Salt Spreader Companion App

This document is the technical handoff, operator guide, and architectural reference for the Android companion app in `app/`. It is written to help developers, testers, operators, and reviewers understand the system architecture, communication contracts, operator flow, and build or troubleshooting steps.

1. Purpose

The app is an Android companion for a salt spreading robot based on an Arduino UNO R4 WiFi platform. The app has four primary jobs:

1. Provision the robot onto Wi-Fi over BLE.
2. Poll and display robot status over HTTP.
3. Stream live telemetry over TCP.
4. Define mission polygons, generate autonomous mission maps, and upload missions. Manual drive is available for testing only.

The current app is organized around a shared `RobotViewModel` and a single `RobotRepository` so that all screens consume the same state and communication layer.

2. Build And Runtime Baseline

Platform

- Android app namespace: `com.example.saltspreader`
- Minimum SDK: `31`
- Target SDK: `36`
- Java compatibility: `11`
- Build features: `ViewBinding`

Key Libraries

- AndroidX Navigation
- AndroidX Lifecycle LiveData and ViewModel
- Retrofit + Gson
- OkHttp logging interceptor
- Material Components
- SwipeRefreshLayout

Android Permissions

- `INTERNET`
- `ACCESS_WIFI_STATE`
- `ACCESS_NETWORK_STATE`
- `ACCESS_COARSE_LOCATION`
- `ACCESS_FINE_LOCATION`
- `BLUETOOTH`
- `BLUETOOTH_ADMIN`
- `BLUETOOTH_SCAN` for Android 12+
- `BLUETOOTH_CONNECT` for Android 12+

