POLITEHNICA UNIVERSITY OF BUCHAREST

SOFTWARE ENGINEERING

PROJECT

Software Requirements Specification

Team:

Sakka Mohamad-Mario

Coordinator:

As. Alexandru IOSUP

Date created:

19.10.2025

DELIVERY REPORT

Name	Group	Project implementation [%, reason]	Signature
Sakka Mohamad-Mario	1241EB	____________________ ____________________ ____________________	

Delivery date:

____

Table of Contents

REQUIREMENTS ANALYSIS

According to the IEEE STD-830-1993, *IEEE Recommended Practice for Software Requirements Specification*.

1. Introduction

1.1 Purpose

This document describes what the **RecipeGPT** system must do and the constraints it must meet. It covers the Android client (Kotlin) and the ExpressJS backend deployed on **Google Cloud Platform (GCP)** using **Cloud Run** and **Artifact Registry**, with automated tests and deployments on every commit.

1.2 History

RecipeGPT helps you ask an AI for recipes, check whether you can cook them with what you already have, and build a shopping list for the rest. It also sprinkles in short cooking quotes to keep things fun.

1.3 Scope

- **Android app:** generate/search recipes, view details and steps, save favorites, manage **inventory** and **shopping list**, foreground notifications during long requests, periodic quote notifications.

- **Express backend:** validates input, calls OpenAI, normalizes responses, and returns JSON over HTTPS. It runs as a **Docker container on Cloud Run**.

- **CI/CD:** each commit triggers tests; on success, the container image is built and pushed to **Artifact Registry**, then a **Cloud Run** revision is deployed. **Secrets are stored as GitHub Actions Secrets** and injected into container environment variables during the workflow.

- **Local data:** Room DB; SharedPreferences for small settings.

1.4 Definitions

- **Cloud Run:** GCP's serverless containers.
- **Artifact Registry:** private container registry on GCP.
- **GitHub Actions Secrets:** encrypted repository/org secrets referenced in workflows and injected as environment variables at deploy time.
- **LLM:** Large Language Model (OpenAI models).
- **Room:** Android local database library.
- **UoM:** Unit of Measure (g, ml, pcs).
- **FR/NFR:** Functional/Non-Functional Requirement.

1.5 References

- Android docs (Room, WorkManager, Notifications).
- Node.js / ExpressJS docs.
- GCP docs (Cloud Run, Artifact Registry, Cloud Build/Deploy).
- GitHub Actions docs (workflows, secrets).

1.6 Structure

Section 1 – Introduction: States the purpose and scope of **RecipeGPT**, the intended users, and the context in which the Android app and the ExpressJS backend operate. It summarizes the GCP deployment (Docker container on Cloud Run, image in Artifact Registry) and the CI/CD approach (GitHub Actions running tests, building, and deploying; secrets provided via **GitHub Actions Secrets** as environment variables).

Section 2 – General Description: Presents a high-level picture of the system: main features (AI recipe generation, inventory, shopping list, cooked/listed flows), user types (everyday cooks and planners), and the operational environment (Android 8.0+, HTTPS to the Cloud Run API). It also lists key constraints, assumptions, and dependencies, and outlines common usage scenarios from "search recipe" to "cook" and "shop".

Section 3 – System Requirements: Details what the system must do and how it should behave.

- *External Interface Requirements* describe the mobile UI screens and the HTTPS JSON APIs between the app and the backend.
- *Functional Requirements* cover flows like generating recipes, viewing details, saving/deleting, inventory CRUD, cooked/listed actions, shopping list acquisition, preferences, notifications, and periodic quotes.
- *Non-Functional Requirements* define performance, reliability, security (HTTPS only; no secrets in the APK; secrets injected from GitHub Actions), usability/accessibility, maintainability, observability, and cost awareness.
- *Design Constraints* note the chosen technologies (Kotlin/Room/WorkManager on Android; Express on Cloud Run; Docker; Artifact Registry; GitHub Actions for CI/CD).

Appendices: Provide supporting materials that complement the main text: PlantUML diagrams (component, deployment, use-case, class/ER, sequence, activity, and state), interview notes that informed requirements, document evolution log, brief meeting reports, and conclusions. The appendices also include a simple traceability view linking requirements to design elements and test cases.

