Potato Stock Tracking Mobile App
Use Case, Scope, and Delivery Backlog

Fish and Chips Franchise (South Africa) | Zone Rollout: 5 Shops + Warehouse

Document date: 7 January 2026

1. Executive summary

This document defines the use case and end-to-end scope for a mobile-first stock tracking system focused on potatoes, with support for other critical items later. The goal is to prevent stock-outs across five shops in a zone, while maintaining enough warehouse buffer to respond quickly when suppliers run out. The system records every stock movement as an auditable event (receive, use, waste, transfer, adjustment), provides real-time on-hand visibility per location, and produces replenishment forecasts and alerts using consumption rates, lead times, and safety stock.

The recommended delivery approach is an offline-capable mobile app with role-based access control, backed by a cloud database and simple forecasting logic that improves over time. A minimal viable product (MVP) can be delivered with accurate on-hand tracking, basic forecasting and reorder alerts, warehouse and inter-store transfers, and reporting for zone managers. The backlog in this document breaks the work into epics and detailed, vibe-coding-ready tickets with acceptance criteria and prompts.

2. Context and problem statement

Potatoes are a core input for fish and chips operations and are subject to variability in supplier availability, delivery timing, and daily consumption. Stock-outs have a direct impact on sales, customer satisfaction, and staff efficiency. Current tracking methods are often manual, fragmented across shops, and do not reliably support forecasting or coordinated replenishment.

The problem to be solved is an operational visibility and planning gap: the business needs to know how much potato stock arrives, how much is used, how much is wasted, how much is held in the warehouse, and how much is available at each shop, so that replenishment can be triggered before stock runs out. The solution must also support situations where the supplier runs out, which increases the importance of safety stock policies and accurate consumption forecasting.

This project will deliver a mobile-first inventory platform that enables consistent capture of stock movements, real-time stock positions, and reliable reorder guidance across five shops and one warehouse, with clear accountability through audit trails and role-based permissions.

3. Goals and success criteria

- Prevent potato stock-outs in shops through timely reordering and warehouse buffering.
- Provide a single source of truth for on-hand stock per shop and warehouse.
- Create an auditable history of stock movements, including who did what and when.
- Enable zone-level visibility, reporting, and exception handling (low stock, unusual usage, supplier shortage).
- Support practical constraints: low-cost Android devices, intermittent connectivity, and fast data entry for frontline staff.

Suggested measurable success criteria (tailor during discovery):

- Stock-out incidents reduced (for example, by at least 70 percent) after adoption and stabilization.
- On-hand accuracy within an agreed tolerance during weekly stocktakes (for example, within plus or minus 3 to 5 percent).
- Reorder alerts generated at least one lead-time ahead for 90 percent or more of replenishment cycles.
- Average time to log a delivery or daily usage entry stays under 60 seconds per action.
- Warehouse can fulfil emergency top-ups within a defined response time because buffer levels are visible and enforced.

4. Scope, assumptions, and boundaries

In-scope for MVP:

- Potatoes inventory only (with extensibility for additional items later).
- Locations: 5 shops (stores) and 1 warehouse, with transfers between them.
- Stock movement events: receiving, usage, waste, transfer out, transfer in, adjustment, and stocktake counts.
- Forecasting and reorder guidance based on recent consumption, lead time, and safety stock.
- Alerts: low stock, reorder due, unusual consumption, supplier shortage recorded, and overdue stocktake.
- Role-based access: staff, store manager, warehouse manager, zone manager, and system admin.
- Offline-capable mobile experience and a web dashboard for managers as an optional phase.

Out-of-scope for MVP (candidate later phases):

- Full recipe and production planning for all menu items (beyond potatoes).
- Direct integrations with third-party POS, accounting, or supplier systems (unless already accessible and low effort).
- Advanced machine learning forecasting. The MVP will use transparent, configurable forecasting heuristics.

Key assumptions: potatoes are tracked in a consistent unit (recommended: kilograms and/or standard bag sizes), each delivery can be recorded with supplier and batch metadata, and shops are willing to log daily usage (or reconcile usage from sales and waste).