Common Commands

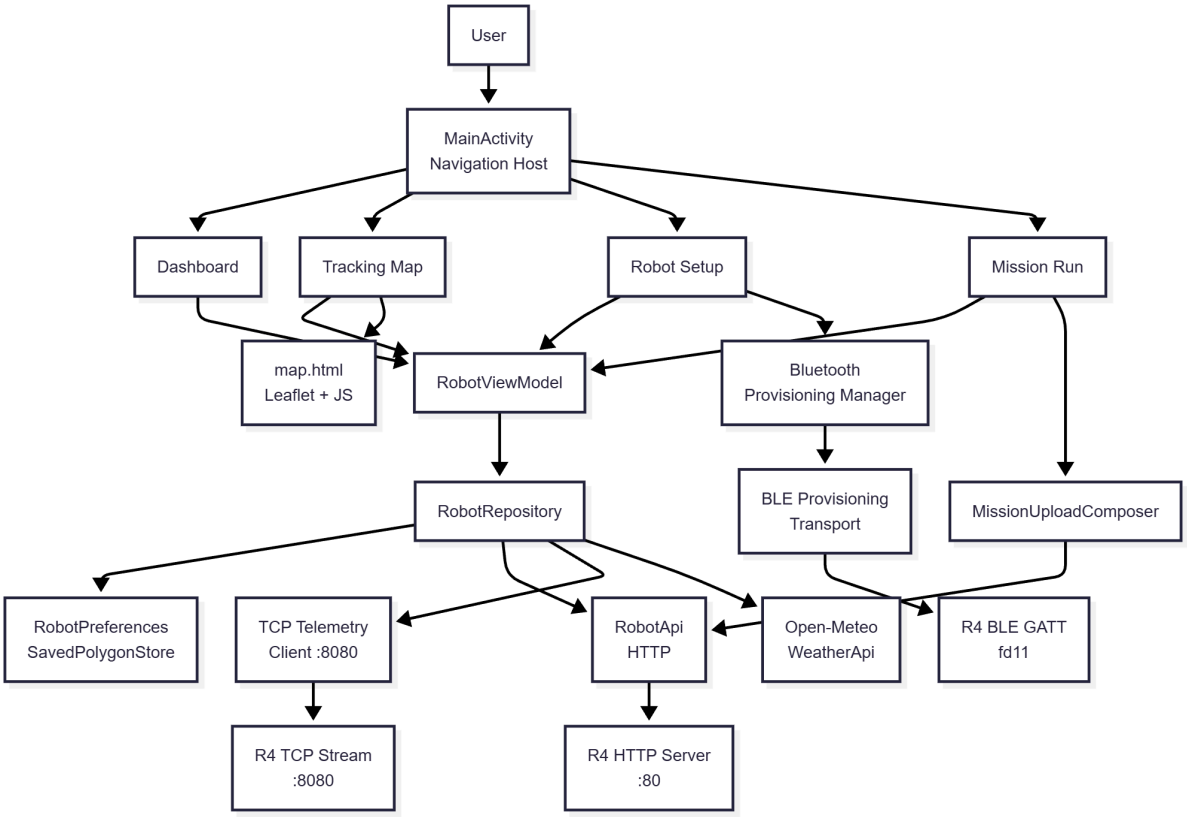
Set-Location e:\SaltSpreader

.\gradlew.bat :app:compileDebugJavaWithJavac

.\gradlew.bat assembleDebug

.\gradlew.bat testDebugUnitTest

3. High-Level Architecture



Architecture Notes

- `MainActivity` hosts the bottom navigation shell and starts periodic refresh through the shared view model.
- `RobotViewModel` is the single UI-facing state owner for all major screens.
- `RobotRepository` is the core communication hub. It owns HTTP polling, HTTP commands, mission uploads, TCP telemetry, weather enrichment, and app-side communication logging.
- BLE provisioning is intentionally separate from the repository because it is a connection setup workflow, not a long-lived robot control channel.
- The map and the service comms console are implemented as WebViews backed by local HTML assets plus Java/JavaScript bridges.

4. Package Map

Area	Main Classes	Responsibility
App shell	`MainActivity`	Navigation host, shared view model bootstrap
Shared state	`RobotViewModel`, `RobotViewModelFactory`	Exposes LiveData to all fragments
Repository	`RobotRepository`	HTTP, TCP telemetry, weather, unified comm logging
BLE setup	`BluetoothProvisioningManager`, `BleProvisioningTransport`, `ProvisioningPayloads`	BLE discovery, GATT connection, provisioning message flow
Networking	`RobotApi`, `DriveControlRequest`, `MissionUploadRequest`, `CommandRequest`	Retrofit contract and request models
Status and telemetry models	`RobotStatus`, `TelemetrySnapshot`, `ServerLogEntry`	Incoming robot data and app-side logs
Planning	`TrackingFragment`, `MissionRunFragment`, `SavedPolygonStore`, `PolygonGeoJsonUtils`, `MissionUploadComposer`	Polygon authoring, storage, mission upload preparation

Service and testing tools are documented separately in the Service And Testing Tools section.

5. Runtime Communication Surfaces

pie showData

title Runtime communication surfaces

"HTTP status and commands" : 40

"TCP telemetry stream" : 25

"BLE provisioning" : 15

"Map and JS bridge" : 10

"Weather enrichment" : 10

This chart is descriptive, not measured. It reflects where the app spends most of its communication complexity.

6. Communication With The R4 Robot

This section reflects the app behavior and the current R4 clean firmware expectations that the app is designed around.