---

2. General Description

2.1. Product Description

RecipeGPT is a mobile application (Android, Kotlin) that helps users generate cooking recipes with the aid of an AI model, then guides them from “idea” to “plate.” The app sends a free-text prompt (e.g., “chocolate cake,” “quick vegetarian dinner”) to a lightweight ExpressJS backend hosted on Google Cloud Run. The backend formats the AI response into structured JSON (title, ingredients with quantities and units, and step-by-step instructions) and returns it to the phone.
On the device, users can save recipes for offline access, maintain a local **inventory** of ingredients, and build a **shopping list** with the exact shortages for the recipes they intend to cook. A small preferences area controls how many recipes to generate per request and how often to fetch short cooking quotes. The system is designed for quick everyday use (students, busy people) and works even without network access for already-saved content and the inventory/shopping list.

2.2. Product Functions

- **AI recipe generation:** Send a prompt and receive N structured recipe candidates (title, ingredients with units/amounts, steps).
- **Recipe browsing & details:** View lists of generated/saved recipes; open details with clear steps and ingredient quantities.
- **Save / delete / share:** Save recipes to local storage (Room); delete when no longer needed; share a recipe as clean text.
- **Inventory management:** Add, edit, or remove ingredients you currently have (name, amount, unit).
- **Sufficiency indicators:** In the recipe details screen, each ingredient shows **green** if you have enough and **red** if you don't.
- **Cooked action:** Deduct the used quantities from the inventory when you mark a recipe as cooked (never below zero).
- **Shopping list:** Mark a recipe as **listed** to automatically add only the missing quantities to the shopping list; mark items **acquired** to restock inventory.
- **Preferences & quotes:** Configure recipes-per-request and quote frequency; receive notifications for progress, results, and quotes.
- **Resilient operations:** Foreground notification (with elapsed time) while generating recipes; retries and user-friendly errors on timeouts.

2.3. User Description

- **Everyday cook / student:** Wants quick, practical ideas that match what's already in the kitchen and a simple way to buy the rest. Values clarity, low friction, and offline access to saved recipes.
- **Planner / power user:** Prepares multiple meals ahead, relies on exact quantities, and expects the shopping list and inventory to stay in sync. Appreciates predictable performance and reliable notifications.

2.4. Constraints

- **Security of secrets:** OpenAI and deployment credentials are **not** bundled with the app; they are stored as **GitHub Actions Secrets** and injected as environment variables during deployment to Cloud Run.

- **Platform:** Android 8.0 (API 26) or higher; uses Room, WorkManager, Notifications, and Foreground Services per Android policies.
- **Networking:** All mobile ↔ backend traffic is HTTPS (TLS 1.2+).
- **Server limits:** Cloud Run request timeouts and concurrency apply; recipe count per request is capped to control latency/cost.
- **Data locality:** User data (saved recipes, inventory, shopping list) is stored on-device; no mandatory account or cloud sync in the MVP.
- **Unit handling:** Only basic unit normalization is guaranteed initially (e.g., g↔kg, ml↔l; pieces remain pieces).

2.5. Assumptions and Dependencies

- **External services:** The backend depends on OpenAI API availability and reasonable latency; costs scale with usage and recipe count.
- **Cloud infrastructure:** Google Cloud Run is available and reachable; images are built and pushed to **Artifact Registry** via GitHub Actions; deployments succeed when tests pass.
- **Device resources:** Users have sufficient local storage for Room DB; intermittent connectivity is expected, but the app must remain useful offline for saved data.
- **Usage patterns:** Most prompts are short and recipe count per request is low (1–3), keeping response times acceptable.
- **Future enhancements:** More robust unit conversion, dietary filters, and optional cloud sync may be added later without breaking the current data model or APIs.

3. Detailed Features (Functional Requirements)

3.1. External Interface Requirements

User Interfaces (Android):

- **Home screen:** search bar, list of generated/saved recipes (card layout), “Details” action, visible loading state.
- **Recipe Details:** title, description (if any), **Ingredients** (name, amount, unit) with red/green sufficiency indicator, **Steps**, actions: **Save/Delete**, **Listed**, **Cooked**, **Share**.
- **Saved Recipes:** list with delete; tap → details.
- **Inventory:** table-like list; add/edit/delete items (name, amount, unit).
- **Shopping List:** shows shortages; edit amounts; **Acquire** to move amounts into Inventory.
- **Settings:** recipes per request (integer ≥1), quote frequency (preset intervals).
- **Notifications:** persistent “in progress” notification (elapsed time) while generating; standard notifications for results, quotes, and errors.
- **Accessibility:** content descriptions for actionable elements, color + text labels for sufficiency (“Enough/Not enough”), minimum 48dp touch targets.