5. Users, roles, and permissions

Primary user groups:

Role	Typical permissions
Shop staff	Log usage and waste, view current on-hand for their shop, submit low-stock notes.
Store manager	Approve adjustments, review forecasts and reorder alerts, initiate warehouse requests, run stocktakes.
Warehouse manager	Record inbound deliveries to warehouse, manage warehouse buffer targets, fulfil shop requests and transfers.
Zone manager	View zone-wide stock positions, compare shops, monitor risk of stock-outs, and review audit reports.
System admin	Configure locations, users, item definitions, units, lead times, safety stock settings, and alert thresholds.

6. Core use cases and user journeys

The system is designed around stock movement events. Every change to inventory is recorded as an event, which produces a computed on-hand balance per item per location. This event-first approach improves auditability, handles offline entry, and reduces errors caused by direct editing of on-hand values.

6.1 Receive stock (warehouse or shop)

A user selects the location, enters supplier details, delivery reference, unit quantity (for example, number of 10 kg bags or total kilograms), optionally captures a photo of the delivery note, and confirms. The app updates on-hand and recalculates risk and forecast.

6.2 Record daily usage and waste (shop)

At least once per day (or per shift), shop staff record how many kilograms were used for production and how much was wasted or rejected. If the shop uses a standard yield rate, the app can suggest usage based on sales, but the logged usage remains the source of truth until POS integration exists.

6.3 Transfer between warehouse and shop (and optionally between shops)

A manager creates a transfer out from a source location, which generates a pending transfer in for the destination location. The destination confirms receipt, supporting partial receipt and discrepancy notes. Transfers maintain a clear chain of custody.

6.4 Stocktake and adjustment

A manager performs a stock count and records the counted quantity. The system calculates variance and requires a reason code for any adjustment. This keeps audit trails intact and improves forecasting by distinguishing shrinkage from genuine consumption.

6.5 Forecast and reorder guidance

The app shows days of cover, projected stock-out date, and recommended reorder quantity. Guidance is configurable per shop and item based on lead time and safety stock. Alerts are triggered when projected cover falls below thresholds.

6.6 Supplier shortage event

When a supplier is out of stock, staff can record a shortage event with supplier name, date, and notes. The system uses this information to encourage higher safety stock and to support reporting on supply reliability.

7. Functionality to be built (MVP and beyond)

7.1 MVP functionality

- Authentication and role-based permissions for staff and managers.
- Location management for 5 shops and 1 warehouse.
- Item catalog for potatoes, including units, conversion (bags to kg), and typical shrink or waste categories.
- Stock movement capture: receive, use, waste, transfer, adjustment, and stocktake.
- Real-time on-hand per location and zone roll-up views.
- Forecasting: consumption rate calculation, days of cover, projected stock-out date, reorder point, and suggested reorder quantity.
- Alerts and notifications: low stock, reorder due, unusual usage, missing usage entry, and overdue stocktake.
- Basic reporting: movement history, weekly usage, waste rate, stock-out risk, and variance from stocktakes.
- Offline-first mobile flows with queued sync and conflict handling.
- Audit trails with user, timestamp, reason codes, and attachment support (photos of delivery notes).

7.2 Phase 2 and future enhancements (examples)

- Web dashboard for zone managers and admins (analytics, configuration, exports).

- Supplier and purchase order workflow (requests, approvals, expected deliveries).
- POS integration to estimate potato usage from sales and standard recipes.
- Multi-item inventory (oil, packaging, fish, spices), with category-based reporting.
- Advanced forecasting: seasonality, promotions, weather signals, and anomaly detection.
- Barcode or QR code scanning for bag tracking and batch traceability.
- Automated reorder suggestions to warehouse and supplier, including consolidated zone orders.

8. Capabilities and how they address the operational pain points

Pain point	Capability	How it helps
Running out of potatoes in shops	Days-of-cover and stock-out projection with alerts	Triggers action before stock reaches zero and enables proactive transfers.
Unclear warehouse buffer	Warehouse target levels and zone risk view	Sets a minimum buffer and shows when replenishment is needed to protect shops.
Supplier sometimes out of stock	Supplier shortage events and safety stock policies	Supports higher safety stock during high-risk periods and provides reporting for supplier decisions.
Manual, inconsistent tracking	Standardized movement event capture with audit trail	Creates a single truth source and reduces errors from spreadsheets or informal notes.
Consumption variability and wastage	Usage and waste logging with variance reporting	Separates genuine demand from waste and helps improve operations and cost control.
Emergency transfers create confusion	Transfer workflow with confirmation and discrepancies	Prevents double-counting and clarifies responsibility across locations.