6.1 Base URL

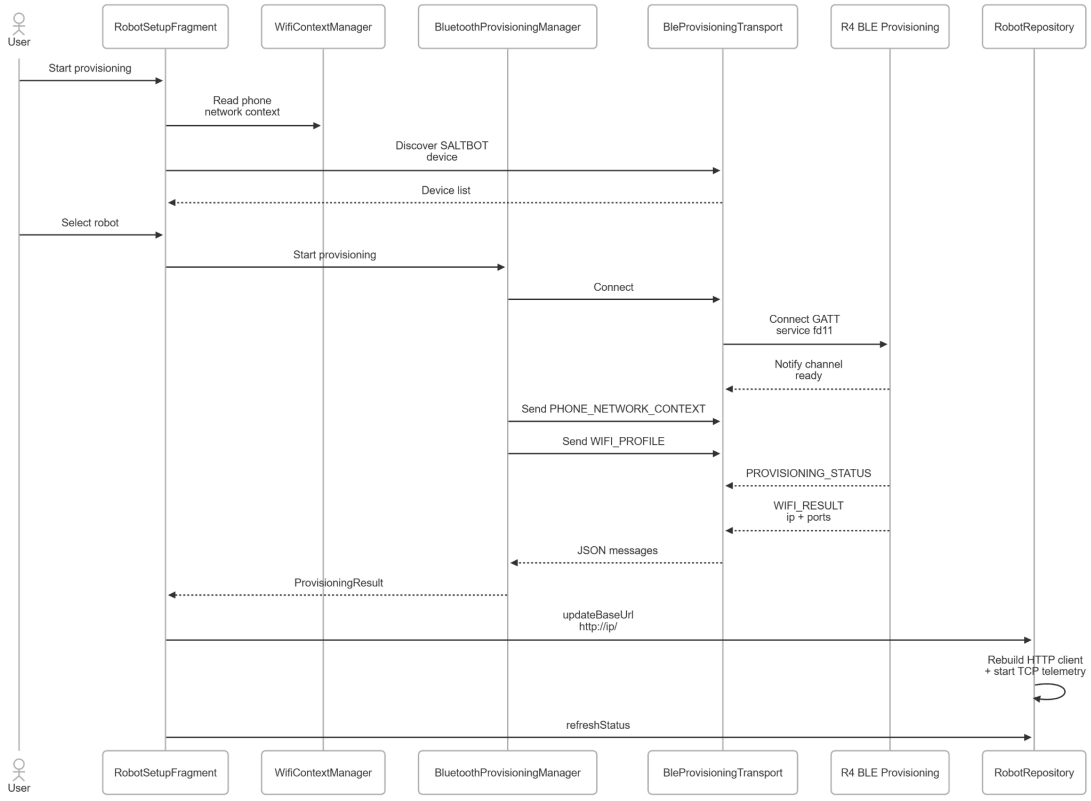
- Robot HTTP base URL format: `http://<robot-ip>/`
- Example: `http://172.20.10.3/`
- The app stores this in `RobotPreferences`.
- BLE provisioning automatically updates it when the robot sends `WIFI_RESULT.ip`.

6.2 BLE Provisioning Contract

BLE identity

- Name prefix: `SALTBOT`
- Service UUID: `0000fd11-0000-1000-8000-00805f9b34fb`
- RX characteristic UUID: `0000fd11-0001-1000-8000-00805f9b34fb`
- TX characteristic UUID: `0000fd11-0002-1000-8000-00805f9b34fb`
- Encoding: newline-delimited JSON messages

BLE provisioning sequence



App message order

1. Connect to GATT service.
2. Wait until notifications are enabled on TX.
3. Send `PHONE_NETWORK_CONTEXT`.
4. Send `WIFI_PROFILE`.
5. Wait for `PROVISIONING_STATUS` updates.
6. Consume `WIFI_RESULT` and update the HTTP base URL.

6.3 HTTP Contract

Method	Endpoint	App usage
`GET`	`/status`	Heartbeat, robot state, robot pose, UI status refresh
`POST`	`/control`	Legacy commands and direct drive control
`POST`	`/mission`	Upload polygon mission

`GET /status`

The app expects a JSON status document with robot identity, timing, current pose, battery, salt level, subsystem indicators, and optionally drive state. The current status model accepts multiple field shapes and aliases.

Important status fields currently consumed by the app:

- `robot_id`
- `timestamp_ms` or equivalent heartbeat timestamp
- `current_pose.lat`
- `current_pose.lng`
- `battery_pct`
- `salt_pct`
- `robot_mode`
- optional Wi-Fi strength and fault/event data

`POST /control`

Two modes are supported:

1. Legacy command request:

```
{ "command": "START" }
```

2. Direct drive request:

```
{
  "type": "DRIVE_CONTROL",
  "left_motor": 3500,
  "right_motor": 3500,
  "salt_spreader_motor": true,
  "salt_spreader_servo": false
}
```

Drive values are clamped in the app to `-10000..10000` before transmit.

`POST /mission`

The app uploads mission payloads in the form:


```
{
  "type": "MISSION_UPLOAD",
  "mission_id": "lot_a_1714850000000",
  "polygon": [
    { "lat": 42.0001, "lng": -93.0001 },
    { "lat": 42.0002, "lng": -93.0002 }
  ],
  "map_lines": [
    "####################",
    "###.....###########",
    "###.....###########",
    "####################"
  ],
  "resolution_m": 0.3,
  "origin_gps": { "lat_e7": 420001000, "lon_e7": -930001000 },
  "resolution_e7": 27,
  "start_gps": { "lat_e7": 420001500, "lon_e7": -930001500 },
  "autonomy": true,
  "grid_resolution_m": 0.3,
```

```
  "spreader_width_m": 0.5,

  "salt_mode": "NORMAL",

  "start_immediately": false

}
```

Notes:

- `map_lines` uses `.` for walkable cells and `#` for blocked cells.
- `resolution_m`, `origin_gps`, and `resolution_e7` define the grid alignment for the firmware.
- `start_gps` is included when the operator places a base station point in the polygon.
- The app still stores additional planning metadata locally (polygon names, base point placement).

