually JSON) and back again is lossless. It specifically checks that the data remains identical, including handling null values for specific attribute fields.	JSON-style object: {attributes: [{key: "size", value: "L"}, {key: "color", value: "red"}, {key: "cold", value: null}]}	The restored list must be exactly equal to the original attributes list
UT-23	Converters	itemAttributes_nullOrBlank_returnsEmptyList	This test verifies the resilience and edge-case handling of the TypeConverters. It ensures that the conversion logic does not crash when encountering "dirty" or missing data (such as nulls, empty strings, or whitespace) and instead returns a safe, empty list	JSON-style object: {input: null} JSON-style object: {input: ""} (Empty String) JSON-style object: {input: " "} (Blank String/Whitespace)	The test uses assertTrue(...isEmpty()) to confirm that the system gracefully defaults to a neutral state rather than returning null or breaking the data flow, ensuring the UI always has a valid list to iterate over.

Test ID	Class tested	Test name	Test Description	Input	Expected Output
UT-24	Converters	itemAttributes_malformedJson_returnsEmptyList	This test verifies the fault tolerance of the TypeConverters when encountering corrupted or unexpected data structures	JSON-style object: {input: "{\"not\":\"a list\"}"}	The test uses assertTrue(restored.isEmpty()) to confirm that the system prioritizes application stability over strict parsing, ensuring that a single piece of malformed metadata doesn't prevent an entire Item from being loaded
UT-25	PromptRobustnessTest	testAddItemWithAttributes	This test verifies the JSON parsing logic for incoming intent responses. It ensures that the parser can correctly transform a raw string into a structured IntentResponse object, specifically handling the extraction of nested maps like attributes	JSON-style object: {action: "add_item", item_name: "jacket", location: ["drawer"], attributes: {color: "red", size: "XL"}}	The action property is exactly "add_item". The item_name property is exactly "jacket". The location list contains exactly one entry: "drawer". The attributes property is not null and contains the key-value pairs for "color" and "size".
UT-26	PromptRobustnessTest	testRetrieveItem	This test verifies the parser's ability to handle optional or null fields within a JSON intent. It ensures that when the AI provides a "retrieve" action—which typically doesn't require specific location or attribute data—the parser correctly maps the provided information and handles the null fields gracefully without failing or throwing a parsing exception.	JSON-style object: {action: "retrieve_item", item_name: "car keys", location: null, attributes: null}	The action property is exactly "retrieve_item". The item_name property is exactly "car keys". The location and attributes properties are both stored as null (or their respective default empty states).
UT-27	PromptRobustnessTest	testUpdateItem	This test verifies the parser's capability to handle update intents, which typically contain a high density of data. It ensures that the system correctly extracts the core item identity, the multi-level nested location path, and specific values from the attributes map	JSON-style object: {action: "update_item", item_name: "trainers", location: ["bedroom", "shoe rack"], attributes: {color: "blue", brand: "Nike"}}	The action property is exactly "update_item". The item_name property is exactly "trainers". The location list preserves the hierarchy: ["bedroom", "shoe rack"]. The brand key within the attributes map is successfully retrieved and its content matches "Nike"

Test ID	Class tested	Test name	Test Description	Input	Expected Output
UT-28	PromptRobustnessTest	testRemoveItem	This test verifies the parser's effectiveness in processing removal intents that include specific qualifiers. It ensures that even when an item is being deleted, the parser correctly captures contextual data like the location and specific attributes. This is vital for scenarios where multiple similar items exist, and the user specifies attributes to ensure the correct instance is removed from the inventory.	JSON-style object: {action: "remove_item", item_name: "umbrella", location: ["garage"], attributes: {condition: "broken"}}	The action property is exactly "remove_item". The item_name property is exactly "umbrella". The location list contains the entry ["garage"]. The condition key within the attributes map is successfully extracted as "broken".
UT-29	PromptRobustnessTest	testJsonWithExtraText	This test verifies the parser's robustness and "noise" filtering capabilities. In real-world scenarios, Large Language Models occasionally wrap their structured JSON output in conversational text (e.g., "Sure!)	JSON-style object: {raw_text: "Sure! {"action":"remove_item"} Done!"}	The action property is correctly extracted as "remove_item". The item_name property is correctly extracted as "umbrella"
UT-30	PromptRobustnessTest	testMultiLevelLocation	This test verifies the parser's handling of multi-level location hierarchies. It ensures that when the AI provides a list of strings representing a path (e.g., from a room to a specific container), the parser preserves the order and structure of that list.	JSON-style object: {action: "add_item", item_name: "socks", location: ["bedroom", "drawer"], attributes: null}	The order of the list remains identical to the JSON input ("bedroom" at index 0, "drawer" at index 1).
UT-31	PromptRobustnessTest	testStringLocationCausesParseFailure	This test verifies the type coercion and data normalization logic within the parser. It checks that the system automatically wraps a single string into a list, maintaining a consistent data format for the rest of the application logic.	JSON-style object: {action: "add_item", item_name: "jacket", location: "drawer", attributes: null}	The test uses assertEquals(listOf("drawer"), result?.location) to confirm that the downstream logic will receive the data in the expected list format, regardless of how the AI formatted the original response
UT-32	PromptRobustnessTest	testCommaSeparatedStringLocationIsSplit	This test verifies the advanced string-to-hierarchy normalization logic. This ensures that the parser can intelligently "unwrap" natural language shortcuts into a properly segmented hierarchy that the database can traverse	JSON-style object: {action: "add_item", item_name: "jacket", location: "bedroom, drawer", attributes: null}	The string "bedroom, drawer" is identified as a path. It is split at the comma, resulting in two distinct strings. The resulting location property is a List containing ["bedroom", "drawer"]

Test ID	Class tested	Test name	Test Description	Input	Expected Output
UT-33	AIModelManager	PromptManager builds correct prompt structure	This test verifies the ChatML formatting logic within the PromptManager. It ensures that raw user input is correctly wrapped in the specialized tokens required by the LLM to distinguish between instructions and user content	A raw user string: "I put my red hammer in the toolbox"	System Boundary: The string must contain <|im_start|>system to define the AI's behavior and constraints. User Boundary: The string must contain <|im_start|>user followed by the actual input text. Data Integrity: The exact user phrase "I put my red hammer in the toolbox" must be present within the prompt. Completion Trigger: The prompt must end with <|im_start|>assistant\n. This acts as a "handoff," telling the model it is now its turn to generate text.
UT-34	AIModelManager	Verify system prompt contains strict JSON instructions	This test verifies the Constraint Enforcement of the system instructions. For an AI to function as a reliable backend component, it must strictly follow a schema. This test ensures that the PromptManager is injecting specific "Guardrail" instructions that force the model to output a predefined JSON structure, explicitly naming the required keys and forbidding conversational filler.	A generic placeholder string: "any"	The prompt must include the core keys required by the IntentResponse data class: "action" (The command type) "item_name" (The subject of the command) "location" (The destination or search area) "attributes" (Metadata/descriptors)
UT-35	AIModelManager	testAddItemOutput	This test verifies the End-to-End integration between the on-device AI model, the prompt management system, and the expected inventory logic. It ensures that the model correctly interprets a natural language sentence and generates a JSON string containing the specific keywords required to process the "add_item" command	A natural language user command: "I put a blue screwdriver in the kitchen drawer"	The resulting string must contain: add_item (The correct intent) screwdriver (The item name) kitchen, drawer (The location hierarchy) blue (The descriptive attribute)

Test ID	Class tested	Test name	Test Description	Input	Expected Output
UT-36	AIModelManager	testRetrieveItemOutput	This test verifies the Natural Language Understanding for information retrieval. It confirms that the on-device AI can differentiate between "storing" an item and "asking" about one.	A natural language query: "Where are my yellow car keys?"	The subject "car keys" (or at least "keys") must be identified as the item name. The descriptor "yellow" must be captured within the attributes or name to narrow the search.
UT-37	AIModelManager	testUpdateItemOutput	This test verifies the contextual reasoning of the AI model when processing movement. Specifically, it ensures that the model can distinguish between the source location (where the item was) and the destination location (where the item is now).	A natural language statement describing a change in location: "I moved the hammer from the garage to the toolbox"	The item_name must be "hammer". The location list (or string) must contain "toolbox", as that is the final state of the item.
UT-38	AIModelManager	testRemoveItemOutput	This test verifies the Semantic Interpretation of removal commands. It ensures that the AI can map colloquial phrases like "threw away" or "got rid of" to the technical remove_item action.	A natural language statement indicating disposal: "I threw away the broken umbrella"	The item_name is identified as "umbrella". The attributes map (or the name string) includes the descriptor "broken"
UT-39	AIModelManager	testComplexLocationOutput	This test verifies the model's ability to handle deeply nested spatial hierarchies. It challenges the AI to decompose a single prepositional phrase ("on the top shelf of the bedroom closet") into a logical sequence of locations.	A directive with multiple nested locations: "Put the batteries on the top shelf of the bedroom closet"	The resulting JSON string must contain: "batteries" (The item) "bedroom" (The room/root location) "closet" (The sub-location) "shelf" (The specific surface)

6.4 Manual Test Cases

Test ID: TC-101
Title/Summary: Verify AI Assistant can add a specific item ("knife") to a specific location ("Kitchen")
Pre-conditions:

- Location "Kitchen" exists in the system.
- User is on the **AI Assistant** tab.

Test Steps:

1. Enter the text "add knife in the kitchen" into the conversational input field.
2. Click the **Send** button.
3. Verify the AI responds with: "Added 'knife' to 'Kitchen'".
4. Navigate to the **Inventory** tab.