9. Data model (high-level)

The recommended model records immutable stock movement events and computes on-hand balances per item and location. To support fast screens and reporting, balances can be denormalized into a current-balance table that is updated after each event, while the event log remains the authoritative history.

Entity	Key fields (example)
User	id, name, role, assigned locations, status
Location	id, type (shop or warehouse), name, address, timezone
Item	id, name, base unit (kg), optional pack unit (bag), conversion rate, active flag

Supplier	id, name, contact, typical lead time, reliability notes
StockEvent	id, timestamp, user_id, location_id, item_id, type, quantity (signed), unit, reason_code, notes, attachments
CurrentBalance	location_id, item_id, on_hand_base_unit, last_updated
ForecastSnapshot	location_id, item_id, consumption_rate, days_of_cover, stockout_date, reorder_point, suggested_order_qty, generated_at
Alert	id, location_id, item_id, type, severity, status, created_at, acknowledged_by

10. Forecasting and replenishment logic (MVP approach)

The MVP uses simple, explainable heuristics that managers can trust and adjust. Consumption rate is computed from recent usage events, for example a 7-day or 14-day moving average, with optional exclusion of outlier days. If usage is not logged daily, the system can fall back to the last known rate or a configured baseline until data quality improves.

Days of cover is calculated as on-hand divided by daily consumption rate. The projected stock-out date is the current date plus days of cover. The reorder point is calculated as (lead time in days multiplied by daily consumption) plus safety stock. Safety stock can be configured per shop and increased during supplier shortage periods. Suggested reorder quantity can be calculated to reach a target level such as (target days of cover multiplied by daily consumption) minus on-hand, bounded by minimum order quantities and warehouse capacity where applicable.

This logic should be implemented so that every recommendation can be explained on screen in plain terms: current on-hand, recent daily usage, lead time, safety stock, reorder point, and the resulting suggested order quantity.

11. Non-functional requirements

- Offline-first: core flows must work without connectivity and sync when back online.
- Performance: home dashboard should load within 2 seconds on mid-range Android devices under normal connectivity.
- Security: least-privilege access, encrypted transport, secure token storage, and audit trails.
- Reliability: clear conflict handling and idempotent event writes to prevent double-posting.
- Data quality: required fields, reason codes, validation for unit conversions, and stock cannot go negative unless explicitly allowed with manager approval.
- Compliance: POPIA-aware handling of personal information and secure storage of delivery note images.

- Observability: error monitoring, event logging, and basic analytics to understand adoption and data quality.

12. Tech stack options (mobile-first)

The stack should optimize for fast delivery, offline capability, and long-term maintainability. Below are three viable stacks. The recommendation is to start with the simplest stack that reliably supports offline, event logging, and role-based access.

Option A: React Native (Expo) + Firebase

- Mobile: React Native with Expo, TypeScript, React Navigation, form validation with Zod or Yup.
- Auth: Firebase Authentication (email, phone, or SSO later).
- Database: Cloud Firestore for event log and current balances, with offline persistence.
- Backend logic: Cloud Functions for balance updates, forecast snapshots, and alert generation.
- Files: Firebase Storage for delivery note photos.
- Push notifications: Firebase Cloud Messaging.
- Pros: fastest path to offline-capable mobile app; strong ecosystem; simple ops.
- Trade-offs: vendor lock-in; careful data modeling required for reporting and complex queries.

Option B: Flutter + Supabase (Postgres)

- Mobile: Flutter, Dart, Riverpod or Bloc for state management.
- Auth and DB: Supabase Auth + Postgres with row-level security, plus edge functions for server logic.
- Storage: Supabase Storage for images.
- Pros: excellent UI performance; relational data and reporting are strong; SQL analytics is straightforward.
- Trade-offs: offline-first requires additional client-side persistence and custom sync; more engineering effort for reliable offline queueing.

Option C: PWA + Capacitor (mobile app wrapper) + Node backend

- Client: Next.js PWA with offline caching and local database (IndexedDB) packaged via Capacitor for app stores.
- Backend: NestJS or Fastify, Postgres, Redis for queues, hosted on a cloud provider.
- Pros: one codebase for web and mobile; good for manager dashboards; flexible.
- Trade-offs: offline stock event syncing is harder than Firestore; more DevOps overhead.