6.4 TCP Telemetry Stream

- Host: same robot IP as HTTP
- Port: `8080`
- Format: line-delimited JSON
- Typical messages:
  - `ROBOT_HELLO`
  - `TELEMETRY`

The app now opens a TCP client from `RobotRepository` when a valid base URL is configured. Parsed telemetry is published as `TelemetrySnapshot` and shown in the service comms console.

6.5 Weather Enrichment

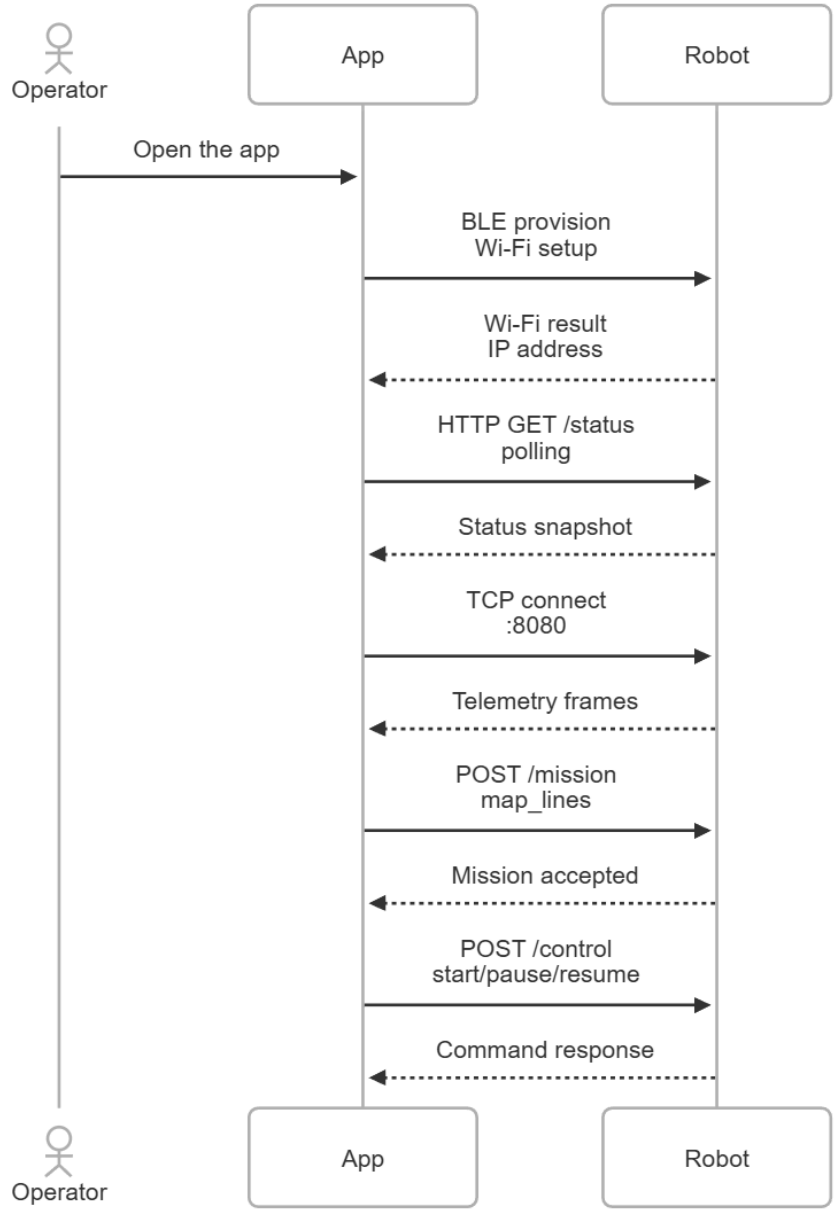
The app also talks to Open-Meteo. This is auxiliary and not part of robot control. Weather is used only to enrich the dashboard.