Software Interfaces:

- **Mobile ↔ Backend (HTTPS/JSON):**
    - POST /api/recipes/generate → body { query: string, count: number } → { recipes: RecipeDTO[] }
    - GET /api/quotes/random → { quote: string, author?: string }
    - Standard headers: Content-Type: application/json; optional User-Agent and client version.
- **Android platform APIs:** Room (SQLite), WorkManager (periodic quotes), Foreground Service + Notifications (long-running fetch), SharedPreferences, BroadcastReceivers, Intents (share via ACTION_SEND).
- **Permissions:** normal network access; no sensitive runtime permissions required.

Hardware Interfaces: none beyond a standard Android device (no special sensors/peripherals).

Communications: TLS 1.2+; JSON UTF-8; client request timeout aligned with Cloud Run (e.g., 30–60s). Retries with backoff on idempotent calls.

3.2. Functional Requirements

Recipe Generation & Search

- **FR-01** Users can enter a free-text query and submit.
- **FR-02** App sends {query, count} (count from Settings) to backend.
- **FR-03** While waiting, show a foreground notification with **elapsed time**.
- **FR-04** Display a list of count recipes (title + short blurb).
- **FR-05** On error/timeout, show a readable message and **Retry**.

Recipe Details & Persistence

- **FR-06** Details show ingredients (name, amount, unit) and numbered steps.
- **FR-07** For each ingredient, show **green** if inventory has enough, **red** otherwise.
- **FR-08** Save/delete recipes locally (Room); persist across app restarts.
- **FR-09** Share a recipe as clean text via Android share sheet.

Inventory & Shopping

- **FR-10** Inventory CRUD (name, amount, unit).
- **FR-11 Cooked** deducts required amounts from inventory; never below zero (clamp).
- **FR-12 Listed** adds only the **shortages** (required − available, min 0) to the shopping list.
- **FR-13** Edit shopping list item amounts.
- **FR-14 Acquire** moves amounts from Shopping List into Inventory and removes/adjusts items.

Preferences, Quotes, Sync

- **FR-15** Set **recipes per request** (integer ≥1).
- **FR-16** Set **quote frequency** (preset intervals).

- **FR-17** Unique periodic WorkManager task fetches quotes; changing preferences reschedules it.
- **FR-18** DB changes broadcast to subscribed fragments; UIs refresh.
- **FR-19** ViewModels preserve state across configuration changes.
- **FR-20** All client–server traffic uses **HTTPS**.

3.3. Performance Requirements

- **PR-01 (Generation latency):** For count ≤ 3, 95% of successful generations complete in **≤ 30 s** under normal backend load.
- **PR-02 (UI responsiveness):** Local UI actions (open details, list scrolling, inventory edit) respond in **≤ 100 ms** on mid-range devices.
- **PR-03 (Cold start tolerance):** First request after Cloud Run cold start may add latency; the app must maintain progress feedback until completion.
- **PR-04 (Background work):** Quote retrieval finishes within **≤ 5 s** when the backend is healthy; failures back off exponentially.
- **PR-05 (App footprint):** APK ≤ **50 MB** (guideline); foreground service CPU usage minimal while waiting (timer tick ≤ 1 Hz).

3.4. Design Constraints

- **Platforms/versions:** Android **API 26+**; server packaged as **Docker** container on **GCP Cloud Run**; images in **Artifact Registry**.
- **Tech stack (fixed):** Kotlin (Android), Room, WorkManager, Retrofit/OkHttp; Node.js 18+ / Express on the server.
- **Secrets: GitHub Actions Secrets** hold credentials (e.g., OPENAI_API_KEY) and are injected as **environment variables** at deploy time. No secrets in the APK or source.
- **Networking/security:** HTTPS only; no direct calls from mobile to OpenAI; API key must never be returned to clients or logged.
- **Budget/quotas:** Limit count per request; set Cloud Run timeouts/concurrency to control cost and protect stability.
- **Data model:** Units are basic (g↔kg, ml↔l, pcs); complex conversions out of scope for v1.

3.5. Software System Attributes

- **Security & Privacy:** TLS 1.2+; least-privilege IAM for deployments; redact logs; store user data locally (no mandatory accounts); secrets via GitHub Actions only.
- **Reliability & Availability:** Graceful handling of timeouts/5xx with backoff; WorkManager survives reboots; CI ensures tests pass before deploy.
- **Maintainability:** Layered app (UI / ViewModel / Services / Repos / DAOs); backend split into routes/controllers/services; formatted/linted code; API schema documented.
- **Usability & Accessibility:** Clear language, large touch targets, content descriptions; color indicators accompanied by text labels.
- **Portability & Scalability:** Serverless container on Cloud Run; scale-to-zero acceptable for MVP; can raise min instances if latency becomes critical.
- **Observability:** Structured logs with request IDs; Cloud Monitoring dashboards for latency and error rates; health endpoint for deploy checks.
- **Internationalization (readiness):** Strings externalized; English baseline.