Pragmatic recommendation for a first release: Option A is often the most efficient for an offline-first inventory MVP, particularly if the team is small and speed matters. Option B can be preferred if you want strong SQL reporting from day one and you can invest in offline synchronization engineering.

13. Delivery plan from start to finish

Discovery and alignment

- Confirm units of measure and operational process (receiving, daily usage, stocktake cadence, transfers).
- Agree on roles, permissions, and who is accountable for each action in each location.
- Define lead times, safety stock policies, and reporting needs per shop.
- Identify data quality risks and design controls (required fields, reason codes, approvals).

Design and prototyping

- Map key screens and flows and validate with staff for speed and clarity.
- Decide on offline behavior and sync conflict rules.
- Define alert types and thresholds and how they will be acknowledged.
- Design the data model and event types with examples.

Build and test (MVP)

- Implement authentication, roles, and location scoping.
- Implement stock movement event capture with validation and audit trail.
- Implement computed balances, dashboards, and movement history.
- Implement forecasting, reorder guidance, and alerts.
- Implement offline queueing and sync with idempotency.
- Quality assurance: functional tests, offline tests, and data integrity tests.

Pilot and rollout

- Pilot in 1 shop and the warehouse, collect feedback, and fix data entry friction.
- Train managers and staff with simple SOPs and in-app guidance.
- Roll out to remaining shops, monitor data quality, and tune thresholds.
- Establish weekly operating rhythm: stocktake, forecast review, and exception handling.

Operate and improve

- Monitor stock-out incidents and alert accuracy; adjust lead times and safety stock.
- Add reporting and export features for deeper analysis.
- Expand to additional SKUs and optional supplier or POS workflows.

14. Development backlog (epics and detailed tickets)

The tickets below are written for fast, iterative development. Each ticket includes a clear objective, acceptance criteria, and a vibe-coding prompt you can paste into your coding assistant. Adjust naming conventions and stack details after selecting a tech option.

Epic A. Foundations (repo, CI, environments, conventions)

A1 - Initialize mobile app repo and baseline architecture

Objective: Create a clean, production-ready mobile app project with navigation, state management, and environment configuration.

Acceptance criteria:

- App builds and runs on Android emulator and a physical device.
- Navigation includes a placeholder authenticated and unauthenticated stack.
- Environment variables supported for dev and prod (API keys, project IDs).
- Linting and formatting are configured and enforced.

Vibe-coding prompt:

```
You are my senior mobile engineer. Create a React Native Expo (TypeScript) project.
Add: React Navigation, a basic auth gate with placeholder screens, a typed navigation
structure,
and ESLint + Prettier. Use an app folder structure: src/screens, src/components,
src/services,
src/state, src/utils. Add dotenv support for env vars. Provide code and file tree.
```

A2 - Set up crash/error monitoring and logging

Objective: Add basic observability so issues can be diagnosed during pilot rollout.

Acceptance criteria:

- Errors are captured in an error monitoring tool (for example Sentry).
- App logs include user id and location id after login.
- Sensitive information is not logged.

Vibe-coding prompt:

```
Add Sentry (or equivalent) to the Expo React Native app with environment-based
configuration.
Instrument global error boundaries and navigation instrumentation. Add a small logging
utility
that attaches context (userId, locationId) after login and strips PII. Provide code
and setup steps.
```

A3 - Define data conventions and stock event types

Objective: Lock down the event-sourcing approach for inventory changes and enforce consistent units and reason codes.

Acceptance criteria:

- Documented list of StockEvent types: RECEIVE, USE, WASTE, TRANSFER_OUT, TRANSFER_IN, ADJUSTMENT, STOCKTAKE.
- Quantity sign rules defined (for example USE is negative).
- Reason codes defined for adjustments and waste (for example spoilage, trimming loss, theft, data correction).
- Unit conversion rules documented (bag to kg) and validated in code.