5. Select the **Kitchen** location.
6. Check for the presence of the **"knife"** entry.

Expected Result: The AI should process the natural language command correctly, and the "knife" should appear within the "Kitchen" inventory list.
Actual Result: AI confirmed the addition and the "knife" item was visible in the Kitchen inventory.
Status: PASSED

Test ID: TC-102
Title/Summary: Verify AI Assistant can add multiple quantities (3) of an item ("batteries") to a location ("Kitchen").
Pre-conditions:

- Location "Kitchen" exists in the system.
- User is on the **AI Assistant** tab.

Test Steps:

1. Type “add 3 batteries in the kitchen” into the chat bubble.
2. Click the **Send** button.
3. Verify the AI responds with: “Added ‘batteries’ to ‘Kitchen’”.
4. Navigate to the **Inventory** tab.
5. Click on the **“Kitchen”** location.
6. Click on the **“batteries”** item to view details.
7. Check the **Quantity** field.

Expected Result: The AI should confirm the insertion of the batteries, and the inventory should reflect a quantity of **3** for "batteries" in the "Kitchen".
Actual Result: The AI added the item, but the quantity was recorded as 1 instead of 3.
Status: FAILED

Test ID: TC-103
Title/Summary: Verify AI Assistant can add an item ("salt") to a nested location ("Kitchen > shelf").
Pre-conditions:

- A parent location ("Kitchen") exists.
- A sub-location ("shelf") exists inside the "Kitchen".
- User is on the **AI Assistant** tab.

Test Steps:

1. Enter the command: “add salt in shelf inside the Kitchen” into the chat bubble.
2. Click the **Send** button.
3. Verify the AI response is: “Added ‘salt’ to ‘Kitchen > shelf’”.
4. Navigate to the **Inventory** tab.
5. Click the **"Kitchen"** bubble.
6. Click the **"shelf"** sub-location bubble.
7. Verify that the **"salt"** item is present.

Expected Result: The system should correctly identify the nested relationship and place "salt" inside the specific sub-location ("shelf") belonging to the "Kitchen."
Actual Result: The AI successfully parsed the hierarchy; the item appeared in the correct sub-location.
Status: PASSED

Test ID: TC-104
Title/Summary: Verify AI Assistant’s ability to handle misspelled locations (**"Kithen"** instead of **"Kitchen"**).
Pre-conditions:

- Location "Kitchen" exists in the system.
- User is on the **AI Assistant** tab.

Test Steps:

1. Enter the command: “add microwave in the kithen” (intentional typo: "kithen").
2. Click the **Send** button.
3. Verify the AI response is: “Added ‘microwave’ to ‘Kitchen’”.
4. Navigate to the **Inventory** tab.
5. Click on the **“Kitchen”** bubble.
6. Check for the presence of the **“microwave”** item.

Expected Result: The AI should recognize "kithen" as "Kitchen" and place the item in the correct existing location.
Actual Result: The AI successfully mapped the misspelled word to the existing "Kitchen" location and confirmed the addition correctly.
Status: PASSED

Test ID: TC-105
Title/Summary: Verify AI Assistant’s fuzzy matching limits with a significantly misspelled location (**"Litchen"** for **"Kitchen"**).
Pre-conditions:

- Location "Kitchen" exists.
- User is on the **AI Assistant** tab.

Test Steps:

1. Enter the command: “add fish in the Litchen” (intentional typo: "Litchen").
2. Click the **Send** button.
3. Observe the AI response.
4. Navigate to the **Inventory** tab to see if the item was added regardless of the error message.

Expected Result: The AI should identify "Litchen" as "Kitchen" (similar to TC-104) and add the item "fish" to the correct location.
Actual Result: The AI failed to map the word and returned an error message: “Location ‘litchen’ not found”. No item was added to the inventory.
Status: FAILED

Test ID: TC-106
Title/Summary: Verify AI Assistant can correctly retrieve and report the location of an item ("salt").
Pre-conditions:

- Item "salt" exists in the location path: **Kitchen > shelf** (verified in TC-103).
- User is on the **AI Assistant** tab.

Test Steps:

1. Enter the query: “Where is the salt” into the chat bubble.
2. Click the **Send** button.
3. Verify the AI response is: “‘salt’ is in ‘Kitchen > shelf’”.
4. Navigate to **Inventory > Kitchen > shelf** to manually confirm the item is still there.

Expected Result: The AI should correctly identify the item's current location and display the full hierarchical path.

Actual Result: The AI successfully retrieved the location and provided the correct nested path in the response.
Status: PASSED

Test ID: TC-107
Title/Summary: Verify AI Assistant can identify and list multiple locations for duplicate items ("salt").
Pre-conditions:

- Location **Kitchen > shelf** contains "salt".
- Location **Bedroom > shelf** contains "salt".
- User is on the **AI Assistant** tab.

Test Steps:

1. Enter the query: “Where is the salt” into the chat bubble.
2. Click the **Send** button.
3. Verify the AI response includes **both** locations: “‘salt’ is in ‘Kitchen > shelf’ and ‘Bedroom > shelf’”
4. Navigate to the **Inventory** tab.
5. Manually confirm "salt" exists in **Kitchen > shelf**.
6. Manually confirm "salt" exists in **Bedroom > shelf**.

Expected Result: The AI should recognize that "salt" exists in two different places and provide both paths to the user without getting confused or only showing the first result.
Actual Result: The AI correctly identified both instances of the item and returned a comprehensive list of locations.
Status: PASSED

Test ID: TC-108
Title/Summary: Verify AI Assistant can update an item's location ("knife") from a simple location ("Kitchen") to a nested location ("Bedroom > shelf").
Pre-conditions:

- Location **"Kitchen"** exists and contains the item **"knife"**.
- Location **"Bedroom > shelf"** exists and is currently empty of knives.
- User is on the **AI Assistant** tab.

Test Steps:

1. Enter the command: “I moved the knife from the kitchen to the bedroom shelf”.
2. Click the **Send** button.
3. Verify the AI response: “Updated ‘knife’ - now in ‘Bedroom > shelf’”.
4. Navigate to the **Inventory** tab.
5. Open the **"Kitchen"** location and verify the "knife" is **no longer present**.
6. Open **"Bedroom > shelf"** and verify the "knife" **is now present**.

Expected Result: The system should successfully transfer the item record from the source location to the destination location, ensuring no duplicates are left behind in the original spot.
Actual Result: The "knife" was successfully removed from the Kitchen and appeared in the Bedroom shelf; the AI confirmed the update correctly.
Status: PASSED

Test ID: TC-109
Title/Summary: Verify AI disambiguation when moving an item ("salt") that exists in multiple locations.

Pre-conditions:
- Location **Kitchen > shelf** contains "salt".
- Location **Bedroom > shelf** contains "salt".
- User is on the **AI Assistant** tab.

Test Steps:
1. Enter the command: “I moved the salt to the Kitchen”.
2. Observe the AI response for a clarification request.
3. Verify that two options are presented: “salt - Bedroom > shelf” and “salt - Kitchen > shelf”.
4. Select the first option (Bedroom > shelf).
5. Confirm the AI final response: “Updated ‘salt’ - now in ‘Kitchen’”.
6. Navigate to the **Inventory** tab.
7. Verify "salt" is now present in **Kitchen** (root level).
8. Verify "salt" is **no longer present** in **Bedroom > shelf**.

Expected Result: The AI should not move an item automatically if duplicates exist; it must prompt the user to select the correct source location and then execute the move accurately.
Actual Result: The system correctly identified the ambiguity, provided a selection UI, and updated the inventory database based on the user's choice.
Status: PASSED

Test ID: TC-110
Title/Summary: Verify AI's ability to delete a specific item ("salt") using a misspelled location command while handling duplicates.
Pre-conditions:
- Location **Kitchen** contains "salt".
- Location **Kitchen > shelf** contains "salt".
- User is on the **AI Assistant** tab.

Test Steps:
1. Enter the command: “Remove the salt from the kitchent” (intentional typo: "kitchent").
2. Observe the AI response for a disambiguation prompt.
3. Verify the options presented include the corrected locations (“salt - Kitchen” and “salt - Kitchen > shelf”).
4. Select the option: “salt - Kitchen”.
5. Confirm the AI response: “Removed ‘salt’ (1 item)”.
6. Navigate to the **Inventory** tab.
7. Open the **Kitchen** location and verify the "salt" is **gone**.
8. Open the **Kitchen > shelf** location and verify that the "salt" there **remains untouched**.

Expected Result: The AI should correctly identify the misspelled location, ask which "salt" to remove, and only delete the item from the user-selected location.
Actual Result: The system handled the typo, prompted for clarification, and successfully deleted only the intended item.
Status: PASSED

6.5 User Acceptance Testing

During the final stages of development, a few team members shared the application with close relatives to get some external feedback about how intuitive the UI/UX is. The testers were allowed to try out the app as if they were using the features for actual household tracking. We first let the testers try the app as they wished, then once an obstacle was met, we intervened and guided them to the

features. Overall, there were no big barriers to entry, all the testers had an easy time understanding the UI, the only issue arose in prompting the chatbot. There were a few cases where the texts the testers typed were not properly interpreted by the AI, and therefore caused incorrect intents.