6.6 Demo Mode

Demo mode is a local simulation layer meant for screenshots, walkthroughs, and operator training without hardware.

- Toggle it from the service comms console.
- The repository seeds demo status, telemetry, and comms logs.
- Network calls are suppressed while demo mode is enabled.

6.7 Robot-App Connection And Communication



7. Data And State Model

Shared runtime state

The following LiveData streams are central to the app:

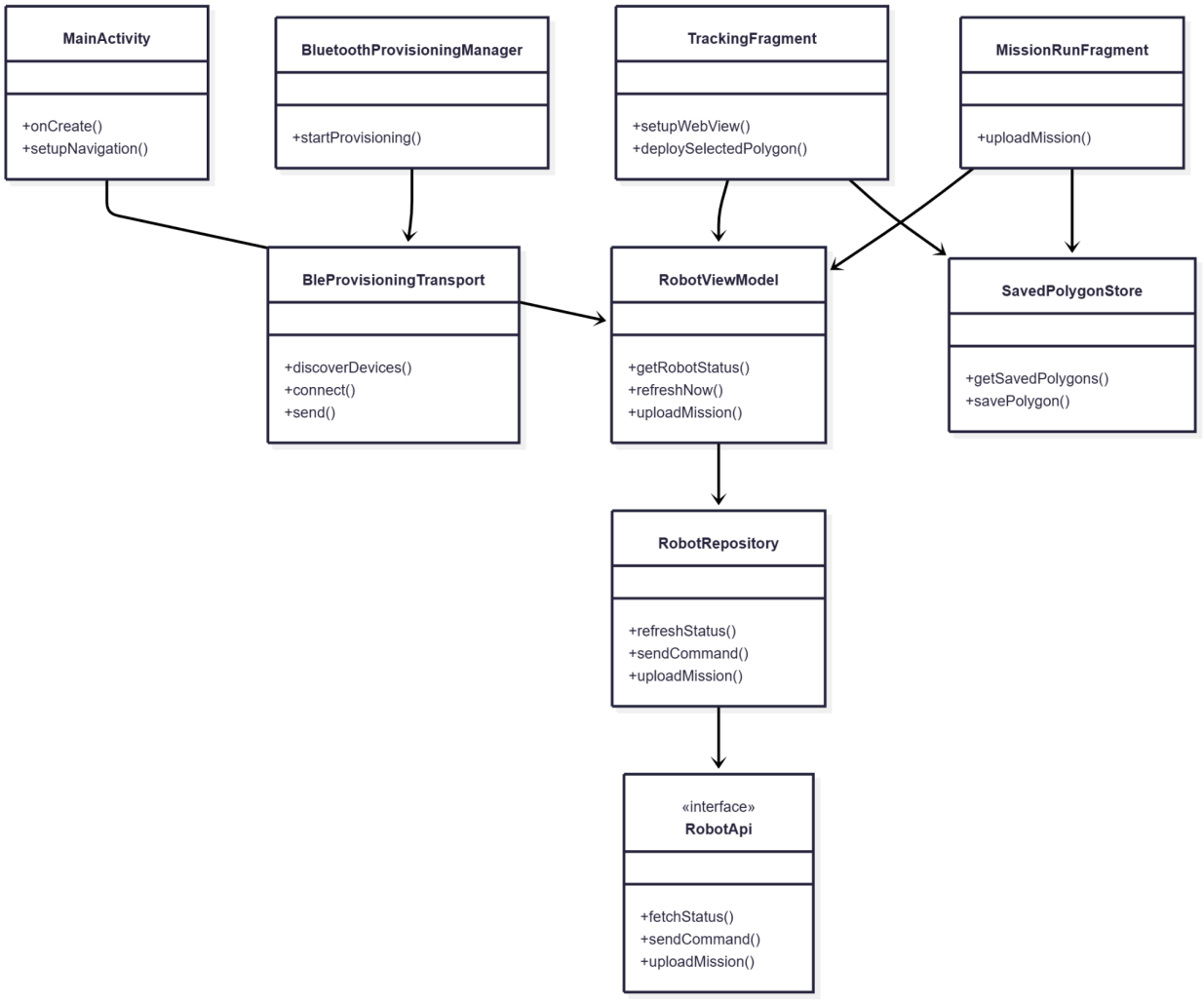
- `RobotStatus`
- `RobotConnectionState`