3.6. Other System Requirements

- **Deployment pipeline:** GitHub Actions → tests → build Docker → push to Artifact Registry → deploy Cloud Run revision; pass env vars from GitHub Secrets.
- **Health checks & rollbacks:** Deployment must verify health before shifting traffic; ability to revert to previous revision.
- **Data export (optional future):** Provide a simple local export/import for recipes, inventory, and shopping list (JSON/CSV) when added; not required for MVP.
- **Legal & compliance:** No unnecessary PII; comply with platform policies (Google Play, Android background execution limits).
- **Documentation:** Keep endpoint contracts and PlantUML diagrams in repo; update SRS history on significant changes.

APPENDICES

A1. Interview with the customer

Purpose: Capture real needs and pain points to shape the requirements.
Participants: Azzam, Layla, Abdelrahman
Format: 30-minute semi-structured interview.

Context & goals (what they want):
Stakeholders want an app that turns short, plain-language ideas into concrete, step-by-step recipes. It should immediately show whether the user can cook with what's already in the kitchen and, if not, build a precise shopping list with the missing quantities.

How they expect to use it:

- Search via **plain text** prompts (e.g., "chocolate cake", "quick vegetarian dinner").
- See a **small set** of strong options first, each with a short blurb.
- Work **locally** without accounts or cloud sync in the first version.
- Ingredient sufficiency should be **obvious** (green = enough, red = not enough).
- After cooking, the app should **automatically deduct** used ingredients.
- Security should be handled in CI/CD (no keys in the app); all traffic over **HTTPS**.
- Waiting **~30 seconds** is acceptable if a clear progress indicator with elapsed time is shown.

Pain points observed:

- Web recipes rarely match what users already have.
- Manual shopping lists are error-prone (missed amounts/units).
- Some apps hide progress, leading users to think the app is frozen.

Decisions taken from the interview:

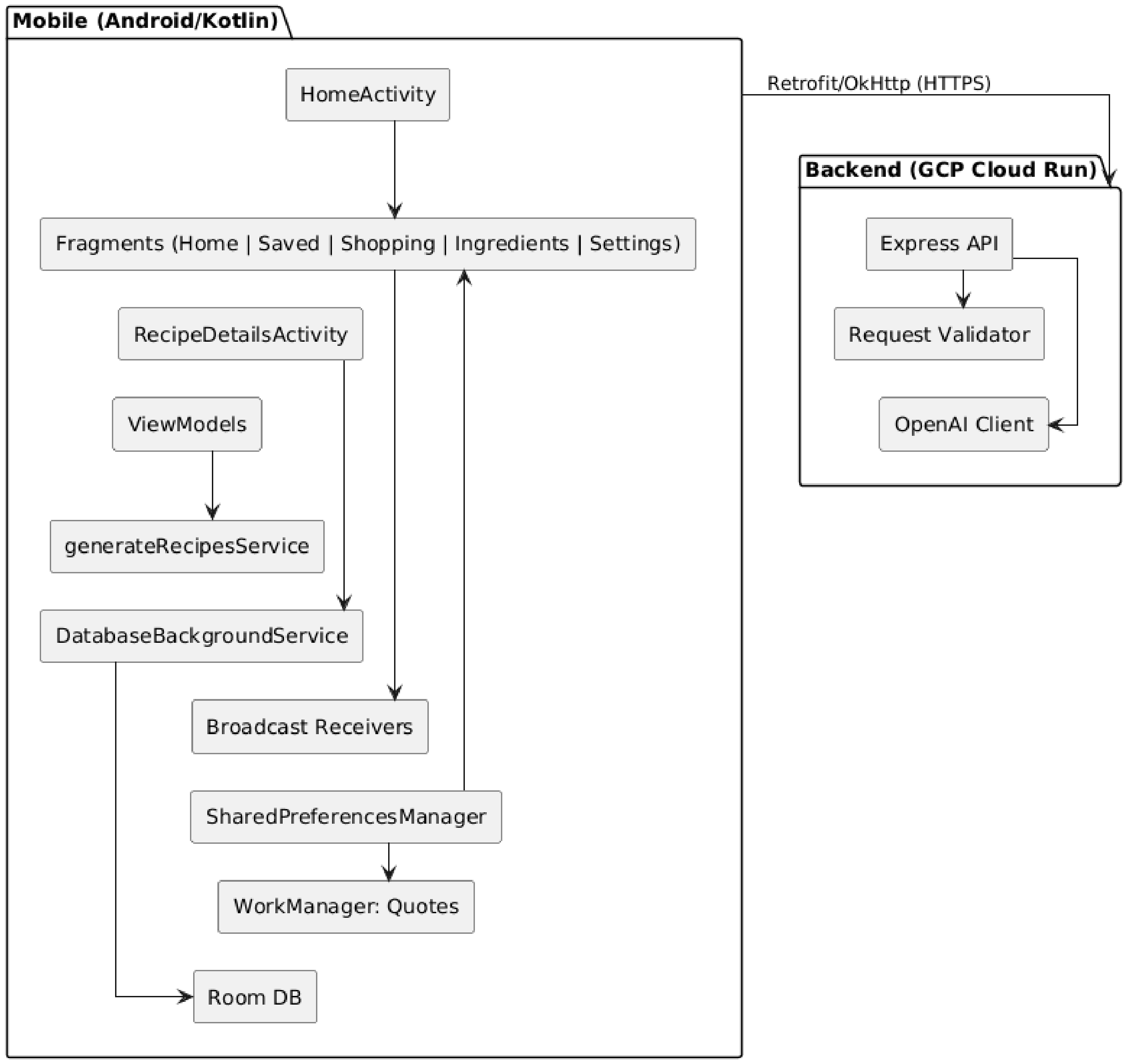
- Keep data on-device with **Room DB** (recipes, inventory, shopping list) for offline use.
- Use an **Express** backend on **GCP Cloud Run** to call OpenAI and return a **normalized JSON** schema.
- Manage secrets with **GitHub Actions Secrets** and inject them as **environment variables** at deploy; never ship secrets in the APK.