Tester	Feedback	Usability	Accessibility	UI/UX	General
Tester 1	I would use this app to track my seasonal items. The use of AI is very useful and natural which makes it easier to list things. Only problem is the AI doesn't understand some things I said.	9/10	8/10	10/10	9/10
Tester 2	Very easy to use and looks very nice. Personally I wouldn't use but I understand why people would use it.	10/10	9/10	10/10	9.5/10
Tester 3	AI needs better tuning for more complex texts. Overall the app looks really good and simple to use.	8/10	7.5/10	9/10	8/10

7. Test Results

7.1 Results Summary Unit Test Cases

Test ID	Class tested	Test name	Status	Expected Output	PASS/FAIL
UT-01	JsonHandler	addItem succeeds with simple single location	Passed	AiActionResult.ItemAdded. This result object contains: itemName: "Hammer" locationPath: ["Shed"]	Actual output matches the Expected one
UT-02	JsonHandler	addItem succeeds with nested location path	Passed	AiActionResult.ItemAdded. This result object contains: itemName: "Bike" locationPath: ["Home", "Garage"]	Actual output matches the Expected one
UT-03	JsonHandler	addItem returns LocationNotFound	Passed	The function jsonHandler.process(response) returns a sealed class object of type	Actual output matches the Expected one

Test ID	Class tested	Test name	Status	Expected Output	PASS/FAIL
		when path is broken		AiActionResult.LocationNotFound	
UT-04	JsonHandler	addItem creates separate entries for items with duplicate names	Passed	For both inputs, the function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemAdded The system performs a lookup for the "Toolbox" location. The system executes the insertItem command twice.	Actual output matches the Expected one
UT-05	JsonHandler	addItem adds two different items to the same location	Passed	For both inputs, the function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemAdded	Actual output matches the Expected one
UT-06	JsonHandler	remove item items	Passed	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemRemoved	Actual output matches the Expected one
UT-07	JsonHandler	removeItem removes an item that has a nested location	Passed	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemRemoved	Actual output matches the Expected one
UT-08	JsonHandler	retrieveItem builds path for match in single location	Passed	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemFound	Actual output matches the Expected one
UT-09	JsonHandler	retrieveItem nested location path for matches	Passed	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemFound	Actual output matches the Expected one
UT-10	JsonHandler	updateItem test	Passed	The function jsonHandler.processChosenItem (chosenId, action) returns a sealed class object of type AiActionResult.ItemUpdated	Actual output matches the Expected one
UT-11	JsonHandler	resolveLocationPath succeeds using	Passed	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemAdded	Actual output matches the Expected one

Test ID	Class tested	Test name	Status	Expected Output	PASS/FAIL
		hierarchy when the order is different			
UT-12	JsonHandler	resolveLocationPath uses tie-breaker for ambiguous hierarchy matches	Passed	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.ItemAdded	Actual output matches the Expected one
UT-13	JsonHandler	process returns Error for unknown action	Passed	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.Error	Actual output matches the Expected one
UT-14	JsonHandler	process returns Error when item_name is missing	Passed	The function jsonHandler.process(response) returns a sealed class object of type AiActionResult.Error	Actual output matches the Expected one
UT-15	ItemDaoDatabase	insertAndReadItem_persistsAttributes	Passed	The name of the loaded object must be "Jacket" The attributes list must be identical to the input list	Actual output matches the Expected one
UT-16	ItemDaoDatabase	searchItems_findsInsertedItem	Passed	The name of the item in the list is "Red Shirt"	Actual output matches the Expected one
UT-17	ItemDaoDatabase	findByName_returnsMatchingItems	Passed	Every item in the resulting list contains the substring "Socks" in its name	Actual output matches the Expected one
UT-18	LocationDao Database	insertAndGetLocation	Passed	The function db.locationDao().getById(id) returns a Location object.	Actual output matches the Expected one
UT-19	LocationDao Database	testHierarchy_parentAndSubLocations	Passed	The size of the returned list is exactly 2. The names of the objects in the list are "Closet" and "Under Bed".	Actual output matches the Expected one
UT-20	LocationDao Database	getRootLocations_returnsOnlyNullParents	Passed	The list contains the objects "Bedroom" and "Kitchen"	Actual output matches the Expected one

Test ID	Class tested	Test name	Status	Expected Output	PASS/FAIL
UT-21	LocationDao Database	deleteLocation_cascadesToSubLocations	Passed	A lookup for the parent ID (1) returns null. A lookup for the sub-location ID (2) also returns null	Actual output matches the Expected one
UT-22	Converters	itemAttributes_roundTrip_preservesData	Passed	The restored list must be exactly equal to the original attributes list	Actual output matches the Expected one
UT-23	Converters	itemAttributes_nullOrBlank_returnsEmptyList	Passed	The test uses assertTrue(...isEmpty()) to confirm that the system gracefully defaults to a neutral state rather than returning null or breaking the data flow, ensuring the UI always has a valid list to iterate over.	Actual output matches the Expected one
UT-24	Converters	itemAttributes_malformedJson_returnsEmptyList	Passed	The test uses assertTrue(restored.isEmpty()) to confirm that the system prioritizes application stability over strict parsing, ensuring that a single piece of malformed metadata doesn't prevent an entire Item from being loaded	Actual output matches the Expected one
UT-25	PromptRobustnessTest	testAddItemWithAttributes	Passed	The action property is exactly "add_item". The item_name property is exactly "jacket". The location list contains exactly one entry: "drawer". The attributes property is not null and contains the key-value pairs for "color" and "size".	Actual output matches the Expected one
UT-26	PromptRobustnessTest	testRetrieveItem	Passed	The action property is exactly "retrieve_item". The item_name property is exactly "car keys". The location and attributes properties are both stored as null (or their respective default empty states).	Actual output matches the Expected one
UT-27	PromptRobustnessTest	testUpdateItem	Passed	The action property is exactly "update_item". The item_name property is exactly "trainers".	Actual output matches the Expected one

Test ID	Class tested	Test name	Status	Expected Output	PASS/FAIL
				The location list preserves the hierarchy: ["bedroom", "shoe rack"]. The brand key within the attributes map is successfully retrieved and its content matches "Nike"	
UT-28	PromptRobustnessTest	testRemoveItem	Passed	The action property is exactly "remove_item". The item_name property is exactly "umbrella". The location list contains the entry ["garage"]. The condition key within the attributes map is successfully extracted as "broken".	Actual output matches the Expected one
UT-29	PromptRobustnessTest	testJsonWithExtraText	Passed	The action property is correctly extracted as "remove_item". The item_name property is correctly extracted as "umbrella"	Actual output matches the Expected one
UT-30	PromptRobustnessTest	testMultiLevelLocation	Passed	The order of the list remains identical to the JSON input ("bedroom" at index 0, "drawer" at index 1).	Actual output matches the Expected one
UT-31	PromptRobustnessTest	testStringLocationCausesParseFailure	Passed	The test uses assertEquals(listOf("drawer"), result?.location) to confirm that the downstream logic will receive the data in the expected list format, regardless of how the AI formatted the original response	Actual output matches the Expected one
UT-32	PromptRobustnessTest	testCommaSeparatedStringLocationIsSplit	Passed	The string "bedroom, drawer" is identified as a path. It is split at the comma, resulting in two distinct strings. The resulting location property is a List containing ["bedroom", "drawer"]	Actual output matches the Expected one
UT-33	AIModelManager	PromptManager builds correct prompt structure	Passed	System Boundary: The string must contain <|im_start|>system to define the AI's behavior and constraints. User Boundary: The string must contain <|im_start|>user	Actual output matches the Expected one

Test ID	Class tested	Test name	Status	Expected Output	PASS/FAIL
				followed by the actual input text. Data Integrity: The exact user phrase "I put my red hammer in the toolbox" must be present within the prompt. Completion Trigger: The prompt must end with <|im_start|>assistant\n. This acts as a "handoff," telling the model it is now its turn to generate text.	
UT-34	AIModelManager	Verify system prompt contains strict JSON instructions	Passed	The prompt must include the core keys required by the IntentResponse data class: "action" (The command type) "item_name" (The subject of the command) "location" (The destination or search area) "attributes" (Metadata/descriptors)	Actual output matches the Expected one
UT-35	AIModelManager	testAddItemOutput	Passed	The resulting string must contain: add_item (The correct intent) screwdriver (The item name) kitchen, drawer (The location hierarchy) blue (The descriptive attribute)	Actual output matches the Expected one
UT-36	AIModelManager	testRetrieveItemOutput	Passed	The subject "car keys" (or at least "keys") must be identified as the item name. The descriptor "yellow" must be captured within the attributes or name to narrow the search.	Actual output matches the Expected one
UT-37	AIModelManager	testUpdateItemOutput	Passed	The item_name must be "hammer". The location list (or string) must contain "toolbox", as that is the final state of the item.	Actual output matches the Expected one
UT-38	AIModelManager	testRemoveItemOutput	Passed	The item_name is identified as "umbrella". The attributes map (or the name string) includes the descriptor "broken"	Actual output matches the Expected one
UT-39	AIModelManager	testComplexLocationOutput	Passed	The resulting JSON string must contain: "batteries" (The item)	Actual output matches the Expected one

Test ID	Class tested	Test name	Status	Expected Output	PASS/FAIL
				"bedroom" (The room/root location) "closet" (The sub-location) "shelf" (The specific surface)	

7.2 Issues Found & Resolutions

During Phase 4 testing, a total of 39 unit tests were executed, all of which passed successfully, indicating that individual components functioned as expected in isolation.

However, manual testing revealed three issues related to system behavior and user interaction that were not captured by automated tests. These issues are documented below.

Issue ID	Description	Severity	Root Cause	Resolution	Retest Result
MT-01	AI fails to parse quantity during addition (Ref: TC-102).	Medium	SLM implementation didn't account for quantity the model doesn't output in the json this information	The SLM doesn't account for outputting quantity, this is a prompt issue, but it brings with it complications including writing json, which we wanted to make sure outputs correct information more often than not	FAIL
MT-02	AI fuzzy matching limitation on first-letter substitution (Ref: TC-105).	Low	We can't control the fuzzy matching done by the SLM, but we also didn't account for and implementation of fuzzy matching on the backend	Most likely it should be handled in the backend, but it is not implemented at the moment since we haven't reached this part of development	FAIL

7.3 Known Limitations

Even though the application meets the main goal of being a fully offline conversational home inventory system, there are still some limitations in the current version.

One limitation is that the user cannot choose their own AI model. In our implementation, the model is bundled with the APK instead of letting the user select and download a compatible open source model themselves. This made the app easier to set up and ensured that the chosen model worked correctly with our prompt format and JSON based pipeline, but it also reduced flexibility for the user.