- `WeatherSnapshot`
- `baseUrl`
- `lastServerHeartbeatMs`
- `lastDriveAckMs`
- `serverLogs`
- `latestTelemetry`
- command and mission result events

Local persistence

Store	**Purpose**
`RobotPreferences`	Base URL, fallback coordinates, demo mode flag
`SavedPolygonStore`	Locally saved polygon GeoJSON and planning metadata

8. Core Class Diagram



9. Screen By Screen Guide

Dashboard

Purpose:

- Show top-level robot summary.
- Surface battery, work status, hopper level, weather, last update time, and recent events.
- Surface alerts for low battery, low salt, and active faults.
- Provide quick access into setup and mission run.

Primary interactions:

- Start and stop commands
- Navigate to setup
- Navigate to mission run

Tracking / Map

Purpose:

- Display robot location on the map using server-provided coordinates.
- Display phone location when permissions allow.
- Let the operator draft, save, edit, preview, and deploy polygon areas.
- Persist saved polygons locally.

Implementation notes:

- Map is a WebView running `app/src/main/assets/map.html`.
- JavaScript bridge methods are exposed through `TrackingMapBridge`.
- Pull-to-refresh is disabled during map touch so panning does not trigger a reload.
- Tile providers now fail over instead of depending on a single source.

Current planning behavior:

- Draft polygons are created by tapping the map.
- Saved polygons are stored as GeoJSON.
- A local base-point placement flow exists on the map page for planning, but it is local planning metadata at present.

Setup

Purpose:

- Read phone Wi-Fi context.
- Scan and choose BLE robot devices.
- Provision the robot over BLE.
- Save the resulting base URL into a shared app state.

Important operator expectations:

- Bluetooth permissions are required.
- The robot must advertise with a `SALTBOT` prefix.
- On success, the app stores `http://<ip>/` and refreshes robot status automatically.

Mission Run

Purpose:

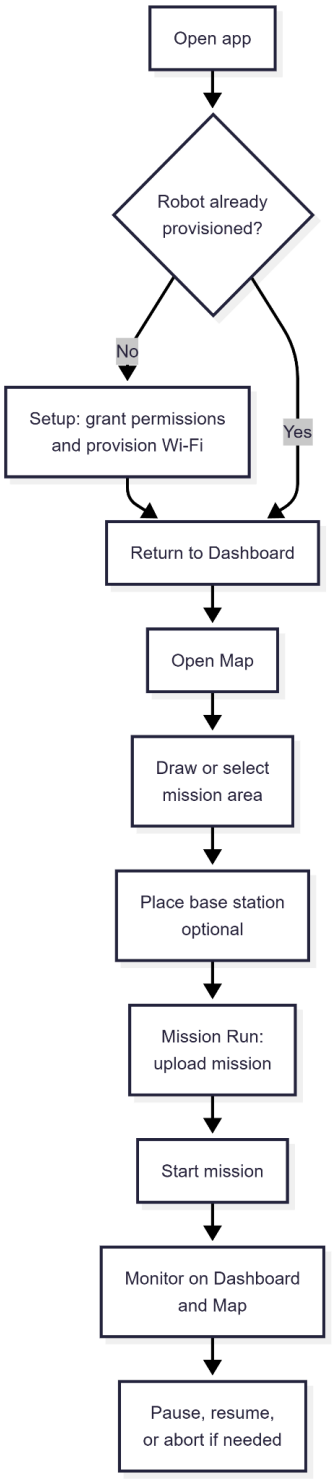
- Select a saved polygon.
- Configure grid resolution, spreader width, and salt mode.
- Upload a mission with `map_lines` and navigation metadata.
- Send mission lifecycle commands such as start, pause, resume, abort, return home, and emergency stop.

Service and testing tools (manual drive, comms console, diagnostics, logs, demo mode) are documented in the Service And Testing Tools section.