- Limit recipes per request to **1–3** to balance latency and cost; show a **foreground notification** with elapsed time during generation.

Success criteria (shared understanding):
A user types a food idea, sees ~3 solid options, and can either cook immediately (green indicators) or add the exact missing items to the shopping list in two taps.

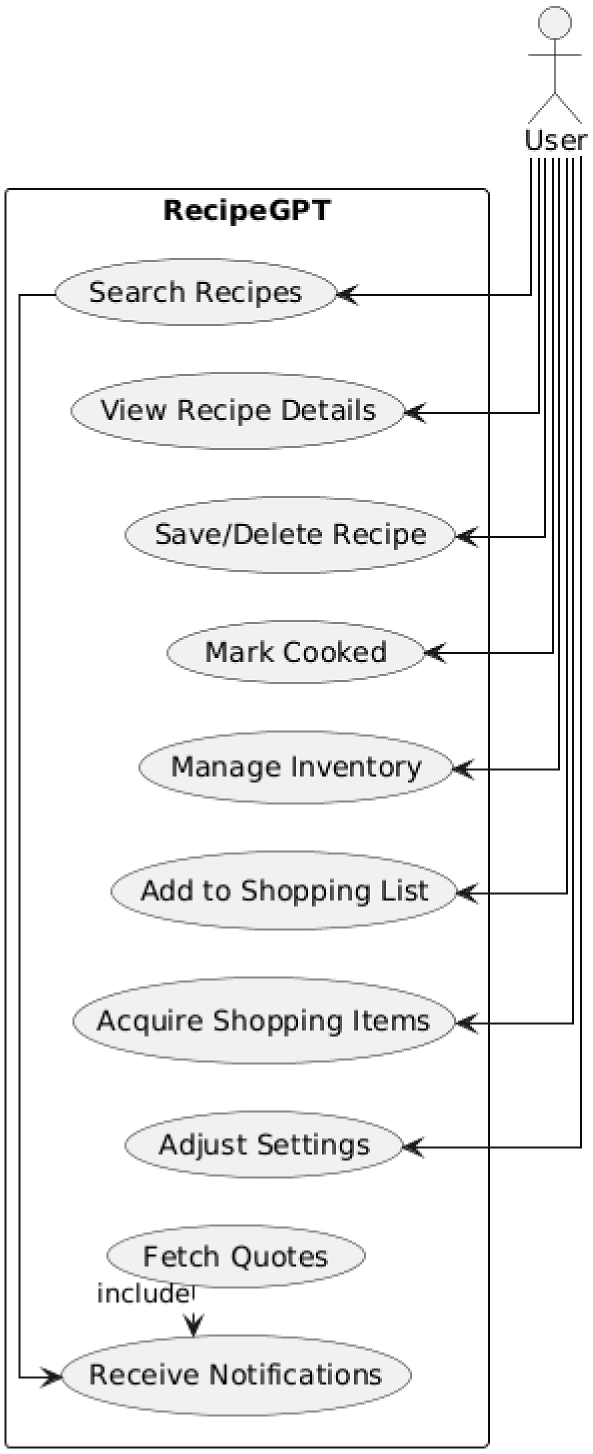
A2. System diagram

System diagram **Explanation.** The Android client owns UX and local data. The Express API on Cloud Run is a thin gateway to OpenAI and returns normalized JSON. Related components are stacked vertically to keep dependencies short and readable.

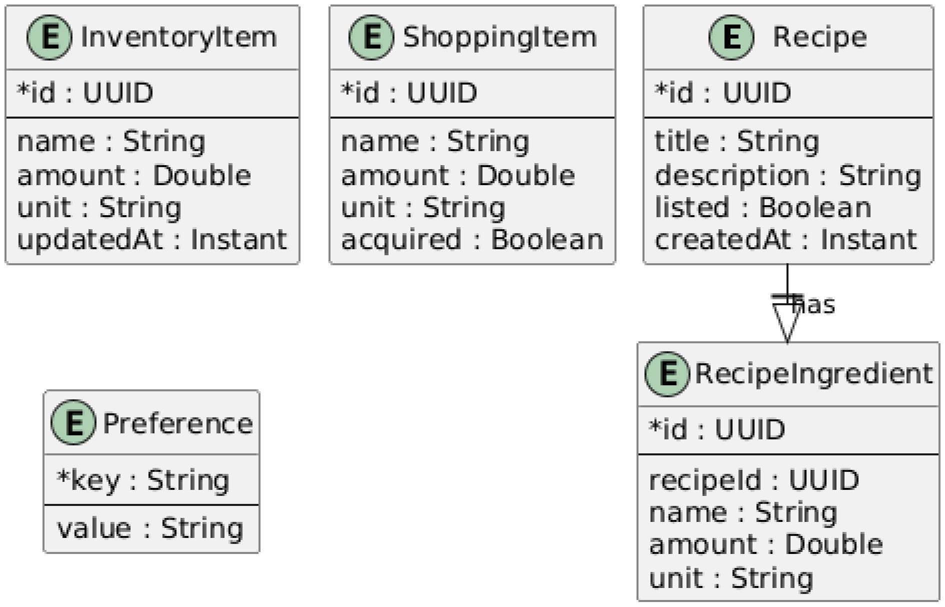


A3. Use Cases Diagrams

Explanation. This diagram captures what the user actually *does*; it's the backbone for test cases and acceptance criteria.