Another limitation is the size and hardware demands of AI models. In practice, many language models are too large to run properly on a phone because they require too much storage, memory, and computation. Because of this, we had to use a smaller model, Qwen2.5-Coder-1.5B-Instruct, as a compromise between performance and what was realistically possible on mobile hardware. While this model worked well for our structured JSON pipeline, it is still less capable than larger models and can be slower or less accurate on more difficult prompts, especially on weaker devices.

A further limitation is synchronization. Since the whole system is offline by design, it is much harder to keep multiple devices in sync than it would be with a cloud backend. To solve this, we used Bluetooth and JSON based transfer between devices. This works for local sharing, but it is still a simplified solution. In practice, syncing is not as robust as a real conflict-resolution system, and in some cases one version of the database may end up overriding or duplicating parts of another.

The app also depends quite heavily on the model producing correct JSON output. Although we added parsing and fallback logic, the system still relies on the model correctly identifying the action, item name, location, and attributes. If the model gives malformed or incomplete output, the result may be incorrect or require extra clarification from the user.

7.4 SLM Evaluation Results

The evaluation was run on a Samsung Galaxy S24 Ultra against the 57 valid test cases. The top level results are summarized below.

Metric	Result	Target	Pass?
Strict pass rate	47.4% (27/57)	>= 70%	No
Partial pass rate	80.7% (46/57)	>= 75%	Yes
Mean partial score	0.869	>= 0.82	Yes
Hallucination count	7	0	No
Average inference latency	9,718 ms	< 4,000 ms	No
Action macro-F1	0.966	>= 0.90	Yes

The most informative result is the action classification performance shown through a 4x4 confusion matrix across all intent classes.

Expected\Predicted	add	update	remove	retrieve
Add (16 cases)	14	-	1	1

Update (10 cases)	-	10	-	-
remove	-	-	11	-
retrieve	-	-	-	20

Per class F1 scores: add = 0.933, update = 1.000, remove = 0.957, retrieve = 0.976. This yields a macro-F1 of 0.966 which means that the model almost never misunderstands what the user intends to do. The table below shows the breakdown by test category.

Category	Cases	Strict %	Avg Score	Hallucination
basic_add	4	75%	0.97	0
basic_remove	7	57%	0.87	1
basic_retrieve	4	50%	0.74	2
basic_update	6	17%	0.86	0
complexity_short	3	67%	0.95	0
complexity_long	4	50%	0.88	0
deep_hierarchy	6	50%	0.93	0
hallucination_probe	8	63%	0.88	1
multi_attribute	8	25%	0.81	2
robustness	7	43%	0.86	1

Strict pass rate failures were because of three root causes: location segmentation errors, where the models returned a merged string rather than a split array (“office safe” instead of [“office”,”safe”]). These cases were handled in the backend through the location resolution mechanism described earlier, which normalises and interprets imperfect model-produced location paths before resolving them to valid database entries. The second cause was over-extracting the attributes where the model would include attributes inferred from context but were not explicitly stated in the input. The third cause is wrong item names. The model would return a compound name rather than the actual name (“blue Nike trainers” instead of “trainers”), this causes a strict match failure but partial points are awarded via the F1 token.

These results demonstrate that the SLM is viable for application’s core purpose. The almost perfect intent classification, macro-F1 being 0.966, shows that the application will almost never perform the wrong action corresponding to the input. The 80.7% partial pass rate confirms that the mode accurately extracts the key fields in the majority of the cases. The identified failures are either fixed by prompt engineering or by backend support. The hallucination detection and penalty mechanism is functioning properly, all the 7 hallucination outputs were graded below the 0.75 partial pass threshold which shows that they will not be silently accepted by the application.

8. Conclusion

8.1 Meetings with Client

Throughout the duration of the project, we had several meetings with our client to get feedback on our design choices, implementations, architectural decisions, etc. We had meetings every week where we would bring him up to date with the current progression and ask or answer any questions he might have had for us. Our feedback was overall positive during our meetings and helped us quickly progress in the development of the app.

8.2 Project Presentations

During the module, we had the opportunity to present our progress to peers and also present the project to a board. During the peer presentations, we were open to receiving harsh criticism about our project, however, every time we got minimal or very simple questions which boosted our confidence. At the board presentation, we had hands-on experience to present our project in a more scientific way, going in depth about the technical side of the project. In this presentation the feedback was also quite simple which showed us that the project was well explained and communicated. Overall, the feedback we had received throughout the whole duration of this module was overwhelmingly positive.

8.3 Challenges Faced During Bluetooth Development

The biggest challenge in the Bluetooth part was synchronizing two different offline databases. Since each device stores its own inventory locally, it was hard to merge the data without causing duplicates, conflicts, or mistakes in the location hierarchy. This made the feature much more complex than we first expected.
Testing was also difficult because Bluetooth functionality always requires two physical devices. Since these were not always available during development, debugging and improving the feature took more time.
Because of these issues, the Bluetooth merge functionality was not completed to a fully stable level. To keep the final product focused on features that worked reliably, the Bluetooth code was placed in a separate folder instead of being part of the main application.

8.4 Lessons Learned in Hierarchical Data Modeling

A key lesson from this project was that hierarchical data modelling is powerful, but also more complex than expected. Using a self referential structure allowed us to represent realistic household storage paths with different levels of depth, which made the system more flexible.
But at the same time, it created challenges in resolving location paths, handling duplicate names, and keeping the data consistent. This showed us that hierarchical models need both database level constraints and backend logic to work reliably, especially when the input comes from natural language instead of fixed forms.

8.5 Potential Future Features

Future improvements could make the application more practical and easier to use. Voice input could make interaction with the chatbot faster and more natural, while barcode scanning could simplify the

process of registering items. Another possible improvement is optional cloud synchronization. Although the current system was intentionally built as a fully offline solution, an opt-in cloud sync feature could make multi-device usage more convenient for users who are willing to trade some privacy for accessibility.

9. AI Disclosure

AI tools were used to generate or assist in the following areas: boilerplate code for Android components, debugging assistance and implementation of some modules where the team faced time constraints due to reduced team capacity. All substantive architectural and design decisions were made by the team without any AI involvement. This includes selection and justification of the SLM model (Qwen2.5), design of the location resolution algorithm, the hierarchical database schema with self-referential location modelling, the structured JSON prompt engineering pipeline and ambiguity resolution strategy.

The team did not have any background in Android development or Kotlin. Setting up the Gradle build system, resolving dependency conflicts between llama.cpp and the target SDK, and configuring the project ABI filters and asset exclusion rules involved a large amount of Android-specific boilerplate that was not directly related to the project. AI assistance allowed us to navigate through this without getting stuck on config issues so that meaningful development could begin without spending the majority of our time on tooling.

Android's framework requires a large amount of structural code (adapters, DAOs, ViewModels, state flows) that follow well established standards but takes a considerable amount of time to write correctly. Since we had absolutely 0 experience in this and the main focus of the project was to focus on the AI and database layers, we used AI to generate some structural boilerplates which were then understood, reviewed, adjusted and integrated with our own project and logic. This allowed us to focus our effect on the parts of the application that require genuine design decisions rather than repetitive frameworks.

Several integration issues arose during development that were difficult to diagnose especially those relating to Android-specific concurrency. We used AI as a diagnostic tool to trace errors which helped us identify the root causes quickly. Then the fixes were designed, implemented and validated by us.

AI was also used to generate Markdown reports summarising major architectural and implementation changes. These reports supported internal knowledge transfer by giving team members a quick way to understand recent changes in the codebase, particularly when the main contributor for that part was unavailable. The reports were used as an initial reference point for orientation, while deeper clarification and final technical understanding still came through direct discussion with the responsible team member.

During the initial development of the database layer, AI was used to generate a simple temporary database interface for internal testing. This made it easier to validate database behaviour and demonstrate progress to other team members, but it was used only as a development aid and was not part of the final application design.

At one stage of development, the team had been working without a fully unified application structure, which led to significant integration issues when merging major branches. In particular, the database branch and the SLM integration branch had diverged into effectively different application setups, including differences in package namespaces, Kotlin and Gradle configuration, UI frameworks, and dependency structure. This resulted in multiple hard merge conflicts and a large number of unresolved reference errors after the initial merge.

Because resolving these issues manually was time-consuming and highly error-prone, AI tools were used to help analyse the divergence, identify the main incompatibilities, and suggest reconciliation steps during the merge process. This included support in tracing dependency mismatches, package migration issues, and conflicts caused by the coexistence of Fragment-based and Compose-based code. The final integration and validation of the merged codebase, however, remained the responsibility of the team.

GitHub Copilot's pull request review functionality was also used as an auxiliary code review tool. It generated automated review comments on proposed changes, such as highlighting package structure inconsistencies, potential performance concerns, accessibility issues, dead code paths, and missing test coverage. These comments were treated as advisory only: the team reviewed them critically, decided which suggestions were relevant, and remained responsible for all final code review and implementation decisions.

Additionally at the end, AI was used to clean up the code and delete legacy or redundant files. Redundant variables, classes and functions were cleaned up manually. Some config files were also cleaned up using the assistance of AI including the removal of debugging code.