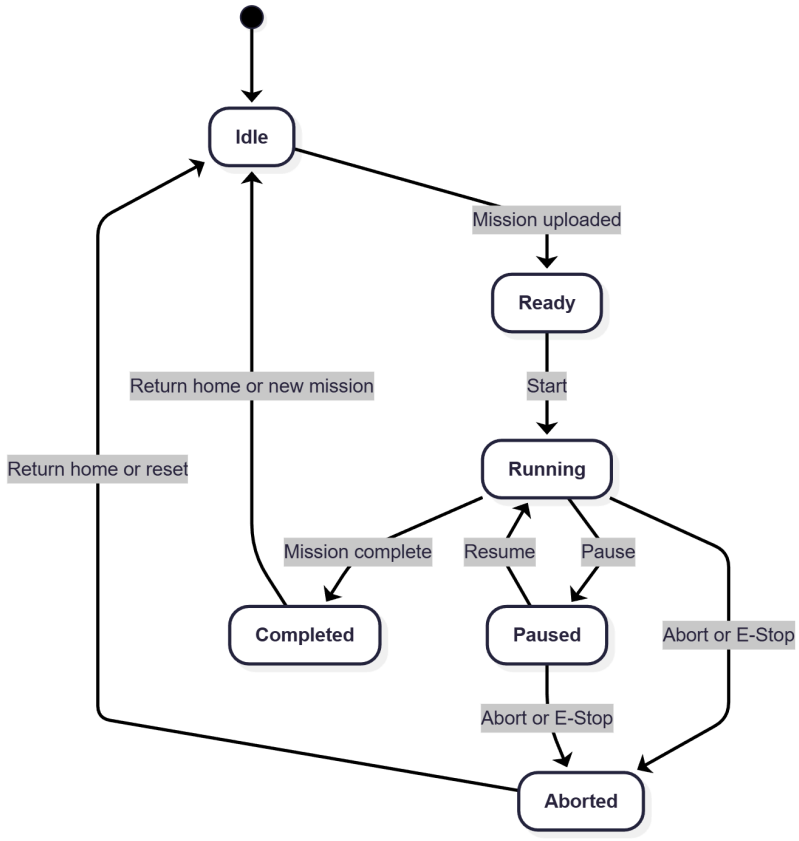
10. User Manual (Operator Guide)

This section is written for operators using the app in the field.

10.1 Quick Start Flow



10.2 Mission Lifecycle



10.3 Setup And Connect

1. Open the app and select Setup if the robot is not provisioned.
2. Grant Bluetooth and location permissions when prompted.
3. Select the robot from the SALTBOT device list and send Wi-Fi credentials.
4. Confirm the Dashboard shows a fresh heartbeat and robot status.

10.4 Create A Mission Area

1. Open Tracking / Map.
2. Tap Add to draw a polygon on the map.
3. Save the polygon with a descriptive name.
4. (Optional) Tap Base to place a base station inside the polygon.

10.5 Upload And Run A Mission

1. Open Mission Run.
2. Choose the saved area, grid resolution, spreader width, and salt mode.
3. Upload the mission and tap Start.

4. Use Pause, Resume, Abort, Return Home, or E-Stop as needed.

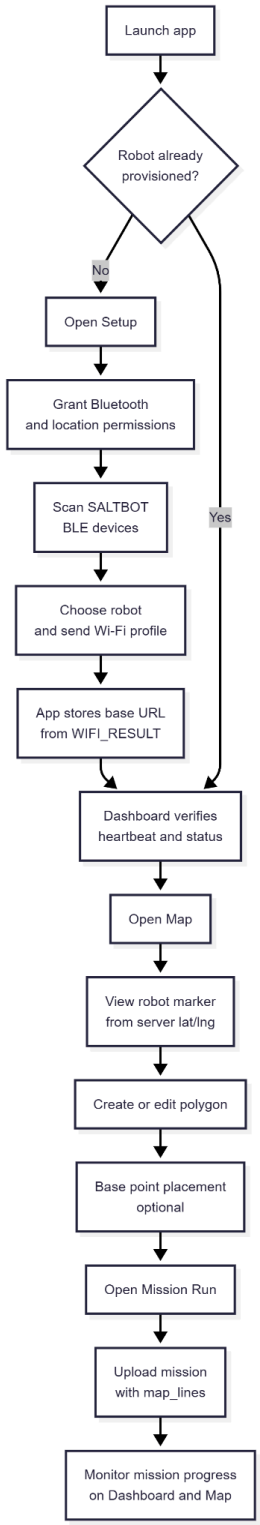
10.6 Monitor Progress And Location

- Track the robot marker on the map for live position.
- Use the Dashboard for current status, alerts, and last update time.
- If the robot stops responding, use Abort or E-Stop and check connectivity.