Vibe-coding prompt:

```
Create a TypeScript module that defines:
1) StockEventType enum, 2) ReasonCode enums, 3) unit conversion helpers, 4) validation
helpers using Zod.
Include doc comments explaining sign rules and examples. Provide tests for conversion
and validation.
```

Epic B. Authentication, authorization, and tenant scoping

B1 - Implement authentication (email or phone) and session persistence

Objective: Allow users to sign in and remain signed in across app restarts.

Acceptance criteria:

- Users can sign in and sign out.
- Session persists across restarts.
- Unauthenticated users cannot access inventory screens.

Vibe-coding prompt:

```
Implement Firebase Auth (email + password) in Expo React Native.
Create screens: Login, ForgotPassword, and a basic Profile with Sign Out.
Add an auth context hook useAuth() and protect routes. Provide code and minimal UI.
```

B2 - Role-based access control (RBAC) and location scoping

Objective: Restrict features by role and ensure each user can only operate on assigned locations.

Acceptance criteria:

- Roles supported: STAFF, STORE_MANAGER, WAREHOUSE_MANAGER, ZONE_MANAGER, ADMIN.

- Each user profile includes assigned location ids.
- UI hides restricted actions and backend rules enforce restrictions.
- Attempted unauthorized actions are blocked and logged.

Vibe-coding prompt:

```
Design and implement RBAC for the inventory app on Firebase.
Create a user profile document structure (role, assignedLocationIds).
Add client-side guards (canPerform(action, context)) and Firestore security rules
that enforce location scoping. Provide example rules and code.
```

Epic C. Core inventory capture and balances

C1 - Create item catalog for potatoes with units and pack conversion

Objective: Represent potatoes as an item with base unit (kg) and optional pack unit (bag) conversion.

Acceptance criteria:

- Potatoes item exists with base unit kg.
- Optional pack unit configured (for example 10 kg bag) with conversion rate.
- UI supports entering quantities in kg or bags and stores normalized kg.
- Validation prevents invalid or negative quantities.

Vibe-coding prompt:

```
Implement an Item model and an ItemService that loads items from Firestore.
Add UI components for quantity entry that allows selecting unit (kg or bags) and
converts to kg.
Use Zod validation. Include unit tests for conversion and edge cases.
```

C2 - Implement StockEvent write path (create event) with idempotency

Objective: Allow creating stock events safely, including offline queueing and no double-posting.

Acceptance criteria:

- StockEvent create API accepts a client-generated idempotency key.
- If the same key is submitted twice, it does not duplicate the event.
- Events include user id, location id, item id, timestamp, type, quantity_kg, reason code, and notes.
- Works offline and syncs later without creating duplicates.

Vibe-coding prompt:

```
Implement a StockEventService for Firestore with:
- client-generated eventId (UUID) used as document id,
- idempotent writes (set with merge false),
```

```
- offline support using Firestore persistence.
Create a function createStockEvent(input) that validates input, writes event, and
returns eventId.
Include tests/mocks and explain offline behavior.
```

C3 - Maintain CurrentBalance per location and item

Objective: Compute and store on-hand balance after each event for fast dashboards.

Acceptance criteria:

- CurrentBalance is updated whenever a StockEvent is created.
- Balances are correct for all event types including transfers and stocktakes.
- Balance updates are atomic and concurrency-safe.

Vibe-coding prompt:

```
Implement a Firebase Cloud Function triggered on StockEvent creation that updates
CurrentBalance.
Use Firestore transactions to apply the delta quantity_kg.
Handle STOCKTAKE as a set-to-count operation (store variance separately).
Write tests for the Cloud Function logic and provide deployment steps.
```

C4 - Movement history screen (per location)

Objective: Allow users to review recent stock events and filter by type and date range.

Acceptance criteria:

- List shows timestamp, type, quantity, user, and notes.
- Filters: event type, last 7 days, last 30 days, custom range.
- Tapping an event shows a detail screen with attachments if present.

Vibe-coding prompt:

```
Build a MovementHistory screen in React Native.
Query Firestore for StockEvents scoped to current location and potatoes item.
Implement filter chips and date filtering. Add an EventDetail screen with formatted
fields
and an attachments section (placeholder for now). Provide code.
```

Epic D. Operational flows

D1 - Receive stock flow (delivery capture + photo attachment)

Objective: Allow warehouse or shop to record deliveries quickly and attach delivery notes.