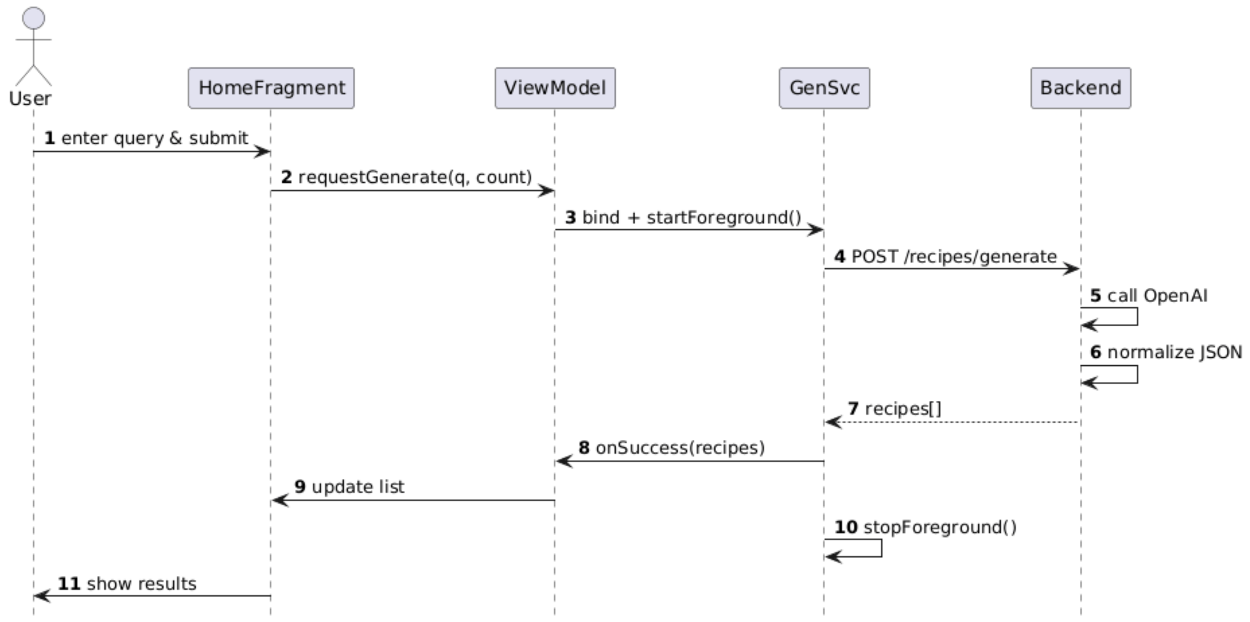
A4. Class Diagrams



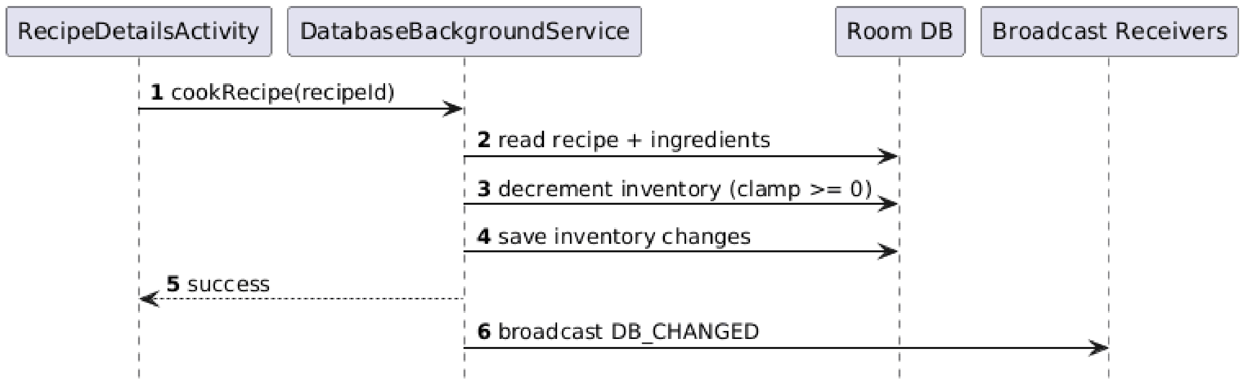
A5. Sequence Diagrams

Explanation. These show message order and responsibilities across UI, services, and DB.