10.7 Demo Mode (Training)

- Enable demo mode from the Comms screen when hardware is not available.
- The app simulates telemetry and robot position for walkthroughs and screenshots.

11. Operator User Flow



12. Map Implementation Notes

The map is one of the more complex screens in the app because it combines Android touch handling, a WebView, Leaflet, and a JavaScript bridge.

Current map design

- Rendering engine: Leaflet inside `map.html`
- Host control: `TrackingFragment`
- Tile behavior: provider fallback rather than single-provider dependency
- Android to JS flow:
    - set robot location
    - set phone location
    - preview polygon
    - begin polygon draft
    - begin base placement
- JS to Android flow:
    - polygon drawn callback
    - draft point count callback
    - draft export failure callback
    - base point placed callback

Autonomy preparation:

- `SurfaceMapBuilder` builds `map_lines` by sampling satellite tiles inside the polygon. Pixels classified as green are treated as blocked.
- If tiles are unavailable, the builder falls back to a polygon mask so missions can still be uploaded.

Why the map previously felt weak

- Pull-to-refresh could interfere with panning.
- A single failing tile source made rendering look unreliable.
- WebView map UX is more sensitive to lifecycle and touch handling than a native map view.

Current mitigation

- Swipe refresh is disabled while the map is actively being touched.
- Tile layer fallback is enabled.
- Map state is resynced after page load and after layout changes.

13. Logging Model

The app now uses a unified communications log entry model.

Each `ServerLogEntry` contains:

- timestamp
- direction such as `OUT`, `IN`, `SYS`, or `LOG`
- channel such as `HTTP`, `BLE`, `WIFI`, or `TCP`
- endpoint or logical path
- summary message
- optional payload body

This powers the comms console described in the Service And Testing Tools section.

14. Troubleshooting Guide

Robot not reachable after provisioning

Check:

1. BLE setup shows a successful `WIFI_RESULT`.
2. The service comms console base URL is the robot IP with `http://` and trailing slash.
3. `GET /status` appears in logs and returns a valid body.

Drive command seems accepted but robot does not move

Check:

1. The service comms console shows a `DRIVE_CONTROL` payload with expected values.
2. The app heartbeat remains fresh after the command.
3. If the payload is correct and accepted, the next investigation is on the robot side: controller wiring, serial drive channel, power, motor controller state, and hardware fault conditions.

Map is hard to use

Check:

1. Network allows the chosen tile provider.
2. The service comms console is not relevant here unless robot coordinates are missing.
3. Device WebView and graphics performance are adequate.

No BLE devices found

Check:

1. Bluetooth permissions are granted.
2. The robot advertises with a `SALTBOT` name prefix.
3. The robot firmware exposes the expected service and characteristic UUIDs.

TCP telemetry missing

Check:

1. Base URL host resolves to the robot IP.
2. Port `8080` is open on the robot.
3. The robot emits newline-delimited JSON telemetry frames.

15. Service And Testing Tools (Not In Operator Flow)

These tools are intended for commissioning, troubleshooting, and validation. They are not part of the primary operator flow.

Manual Drive Console (Testing Only)

- Screen: `ControlFragment`
- Purpose: validate motors, spreader actuators, and drive responsiveness.
- Caution: use only in controlled environments; missions should be run from the map and mission flow.

Comms Console

- Screen: `LiveCameraFragment`
- Purpose: consolidated HTTP, BLE, Wi-Fi, and TCP telemetry logs.
- Demo mode toggle: use when hardware is unavailable for walkthroughs and screenshots.

Diagnostics, Calibration, Logs

- Screens: `DiagnosticsFragment`, `CalibrationFragment`, `LogsFragment`
- Purpose: service checks, calibration routines, and support logs during bring-up or field service.

Key service/testing files:

- app/src/main/java/com/example/saltspreader/ui/control/ControlFragment.java
- app/src/main/java/com/example/saltspreader/ui/camera/LiveCameraFragment.java
- app/src/main/java/com/example/saltspreader/ui/diagnostics/DiagnosticsFragment.java
- app/src/main/java/com/example/saltspreader/ui/diagnostics/CalibrationFragment.java
- app/src/main/java/com/example/saltspreader/ui/diagnostics/LogsFragment.java
- app/src/main/assets/client_monitor.html