Acceptance criteria:

- User selects location (within their assigned set).
- User enters supplier, reference, quantity (kg or bags), and optional notes.
- User can take a photo and attach it.
- A RECEIVE StockEvent is created and CurrentBalance updates.

Vibe-coding prompt:

```
Create a ReceiveStock screen with a fast form:
supplier (text), reference (text), quantity (number), unit selector (kg/bags), notes,
attach photo.
Store photo in Firebase Storage and save URL in the StockEvent.
Include loading states, validation, and success toast.
```

D2 - Daily usage entry flow

Objective: Make it easy for shops to log potato usage at least daily and flag missing entries.

Acceptance criteria:

- Usage can be entered as total kg used for the day or shift.
- USE event is created with negative quantity.
- System can show a reminder if no usage was logged for the previous day.

Vibe-coding prompt:

```
Implement a DailyUsage screen:
- shows today's date, last 7 days summary, and an entry form.
- create USE StockEvent with negative quantity_kg.
Add a background check (on app open) that detects missing usage events for yesterday
and shows a reminder banner. Provide code.
```

D3 - Waste logging flow

Objective: Track wastage separately to improve forecasting and operational control.

Acceptance criteria:

- Waste entry requires a reason code and quantity.
- WASTE event is created with negative quantity.
- Waste rate is visible in weekly report.

Vibe-coding prompt:

```
Implement a WasteEntry screen similar to DailyUsage.
Require a reason code dropdown (spoilage, trimming loss, dropped, other).
```

```
Create WASTE StockEvent with negative quantity_kg and reason_code.
Add a small chart placeholder component for weekly waste (data only for now).
```

D4 - Transfer workflow (warehouse to shop) with confirmation

Objective: Support transferring potatoes to shops with a two-step confirm process.

Acceptance criteria:

- Source creates TRANSFER_OUT with destination location id and quantity.
- Destination sees pending transfer and confirms receipt, creating TRANSFER_IN.
- Partial receipt supported with discrepancy notes.
- Balances update correctly at both locations.

Vibe-coding prompt:

```
Design a transfer flow:
1) CreateTransfer screen for source location creates a Transfer record + TRANSFER_OUT
event.
2) PendingTransfers screen for destination lists open transfers.
3) ConfirmTransfer screen allows confirm full or partial receipt and creates
TRANSFER_IN event.
Model Transfer with status OPEN/RECEIVED/PARTIAL and link event ids.
Provide Firestore schema and React Native screens.
```

D5 - Stocktake and adjustment with variance reason

Objective: Enable periodic counts and controlled adjustments.

Acceptance criteria:

- Stocktake records counted kg and calculates variance from system on-hand.
- If variance is non-zero, manager must select a reason code and confirm.
- An ADJUSTMENT event is recorded with variance amount.

Vibe-coding prompt:

```
Implement Stocktake screen:
- fetch current balance,
- user enters counted quantity,
- show variance,
- require manager role to submit if variance != 0,
- create STOCKTAKE record and ADJUSTMENT StockEvent for the variance with reason code.
Include UI and validation. Provide code.
```

Epic E. Forecasting, reorder guidance, and alerts

E1 - Compute consumption rate and forecast snapshot per location

Objective: Generate daily consumption rates and forecasts that can be displayed and explained.

Acceptance criteria:

- Consumption rate calculated from last N days of USE and WASTE events (configurable).
- ForecastSnapshot stored per location daily or on-demand.
- Forecast includes days of cover and projected stock-out date.

Vibe-coding prompt:

```
Implement a backend job (Cloud Function scheduled daily) that:
- reads StockEvents for each location for the last 14 days,
- computes daily consumption rate using moving average,
- writes ForecastSnapshot (rate, daysOfCover, stockoutDate, generatedAt).
Include configuration in Firestore for N days and outlier trimming.
Provide code and tests.
```

E2 - Reorder point and suggested order quantity

Objective: Calculate reorder guidance using lead time and safety stock policies per location.

Acceptance criteria:

- Reorder point = lead_time_days * daily_consumption + safety_stock_kg.
- Suggested order quantity targets a configurable days-of-cover (for example 7 days).
- Rules are explainable on the UI.