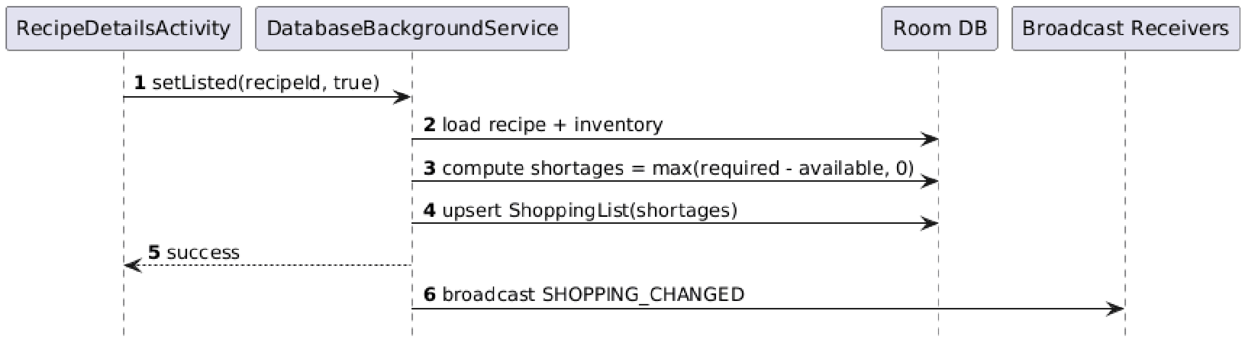
Generate Recipes



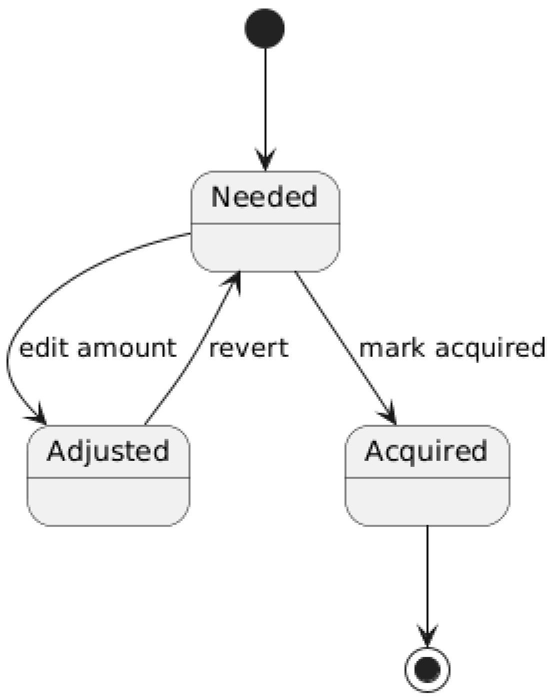
Cooked Action



Listed → Shopping


RecipeDetailsActivity
DatabaseBackgroundService
Room DB
Broadcast Receivers
1 setListed(recipeId, true)
2 load recipe + inventory
3 compute shortages = max(required - available, 0)
4 upsert ShoppingList(shortages)
5 success
6 broadcast SHOPPING_CHANGED

A6. State Diagrams



A7. Document Evolution

- As features evolve (dietary filters, multi-profile, sync), update Sections 2–3 and append new diagrams here. Record each revision in **1.2 History**.

A8. Report regarding team meetings

This project was developed **solo**. I first ran a **personal brainstorming session**, then held a **short feedback interview** with two colleagues to validate assumptions. The notes below capture how I shaped and refined the requirements.

Session S1 — Solo Brainstorming (ideas → scope)
Format: Rapid ideation, clustering, quick prioritization.

Highlights / raw ideas

- Plain-text AI prompts (no complex filters).
- Red/green ingredient sufficiency on the details screen.
- One-tap **Cooked** to decrement inventory.
- **Listed** flag adds **only shortages** to Shopping List.
- Save/share recipes; offline access for saved data and inventory.
- Keep secrets out of the APK; backend as Docker on GCP Cloud Run; inject env vars from **GitHub Actions Secrets**.

Decisions distilled

- **MVP:** generate recipes (1–3/request), details, save/delete/share, inventory CRUD, cooked/listed, shopping list, quotes.
- Android first (API 26+); Room for persistence; WorkManager for quotes.
- ExpressJS on Cloud Run; image in Artifact Registry; CI/CD: tests → build → push → deploy.
- HTTPS only; no accounts/sync in MVP.

Action items