References

1. Android Room / local database

Android Developers. Save data in a local database using Room. https://developer.android.com/training/data-storage/room

2. SQLite foreign keys / referential integrity

SQLite Documentation. SQLite Foreign Key Support. https://sqlite.org/foreignkeys.html

3. Jetpack Compose

Android Developers. Jetpack Compose Material 3 and Material Design 3 in Compose. https://developer.android.com/jetpack/androidx/releases/compose-material3

4. Bluetooth Classic / RFCOMM

Android Developers. Bluetooth overview and Connect Bluetooth devices. https://developer.android.com/develop/connectivity/bluetooth

5. Bluetooth permissions

Android Developers. Bluetooth permissions. https://developer.android.com/develop/connectivity/bluetooth/bt-permissions

6. Android local file storage / privacy

Android Developers. Data and file storage overview and Access app-specific files. https://developer.android.com/training/data-storage

7. llama.cpp for on-device inference

ggml-org. llama.cpp https://github.com/ggml-org/llama.cpp

8. Qwen2.5 with llama.cpp

Qwen Documentation. Run Qwen2.5 with llama.cp*. https://qwen.readthedocs.io/en/v2.5/run_locally/llama.cpp.html

9. Local-first design philosophy

Kleppmann, M., et al. Local-First Software: You Own Your Data, in spite of the Cloud. https://www.inkandswitch.com/essay/local-first/local-first.pdf

10. On-device AI background

Wang, X., et al. A Comprehensive Survey on On-Device AI Models. https://arxiv.org/pdf/2503.06027

Appendices

Appendix A: Source Code

A.1 - Kotlin implementation of multi-stage location path resolution

This appendix contains the main Kotlin methods used to normalize, resolve, and disambiguate model-generated location paths.

A.1.1 Entry-point resolver

```kotlin
private suspend fun resolveLocationPath(path: List<String>): ResolvedLocation? {
        val normalizedPath = mutableListOf<String>()
        for (segment in path) {
            val trimmed = segment.trim()
            if (trimmed.isNotEmpty()) {
                normalizedPath += trimmed
            }
        }
        if (normalizedPath.isEmpty()) return null

        Log.d(TAG, "Resolving location path=$normalizedPath")

        val allLocations = repository.getAllLocations().first()
        val expandedCandidates = buildExpandedPathCandidates(normalizedPath, allLocations)
        if (expandedCandidates.size > 1) {
            Log.d(TAG, "Expanded into ${expandedCandidates.size} candidate path(s):
             $expandedCandidates")
        }

        val exactResolved = tryResolveExactCandidates(expandedCandidates)
        if (exactResolved != null) {
            return exactResolved
        }

        val hierarchyResolved = tryResolveHierarchyCandidates(expandedCandidates,
            allLocations)
        if (hierarchyResolved != null) {
            return hierarchyResolved
        }

        Log.w(TAG, "Failed to resolve location path=$normalizedPath")

        return null
```

```
    }
```

A.1.2 Exact candidate resolution

```
    private suspend fun tryResolveExactCandidates(candidates: List<List<String>>):
ResolvedLocation? {
        for (candidate in candidates) {
            // Prefer AI-provided (or expanded) order first.
            val exactResolved = resolveExactPath(candidate)
            if (exactResolved != null) {
                Log.d(TAG, "Resolved using exact order, locationId=${exactResolved.id},
path=${exactResolved.path}, candidate=$candidate")
                return exactResolved
            }

            // Cheap fallback for fully reversed responses.
            val reversedResolved = resolveExactPath(candidate.asReversed())
            if (reversedResolved != null) {
                Log.d(TAG, "Resolved using reversed order,
locationId=${reversedResolved.id}, path=${reversedResolved.path}, candidate=$candidate")
                return reversedResolved
            }
        }

        return null
    }
```

A.1.3 Hierarchy-aware fallback resolution

```
    private suspend fun tryResolveHierarchyCandidates(
        candidates: List<List<String>>,
        allLocations: List<Location>
    ): ResolvedLocation? {
        // Hierarchy-aware fallback: treat each candidate as unordered names
        // and find a valid parent-child chain in the location tree.
        for (candidate in candidates) {
            val hierarchyResolved = resolveByHierarchy(candidate, allLocations)
            if (hierarchyResolved != null) {
                Log.d(TAG, "Resolved using hierarchy matching,
locationId=${hierarchyResolved.id}, path=${hierarchyResolved.path}, candidate=$candidate")
                return hierarchyResolved
            }
        }
        return null
    }
```

A.1.4 Exact parent-child path matching

```
    private suspend fun resolveExactPath(path: List<String>): ResolvedLocation? {
        var currentParentId: Long? = null
        val resolvedPath = mutableListOf<String>()

        for (segment in path) {
            val loc = repository.findLocationByNameAndParent(segment, currentParentId)
            if (loc == null) {
                return null
            }

            currentParentId = loc.id
            resolvedPath += loc.name
        }

        if (currentParentId == null) {
            return null
        }

        return ResolvedLocation(id = currentParentId, path = resolvedPath)
    }
```

A.1.5 Unordered hierarchy matching with tie-breaker

```
    private suspend fun resolveByHierarchy(path: List<String>, allLocations:
List<Location>? = null): ResolvedLocation? {
        val targetCounts = mutableMapOf<String, Int>()
        for (name in path) {
            val normalized = normalizeName(name)
            val current = targetCounts[normalized] ?: 0
            targetCounts[normalized] = current + 1
        }

        val locations = if (allLocations != null) {
            allLocations
        } else {
            repository.getAllLocations().first()
        }
        if (locations.isEmpty()) return null

        val byId = mutableMapOf<Long, Location>()
        for (location in locations) {
            byId[location.id] = location
        }

        val candidates = mutableListOf<Location>()
        for (location in locations) {
```

```
            val normalizedName = normalizeName(location.name)
            if (targetCounts.containsKey(normalizedName)) {
                candidates += location
            }
        }

        val matches = mutableListOf<Location>()
        for (candidateLeaf in candidates) {
            val chain = buildRootToLeafChain(candidateLeaf, byId)
            if (chain.size != path.size) continue

            val chainCounts = mutableMapOf<String, Int>()
            for (locationInChain in chain) {
                val normalizedName = normalizeName(locationInChain.name)
                val current = chainCounts[normalizedName] ?: 0
                chainCounts[normalizedName] = current + 1
            }

            if (chainCounts == targetCounts) {
                matches += candidateLeaf
            }
        }

        if (matches.isEmpty()) return null
        if (matches.size == 1) {
            val leaf = matches.first()
            val chain = buildRootToLeafChain(leaf, byId)
            val resolvedNames = mutableListOf<String>()
            for (locationInChain in chain) {
                resolvedNames += locationInChain.name
            }
            return ResolvedLocation(id = leaf.id, path = resolvedNames)
        }

        Log.d(TAG, "Multiple hierarchy matches found (${matches.size}) for path=$path;
applying tie-breaker")

        // Tie-breaker: choose the match whose root->leaf order is closest to AI order.
        val normalizedInput = mutableListOf<String>()
        for (segment in path) {
            normalizedInput += normalizeName(segment)
        }

        var selectedLeaf: Location? = null
        var bestScore = Int.MIN_VALUE
        for (leaf in matches) {
```

```
            val chain = buildRootToLeafChain(leaf, byId)
            var score = 0

            for (index in normalizedInput.indices) {
                val chainName = normalizeName(chain[index].name)
                if (chainName == normalizedInput[index]) {
                    score += 1
                }
            }

            if (score > bestScore) {
                bestScore = score
                selectedLeaf = leaf
            }
        }

        if (selectedLeaf == null) {
            return null
        }

        val selectedChain = buildRootToLeafChain(selectedLeaf, byId)
        val resolvedNames = mutableListOf<String>()
        for (locationInChain in selectedChain) {
            resolvedNames += locationInChain.name
        }
        return ResolvedLocation(id = selectedLeaf.id, path = resolvedNames)
    }
```

A.2

A.2.1 - Bluetooth Manager class declaration

```
class BluetoothManager(private val context: Context, private val listener:
BluetoothListener)
```

A 2.2 - Bluetooth Manager class content

```
            }
fun startServer() {
   scope.launch {
        try {
            val adapter = BluetoothAdapter.getDefaultAdapter()
            if (ActivityCompat.checkSelfPermission(
```

```
                    context,
Manifest.permission.BLUETOOTH_CONNECT
                ) != PackageManager.PERMISSION_GRANTED
            ) {
                listener.onError("Missing BLUETOOTH_CONNECT permission")
                return@launch
            }
                                                        serverSocket      =
adapter.listenUsingRfcommWithServiceRecord("SyncServer", uuid)
            val socket = serverSocket!!.accept()
            _connectedDevice = socket.remoteDevice // <-- track device
            output = socket.outputStream
            input = socket.inputStream
            listener.onConnected()
            listenForData()
        } catch (e: Exception) {
            listener.onError("Server error: " + e.message)
        }
    }
}

@SuppressLint("MissingPermission")
fun connect(device: BluetoothDevice) {
    scope.launch {
        try {
            adapter?.cancelDiscovery()

            try {
                socket = device.createRfcommSocketToServiceRecord(uuid)
                socket?.connect()
            } catch (e: Exception) {
                // fallback
                socket = device.javaClass
                    .getMethod("createRfcommSocket", Int::class.java)
                    .invoke(device, 1) as BluetoothSocket
                socket?.connect()
```

```
            _connectedDevice = device
            output = socket?.outputStream
            input = socket?.inputStream

            listener.onConnected()
            listenForData()

        } catch (e: Exception) {
            e.printStackTrace()
            listener.onError("Connection failed: ${e.javaClass.simpleName}")
            close()
        }
    }
}
```

```
fun sendData(data: String) {
 scope.launch {
        try {
            val chunkSize = 512
            val chunks = (data + "<<END>>").chunked(chunkSize)
            for (chunk in chunks) {
                output?.write(chunk.toByteArray())
                delay(30)
            }
        } catch (e: Exception) {
            listener.onError("Send failed: ${e.message}")
        }
    }
}

private fun listenForData() {
    scope.launch {
        val buffer = ByteArray(1024)
        val sb = StringBuilder()
        try {
            while (true) {
                val bytes = input?.read(buffer) ?: break
                val chunk = String(buffer, 0, bytes)
                sb.append(chunk)
                if (sb.contains("<<END>>")) {
                    val full = sb.toString().replace("<<END>>", "")
                    sb.clear()
                    listener.onDataReceived(full)
                }
            }
        } catch (e: Exception) {
            listener.onDisconnected()
            _connectedDevice = null
            close()
        }
    }

}
```