Vibe-coding prompt:

```
Extend the forecasting function to compute reorderPointKg and suggestedOrderKg.
Load per-location settings: leadTimeDays, safetyStockKg, targetCoverDays.
Write the calculations, store in ForecastSnapshot, and add an explain() payload that
breaks down numbers.
Provide code and example output.
```

E3 - Alert generation and notification delivery

Objective: Create actionable alerts and optionally push notifications.

Acceptance criteria:

- Alert types: LOW_STOCK, REORDER_DUE, UNUSUAL_USAGE, MISSING_USAGE_ENTRY, OVERDUE_STOCKTAKE.
- Alerts created based on ForecastSnapshot and data completeness checks.

- Users can acknowledge alerts; acknowledged status is stored.
- Optional: push notifications for managers.

Vibe-coding prompt:

```
Implement an Alerts service:
- backend evaluates ForecastSnapshot and creates Alert documents with severity,
- client displays Alerts inbox and supports acknowledge action,
- optional push notification via FCM to managers for HIGH severity.
Provide Firestore schema, Cloud Function logic, and client UI screens.
```

E4 - Supplier shortage event capture and safety stock bump

Objective: Track supplier stock-outs and increase safety buffer when shortages occur.

Acceptance criteria:

- User can log SupplierShortage with supplier name, date, and notes.
- System increases safety stock for a configurable period after a shortage event (for example 7 days).
- Shortage events are reportable.

Vibe-coding prompt:

```
Implement SupplierShortage feature:
- Create a shortage log screen that writes SupplierShortage records.
- Modify forecast calculation to apply a 'shortageMultiplier' to safetyStockKg when a
recent shortage exists.
Add config fields: shortageWindowDays, safetyStockMultiplierDuringShortage.
Provide code, tests, and UI.
```

Epic F. Dashboards and reporting

F1 - Shop dashboard (today view)

Objective: Provide at-a-glance status for staff and managers in a shop.

Acceptance criteria:

- Shows on-hand kg, days of cover, next reorder recommendation, and active alerts.
- Shows last delivery date and last usage entry date.
- Loads quickly and works offline with last known data.

Vibe-coding prompt:

```
Build a ShopDashboard screen that reads CurrentBalance, ForecastSnapshot, and Alerts
for the selected location.
Design UI cards for On-hand, Forecast, Alerts, and Recent Activity.
Include offline cache behavior and loading skeletons. Provide code.
```

F2 - Zone dashboard (5 shops + warehouse roll-up)

Objective: Give zone managers a view across all locations to spot risk early.

Acceptance criteria:

- Shows each location with on-hand, days of cover, and risk indicator.
- Supports sorting by lowest days of cover.
- Allows drilling into a location detail view.

Vibe-coding prompt:

```
Build a ZoneDashboard screen restricted to ZONE_MANAGER and ADMIN roles.
Query summaries for all locations, show a sortable list/table, and a LocationDetail
view.
Include simple risk banding based on days of cover. Provide code and Firestore query
plan.
```

F3 - Weekly report: usage, waste, and variance

Objective: Provide management reporting for consumption and stock accuracy.

Acceptance criteria:

- Report shows weekly totals for usage and waste per shop.
- Shows stocktake variance when a stocktake occurred in the week.
- Allows export to CSV (Phase 2) or share as PDF (optional).

Vibe-coding prompt:

```
Implement a WeeklyReport screen:
- backend aggregate function computes weekly totals per location,
- client displays totals and variance.
Implement a first version with on-screen tables.
Add a TODO placeholder for export/share. Provide code.
```

Epic G. Offline-first, sync integrity, and conflict handling

G1 - Offline queue and retry strategy for writes

Objective: Ensure users can create events offline and sync safely.

Acceptance criteria:

- Writes made offline are queued automatically and retried when online.
- User can see sync status and retry failures.

- No duplicate StockEvents are created when retrying.

Vibe-coding prompt:

```
Implement offline behavior for StockEvent writes.
If using Firestore: enable persistence and show sync state (pending writes).
If using a custom backend: implement a local outbox table (SQLite) with retry and
idempotency keys.
Provide code for whichever architecture you choose, plus a SyncStatus component.
```

G2 - Conflict rules for transfers and stocktake edits

Objective: Define how conflicts are handled when multiple users act on the same workflow.