- Draft SRS v1.0, design Room entities/services, and create initial PlantUML diagrams.

Session S2 — Colleague Interview (validation)
Participants: abd (interviewer), **Azzam** and **Layla** (colleagues / representative users)

Key prompts & findings

- *Details page content?* → Ingredients (name, amount, unit) + numbered steps.
- *Sufficiency indicator?* → Color **and** text ("Enough / Not enough").
- *After cooking?* → Auto-decrement inventory; never below zero.
- *Shopping behavior?* → Add only shortages; quick edit; **Acquire** to restock inventory.
- *Waiting tolerance?* → ~30s if a clear progress/elapsed-time notification is shown.
- *Privacy/security?* → Keep data on device; HTTPS; secrets via GitHub Actions.

Outcomes

- Finalized **Cooked**, **Listed**, and **Acquire** flows.
- Kept default recipes-per-request to **1–3** for latency/cost.
- Reaffirmed **no login/sync** for MVP; focus on reliable local UX.

A9. Conclusions regarding the activity

RecipeGPT's MVP focuses on fast AI recipe generation, transparent ingredient sufficiency, and a tight inventory ↔ shopping list loop. Deploying the Express API as a Docker container on **Cloud Run** keeps ops simple and costs predictable. Using **GitHub Actions** for tests, build, and deploy—with env vars injected from **GitHub Actions Secrets**—keeps secrets off devices and ensures repeatable releases. The system is ready for iterative growth (dietary filters, stronger unit conversions, optional sync) without breaking the current data model or APIs. Conclusions regarding the activity - The current MVP balances **cost** (Cloud Run serverless) and **security** (no device secrets). Inventory + shopping list provide clear, testable value. CI/CD ensures repeatable releases.