A.3

A.3.1 - Room database configuration

```
@Database(entities = [Item::class, Location::class], version = 1, exportSchema = false)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {
    abstract fun itemDao(): ItemDao
    abstract fun locationDao(): LocationDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase =
```

```
            INSTANCE ?: synchronized(this) {
                val db = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "offline_inventory_database"
                )
                    .build()

                INSTANCE = db
                db
            }}}
```

A.3.2 - Location Entity

```
@Entity(
    tableName = "locations",
    foreignKeys = [
        ForeignKey(
            entity = Location::class,
            parentColumns = ["id"],
            childColumns = ["parentId"],
            onDelete = ForeignKey.CASCADE
        )
    ],
    indices = [
        Index("parentId"),
        Index(value = ["parentId", "name"], unique = true)
    ]
)
data class Location(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val name: String,
    val description: String? = null,
    val parentId: Long? = null,
    val createdAt: Long = System.currentTimeMillis()
)
```

A.3.3 - Item Entity

```
@Entity(
    tableName = "items",
    foreignKeys = [
        ForeignKey(
            entity = Location::class,
            parentColumns = ["id"],
            childColumns = ["locationId"],
            onDelete = ForeignKey.SET_NULL
        )
    ],
    indices = [Index(value = ["locationId"])]
)
data class Item(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0,
    val name: String,
    val description: String? = null,
    // Flexible per-item metadata (e.g., size=L, color=red, cold=null).
    val attributes: List<ItemAttribute> = emptyList(),
    val quantity: Int = 1,
```

```
    val locationId: Long? = null,
    val imageBytes: ByteArray? = null,
    val createdAt: Long = System.currentTimeMillis()
)

@Serializable
data class ItemAttribute(
    // Attribute name, such as "size" or "color".
    val key: String,
    // Nullable so an attribute can be present but unspecified.
    val value: String? = null
)
```

A.3.4 DAO and repository layer snippets

A.3.4.1 LocationDao

```
@Dao
interface LocationDao {
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun insert(location: Location): Long

    @Update
    suspend fun update(location: Location)

    @Delete
    suspend fun delete(location: Location)

    @Query("SELECT * FROM locations WHERE id = :id")
    fun getLocation(id: Long): Flow<Location>

    //idea get the value of the storage
    @Query("SELECT id FROM locations WHERE name = :name")
    fun getLocationID(name: String): Flow<Long>

    //return the storages with given parent ID
    @Query("SELECT name FROM locations WHERE parentId = :id")
    fun getLocationsByParentID(id: Long): List<String>

    @Query("SELECT EXISTS(SELECT 1 FROM items WHERE name = :name AND locationId = :idParent
LIMIT 1)")
    fun checkLocationExistsWithGivenParent(name: String, idParent: Long): Flow<Boolean>

    @Query("SELECT * FROM locations WHERE parentId IS NULL ORDER BY name ASC")
    fun getRootLocations(): Flow<List<Location>>

    @Query("SELECT * FROM locations WHERE parentId = :parentId ORDER BY name ASC")
    fun getSubLocations(parentId: Long): Flow<List<Location>>

    @Query("SELECT * FROM locations")
    fun getAllLocations(): Flow<List<Location>>

    @Query("SELECT * FROM locations WHERE TRIM(name) = TRIM(:name) COLLATE NOCASE LIMIT 1")
    suspend fun findByName(name: String): Location?

    @Query("SELECT * FROM locations WHERE id = :id LIMIT 1")
    suspend fun getById(id: Long): Location?
```

```
    @Query("""
        SELECT * FROM locations
        WHERE TRIM(name) = TRIM(:name) COLLATE NOCASE
        AND ((:parentId IS NULL AND parentId IS NULL) OR parentId = :parentId)
        LIMIT 1
    """)
    suspend fun findByNameAndParent(name: String, parentId: Long?): Location?
}
```

A.3.4.2 ItemDao

```
@Dao
interface ItemDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(item: Item): Long

    @Update
    suspend fun update(item: Item)

    @Delete
    suspend fun delete(item: Item)

    @Query("SELECT * FROM items WHERE id = :id")
    fun getItemFromId(id: Long): Flow<Item>

    @Query("SELECT * FROM items WHERE id = :id LIMIT 1")
    suspend fun getById(id: Long): Item?

    @Query("SELECT locationId FROM items WHERE name = :name")
    fun getItemFromName(name: String): List<Long>

    @Query("SELECT * FROM items ORDER BY name ASC")
    fun getAllItems(): Flow<List<Item>>

    @Query("SELECT * FROM items WHERE locationId = :locationId ORDER BY name ASC")
    fun getItemsInLocation(locationId: Long): Flow<List<Item>>

    @Query("SELECT * FROM items WHERE name LIKE '%' || :searchQuery || '%'")
    fun searchItems(searchQuery: String): Flow<List<Item>>

    @Query("SELECT * FROM items WHERE name LIKE '%' || :name || '%' COLLATE NOCASE")
    suspend fun findByName(name: String): List<Item>

    @Query("SELECT * FROM items WHERE name LIKE '%' || :searchQuery || '%' ORDER BY name ASC")
    fun searchItemsLike(searchQuery: String): Flow<List<Item>>


    @Query("DELETE FROM items WHERE locationId IS NOT NULL AND locationId NOT IN (SELECT id FROM locations)")
    suspend fun deleteOrphanedItems(): Int
}
```

A.3.4.3 InventoryRepository

```
class InventoryRepository(private val itemDao: ItemDao, private val locationDao:
LocationDao) {

    // ITEMS ———————————————————————————————————————————————————————
    val allItems: Flow<List<Item>> = itemDao.getAllItems()

    suspend fun insertItem(item: Item) = itemDao.insert(item)
```

```
    suspend fun updateItem(item: Item) = itemDao.update(item)
    suspend fun deleteItem(item: Item) = itemDao.delete(item)

    fun getItemsInLocation(locationId: Long): Flow<List<Item>> =
itemDao.getItemsInLocation(locationId)
    fun searchItems(query: String): Flow<List<Item>> = itemDao.searchItems(query)

    /** Case-insensitive substring search — returns all matching Item rows. */
    suspend fun findItemsByName(name: String): List<Item> = itemDao.findByName(name)

    suspend fun getItemById(id: Long): Item? = itemDao.getById(id)

    // LOCATIONS ──────────────────────────────────────────────────────────
    val rootLocations: Flow<List<Location>> = locationDao.getRootLocations()

    fun getSubLocations(parentId: Long): Flow<List<Location>> =
locationDao.getSubLocations(parentId)
    fun getAllLocations(): Flow<List<Location>> = locationDao.getAllLocations()

    suspend fun insertLocation(location: Location): Long = locationDao.insert(location)
    suspend fun updateLocation(location: Location) = locationDao.update(location)
    suspend fun deleteLocation(location: Location) = locationDao.delete(location)

    /** Returns the first Location whose name matches exactly (case-insensitive). */
    suspend fun findLocationByName(name: String): Location? = locationDao.findByName(name)

    suspend fun findLocationByNameAndParent(name: String, parentId: Long?): Location? =
        locationDao.findByNameAndParent(name, parentId)

    suspend fun getLocationById(id: Long): Location? = locationDao.getById(id)

    /** Deletes items whose locationId points to a non-existent location (orphans). */
    suspend fun cleanupOrphanedItems(): Int {
        return itemDao.deleteOrphanedItems()
    }
}
```

A.3.5 - Type converter implementation

```
class Converters {
    // Shared JSON instance for Room list<->string conversions.
    private val json = Json { ignoreUnknownKeys = true }

    @TypeConverter
    // Room calls this when writing Item.attributes to SQLite.
    // We store the list as a JSON string in a single TEXT column.
    fun fromItemAttributes(value: List<ItemAttribute>?): String =
        json.encodeToString(value ?: emptyList())

    @TypeConverter
    // Room calls this when reading the TEXT column back into Item.attributes.
    fun toItemAttributes(value: String?): List<ItemAttribute> {
        // Empty/null column means "no attributes".
        if (value.isNullOrBlank()) return emptyList()
        return try {
            // Parse stored JSON into strongly-typed Kotlin objects.
            json.decodeFromString(value)
```

```
        } catch (e: Exception) {
            // Defensive fallback for malformed/legacy JSON to avoid app crashes.
            emptyList()
        }
    }
}
```

A.4

A.4.1 - Message Data Classes

```
data class MessageChoice(
    val itemId: Long,
    val label: String
)

data class Message(
    val text: String,
    val isUser: Boolean,
    val hasImageAttachment: Boolean = false,
    val choices: List<MessageChoice> = emptyList()
)
```

A.4.2 - Model State

```
sealed class ModelState {
    object Loading : ModelState()
    object Ready : ModelState()
    data class Failed(val msg: String) : ModelState()
}
```