Acceptance criteria:

- Transfers cannot be confirmed twice; the system is idempotent.
- Stocktake submission locks the stocktake record; edits require manager override and leave audit trail.
- Any conflict shows a clear user message and does not corrupt balances.

Vibe-coding prompt:

```
Implement integrity rules:
- Transfer documents use status transitions enforced server-side.
- Stocktake records are immutable after submission; create a correction record for
changes.
Add server-side checks (Cloud Functions or security rules) and client messages for
conflict cases.
Provide code and examples.
```

Epic H. Admin configuration (minimum for MVP)

H1 - Admin: manage locations and users

Objective: Allow admins to create locations and assign users to roles and locations.

Acceptance criteria:

- Admin can create and edit locations (shops and warehouse).
- Admin can create users, set roles, and assign location ids.
- Changes take effect immediately on user login.

Vibe-coding prompt:

```
Create an Admin panel (mobile screens or a simple web page) that allows:
- CRUD for locations,
- CRUD for users (role, assigned locations).
```

```
Implement Firestore collections: locations, users.
Add RBAC checks so only ADMIN can access. Provide code.
```

H2 - Admin: configure forecasting settings per location

Objective: Make lead time and safety stock tunable and visible.

Acceptance criteria:

- Settings include leadTimeDays, safetyStockKg, targetCoverDays, movingAverageDays.
- Settings are stored per location and applied to forecasts.
- Changes are audited.

Vibe-coding prompt:

```
Implement ForecastSettings:
- Firestore doc per location with fields leadTimeDays, safetyStockKg, targetCoverDays,
movingAverageDays.
- Admin UI to edit settings with validation.
- Add audit log entry on change (who, when, before/after).
Provide code.
```

Epic I. Quality assurance, security, and release

I1 - Automated tests for calculation and balance integrity

Objective: Prevent regressions in core inventory logic.

Acceptance criteria:

- Unit tests cover unit conversion, event sign rules, and reorder calculations.
- Integration tests cover balance update logic for each StockEvent type.
- At least one test covers offline idempotent writes.

Vibe-coding prompt:

```
Create a test suite for the inventory domain:
- unit tests for conversion and forecasting formulas,
- integration tests for balance updates (simulate sequences of events),
- include edge cases: negative prevention, stocktake set-to-count, partial transfer.
Provide code and how to run tests in CI.
```

I2 - Security review: Firestore rules and least-privilege

Objective: Ensure data is protected and access is correctly scoped.

Acceptance criteria:

- Users can only read/write data for permitted locations.
- Only managers can create adjustments and stocktakes.
- Rules are tested with emulator tests.
- Images are protected with signed URLs or authenticated access.

Vibe-coding prompt:

```
Write Firestore security rules for all collections and add emulator tests.
Include role checks, location scoping, and restrictions on event types.
Also configure Storage rules for attachments. Provide code and test steps.
```

I3 - Release pipeline and deployment checklist

Objective: Enable repeatable builds and safe rollout.

Acceptance criteria:

- Build pipeline produces Android builds for testing and release.
- Environment configuration supports dev and prod projects.
- A release checklist exists (migrations, rules deployment, smoke tests).

Vibe-coding prompt:

```
Set up EAS Build for Expo.
Create scripts for dev and prod builds, configure env vars, and write a release
checklist.
Include smoke tests: login, receive stock, usage entry, transfer, dashboard, alerts.
Provide documentation in a RELEASE.md file.
```

Appendix A. Suggested screen list (MVP)

- Login / Forgot password / Profile
- Select location (if user has multiple)
- Shop dashboard
- Warehouse dashboard (same components with transfers focus)
- Receive stock
- Daily usage
- Waste entry
- Create transfer (source)
- Pending transfers (destination)
- Stocktake
- Movement history + event detail
- Alerts inbox + alert detail

- Zone dashboard (manager)
- Admin: locations, users, forecast settings

Appendix B. Glossary

Term	Definition
On-hand	The currently available quantity at a location, in kilograms.
StockEvent	An immutable record of a change in stock (receive, use, waste, transfer, adjustment, stocktake).
Days of cover	Estimated number of days the current stock will last given recent consumption.
Lead time	Days between ordering and receiving stock.
Safety stock	Extra buffer kept to reduce risk of stock-out when supply is uncertain.
Reorder point	Stock level at which replenishment should be initiated.