A.4.3 - Message dispatch and thinking placeholder

```
fun sendMessage(userText: String, attachedImageBytes: ByteArray? = null) {
    Log.d(TAG, "sendMessage: chars=${userText.length},
hasImage=${attachedImageBytes != null}")
    val current = _messages.value.toMutableList()
    current += Message(userText, isUser = true, hasImageAttachment =
attachedImageBytes != null)
    current += Message("Thinking…", isUser = false)
    _messages.value = current

    viewModelScope.launch {
        val pending = pendingChoice
        if (pending != null) {
            val updated = _messages.value.toMutableList()
            if (updated.isNotEmpty()) updated.removeLast()
            updated += Message("Please choose an option using the selection
box.", isUser = false)
            _messages.value = updated
            Log.d(TAG, "sendMessage completed (choice flow)")
            return@launch
        }

        val raw = try {
            modelManager.generateResponse(promptManager.buildPrompt(userText))
        } catch (e: Exception) {
            Log.e(TAG, "Model generation failed", e)
            null
        }
        Log.d(TAG, "Raw model response: ${raw}")
        val response = raw?.let {
            val parsed = intentParser.parse(it)
            if (parsed != null) {
```

```
                Log.d(TAG, "Parsed intent: action=${parsed.action},
item=${parsed.item_name}")
                    val imageForAction = if (parsed.action == "add_item")
attachedImageBytes else null
                    val result = processAiResponse(parsed, imageForAction)
                Log.d(TAG, "Processed intent result
type=${result::class.simpleName}")
                toBotMessage(result)
            } else {
                Log.w(TAG, "Intent parser returned null")
                Message("I couldn't understand that action.", isUser = false)
            }
        } ?: Message("Something went wrong.", isUser = false)

        val updated = _messages.value.toMutableList()
        if (updated.isNotEmpty()) updated.removeLast() // remove Thinking…
        updated += response
        _messages.value = updated
        Log.d(TAG, "sendMessage completed")
    }
}
```

A.4.4 - Disambiguation Choice Handling

```
fun selectPendingChoice(selectedItemId: Long) {
    val pending = pendingChoice ?: return
    viewModelScope.launch {
        Log.d(TAG, "Choice clicked: itemId=$selectedItemId")
        val selectedName = pending.options.firstOrNull { it.id ==
selectedItemId }?.name
        pendingChoice = null
        _isChoicePending.value = false
        val result = jsonHand.processChosenItem(selectedItemId,
pending.actionToBePerformed)
        val botMessage = toBotMessage(result)
        val updated = _messages.value.toMutableList()
        removeLatestChoiceMessage(updated)
        updated += Message("Selected item: ${selectedName ?: selectedItemId}",
isUser = true)
        updated += botMessage
        _messages.value = updated
    }
}

fun cancelPendingChoice() {
    if (pendingChoice == null) return
    pendingChoice = null
    _isChoicePending.value = false

    val updated = _messages.value.toMutableList()
    removeLatestChoiceMessage(updated)
    _messages.value = updated
}
```

A.4.5 - Chat Fragment: Input Row and State Observation

```
btnSend.setOnClickListener {
    val text = editText.text.toString().trim()
    if (text.isNotBlank()) {
        val attachedImage = pendingImageAttachment
```

```
        pendingImageAttachment = null
        btnAttach.alpha = 0.75f
        editText.text.clear()
        hideKeyboard(editText)
        viewModel.sendMessage(text, attachedImage)
    }
}
viewLifecycleOwner.lifecycleScope.launch {
    viewModel.modelState.collect { state ->
        when (state) {
            is ModelState.Loading -> {
                editText.hint = getString(R.string.hint_loading_ai)
                btnSend.isEnabled = false
                btnSend.alpha = 0.4f
            }
            is ModelState.Ready -> {
                editText.hint = getString(R.string.hint_type_here)
                btnSend.isEnabled = true
                btnSend.alpha = 1f
            }
            is ModelState.Failed -> {
                editText.hint = "AI failed to load"
                btnSend.isEnabled = false
                btnSend.alpha = 0.4f
            }
        }
    }
}
viewLifecycleOwner.lifecycleScope.launch {
    viewModel.isChoicePending.collect { isPending ->
        val enabled = !isPending
        editText.isEnabled = enabled
        btnSend.isEnabled = enabled && viewModel.modelState.value is
ModelState.Ready
        btnAttach.isEnabled = enabled
        editText.alpha = if (enabled) 1f else 0.5f
        btnAttach.alpha = if (enabled) btnAttach.alpha else 0.35f
        if (!enabled) {
            editText.hint = getString(R.string.hint_complete_selection)
        } else if (viewModel.modelState.value is ModelState.Ready) {
            editText.hint = getString(R.string.hint_type_here)
        }
    }
}
viewLifecycleOwner.lifecycleScope.launch {
    viewModel.messages.collect { messages ->
        val prevSize = displayedMessages.size
        displayedMessages.clear()
        displayedMessages.addAll(messages)
        if (messages.size > prevSize) {
            chatAdapter.notifyItemRangeInserted(prevSize, messages.size -
prevSize)
        } else {
            chatAdapter.notifyDataSetChanged()
        }
        if (displayedMessages.isNotEmpty()) {
            recycler.scrollToPosition(displayedMessages.size - 1)
        }
    }
}
```

A.4.6 - Camera Capture

```
private val takePicturePreview =
registerForActivityResult(ActivityResultContracts.TakePicturePreview()) {
bitmap ->
    if (bitmap == null) {
        Toast.makeText(requireContext(),
getString(R.string.photo_capture_failed), Toast.LENGTH_SHORT).show()
        return@registerForActivityResult
    }

    pendingImageAttachment = bitmap.toJpegBytes()
    Toast.makeText(requireContext(), getString(R.string.photo_attached),
Toast.LENGTH_SHORT).show()
    view?.findViewById<ImageView>(R.id.buttonAttachPhoto)?.alpha = 1f
}
private fun Bitmap.toJpegBytes(quality: Int = 98): ByteArray {
    val stream = ByteArrayOutputStream()
    compress(Bitmap.CompressFormat.JPEG, quality, stream)
    return stream.toByteArray()
}
```

Appendix B: User Manual

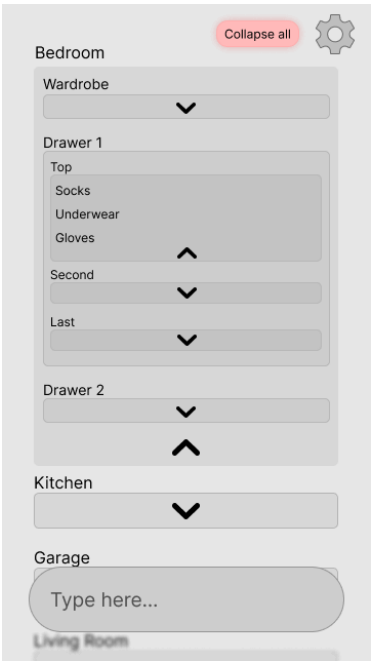
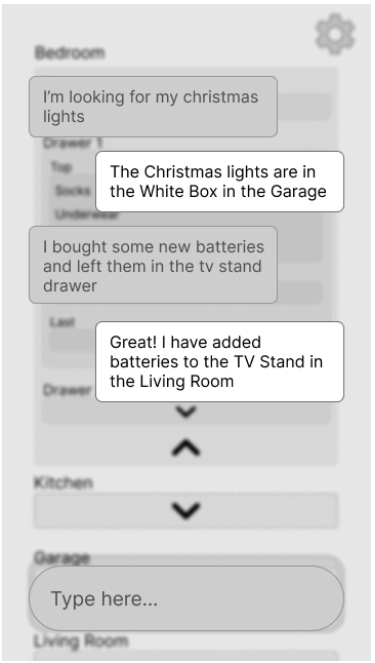
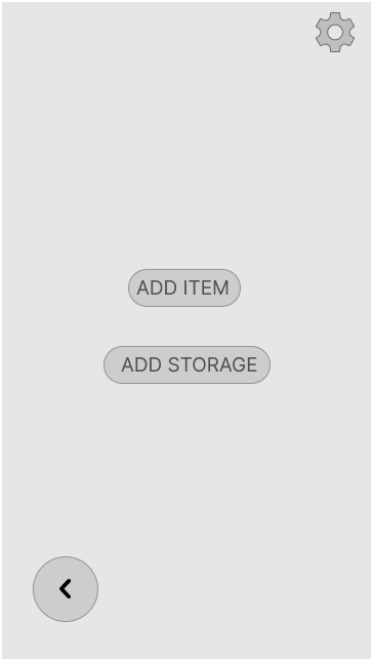
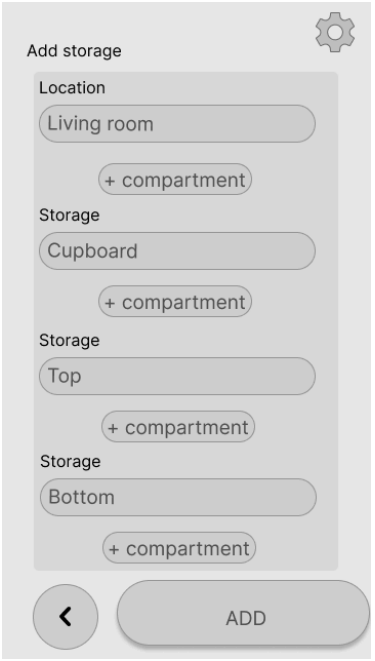
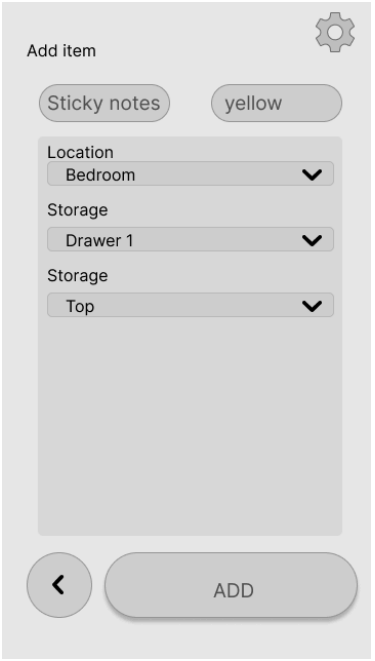
This User Manual provides step-by-step instructions for installing, configuring, and operating the system. It is intended for end users and administrators to ensure correct usage, troubleshooting, and maintenance.

User Manual Link:

[Inventory_ChatBot_User_Manual_v1.0.pdf]

Appendix C: UI/UX Prototype

C.1	C.2	C.3

<table>
<tr><td> 
</td><td> 
</td><td> 
</td></tr>
<tr><td>C.4</td><td>C.5</td><td>C.6</td></tr>
<tr><td> 
</td><td> 
</td><td> 
</td></tr>
</table>

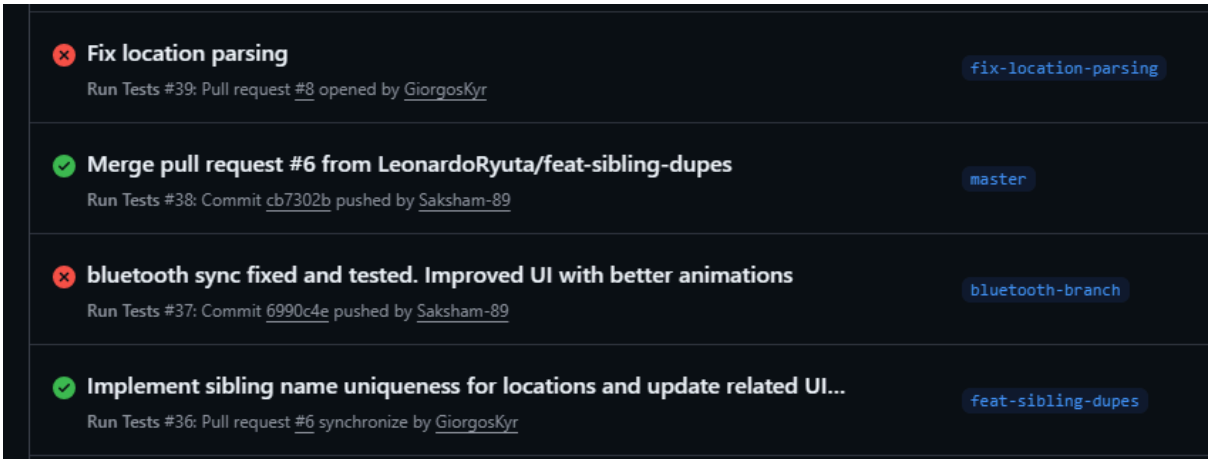
Appendix D: Requirements Traceability Matrix

Requirement ID	Priority	Description (short)	Design Section	Test Case(s)
FR-L1	H	Define/edit/remove hierarchical locations	4.1	TC-F01
FR-L2	H	No duplicate sibling location names	4.1.3	TC-F02
FR-C1	H	Chatbot interface	4.2	TC-F03, TC-F04
FR-C2	H	Register items via chatbot	4.4.1	TC-F03
FR-C3	H	Find items via chatbot	4.2, 4.3	TC-F04
FR-C4	H	Multiple results display	4.2.1	TC-F05
FR-C5	H	Ambiguity clarification	4.3.3	TC-F06
FR-I3	M	Photo attachments	4.1.2	TC-F07
FR-S1	M	Inventory sharing	4.5	TC-F08
NFR-1	H	100% offline operation	3.1	TC-P01, TC-S01
SR-3	H	No external data transmission	3.1	TC-S01

Appendix E: Continuous Integration Pipeline Evidence

E.1

E.1.1 - GitHub Actions workflow run overview



E.1.2 - Example test artifact generated by the CI pipeline

Class org.example.offlineinventory.data.merger.JsonHandlerTest

all > org.example.offlineinventory.data.merger > JsonHandlerTest

tests	failures	ignored	duration	successful
14	0	0	3.395s	100%

- Tests

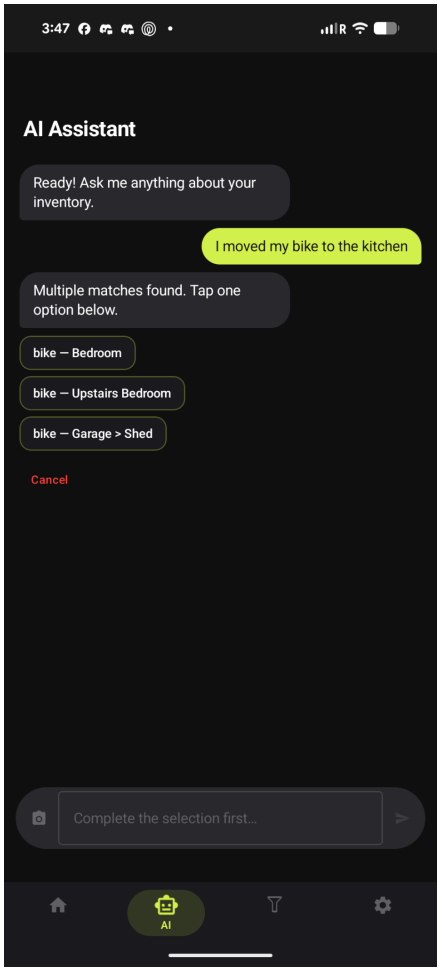
Tests

Test	Duration	Result
addItem adds two different items to the same location	0.038s	passed
addItem creates separate entries for items with duplicate names	0.025s	passed
addItem returns LocationNotFound when path is broken	0.019s	passed
addItem succeeds with nested location path	0.114s	passed
addItem succeeds with simple single location	0.018s	passed
process returns Error for unknown action	0.009s	passed
process returns Error when item_name is missing	0.011s	passed
remove item items	0.023s	passed
removeItem removes an item that has a nested location	0.090s	passed
resolveLocationPath succeeds using hierarchy when order is different	0.023s	passed
resolveLocationPath uses tie-breaker for ambiguous hierarchy matches	0.026s	passed
retrieveItem builds path for match in single location	0.017s	passed
retrieveItem nested location path for matches	2.888s	passed
updateItem test	0.094s	passed

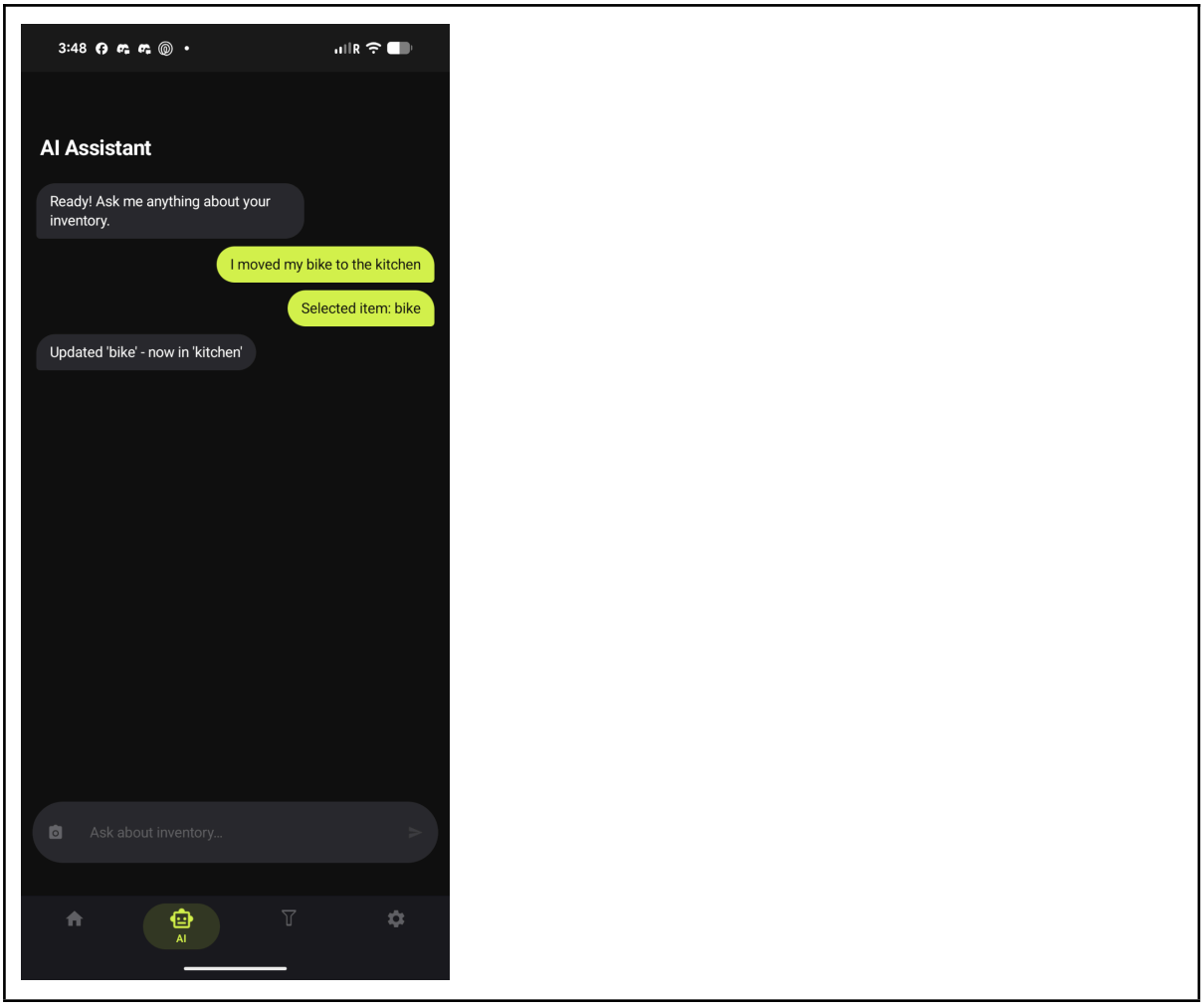
Wrap lines ☐
Generated by Gradle 8.13 at Apr 13, 2026, 9:08:37 PM

Appendix F: Ambiguity Example

F.1 User asks for an item included in multiple locations so chatbot asks for clarification



F.2 User selects an option and chatbot proceeds with intent


3:48
AI Assistant
Ready! Ask me anything about your inventory.
I moved my bike to the kitchen
Selected item: bike
Updated 'bike' - now in 'kitchen'
Ask about inventory...
AI