**Week 1 — Repo & core skeleton (goal: working KMM + GraphQL skeleton + admin product upsert)**

**Day 1 — Project setup**

* Create monorepo layout:
  + /shared (KMM)
  + /ios-app (Xcode + SwiftUI shell)
  + /android-app (Android Studio + Compose shell)
  + /backend (Node TS + Apollo)
  + /infra (Dockerfiles, terraform stubs)
* Initialize GitHub repo, initial README, branch protection rules.
* Setup GitHub Actions skeleton for lint/build.

**Day 2 — Backend skeleton**

* Node.js TypeScript project scaffold:
  + Express + Apollo Server (or GraphQL Yoga)
  + Basic GraphQL server with health query.
  + Dockerfile for backend.
* Provision Postgres (locally / managed) and Redis instance.
* Create initial DB schema (users, products, orders, coupons).
  + SQL: products(id, sku, name, price, cost\_price, stock, metadata, created\_at).

**Day 3 — KMM skeleton & GraphQL client**

* Initialize KMM shared module (Gradle).
* Add Apollo Kotlin to shared module; generate types from simple GraphQL schema (start with product upsert/mutation & queries).
* Expose ProductRepository interface in commonMain.

**Day 4 — Admin upsert flow (end-to-end)**

* Backend: Implement upsertProduct(input) GraphQL mutation that writes to Postgres and publishes index job.
* KMM: implement upsertProduct() call using Apollo client.
* iOS/Android: wire a simple native form UI that calls into KMM to upsert a product (no camera yet).

**Day 5 — Tests & CI**

* Add unit tests for backend mutation (jest / ts-jest).
* Add KMM unit test for repository layer (simple mock).
* CI: run backend tests + KMM build in GitHub Actions.
* Deliverable: End-to-end upsert (manual test documented).

**Week-1 Deliverables**

* Monorepo + CI skeleton
* Working GraphQL backend with Postgres connection
* KMM module with Apollo Kotlin setup
* Native apps able to call upsertProduct (manual form)
* README with run steps

**Week 2 — Search + barcode + admin polish (goal: product discovery + barcode scanner)**

**Day 6 — OpenSearch & indexer**

* Stand up OpenSearch (docker or managed).
* Create index mapping for products (name, description, tags, price, sku).
* Implement background indexer in backend: after upsertProduct, index to OpenSearch (sync for demo).

**Day 7 — Search backend & GraphQL**

* Add GraphQL query searchProducts(q, filters, limit, offset).
* Implement search resolver using OpenSearch queries (autocomplete + faceting).
* Add GraphQL types for search results and latency metadata.

**Day 8 — Mobile search UI**

* KMM: add SearchRepository to call searchProducts.
* iOS/Android: build search screens with autocomplete UI. Show query latency and highlight matched text.
* Measure and log response times for demo (display on screen).

**Day 9 — Barcode/QR scanning**

* Implement native camera/scan integration:
  + iOS: AVFoundation or ML Kit camera capture + barcode decode → send code to KMM.
  + Android: CameraX + ML Kit.
* KMM: expose handleScannedCode(code: String) method that resolves product details (GraphQL query getProductBySku).

**Day 10 — Polishing + demo script**

* Ensure admin can scan and upsert product with minimal UI.
* Document demo steps: scanning → product appears in search.
* Add tests:
  + OpenSearch indexing integration test.
  + Mobile manual test checklist.
* Deliverable: working search + barcode demo.

**Week-2 Deliverables**

* OpenSearch integrated & product index
* Search UI with autocomplete & latency display
* Barcode scan wired to product fetch/upsert
* Demo script: scan → index → search

**Week 3 — Offline cart, sync, checkout (goal: robust offline-first cart + safe checkout)**

**Day 11 — Choose & integrate Realm Kotlin**

* Add realm-kotlin to KMM shared module (commonMain).
* Define Realm schema for CartItem, PendingChange, and CachedProduct.
* Implement CartRepository in shared module:
  + addItem, removeItem, cartFlow() (Flow<Cart>), and syncPendingChanges().

**Day 12 — Offline flows**

* Mobile: UI for cart (add/remove/qty adjust); all operations immediate and local (Realm).
* KMM: implement SyncManager that:
  + Queues graphQL mutations in PendingChange.
  + On network available: performs mutations, marks pending as done.
  + Emits merge events to cartFlow().

**Day 13 — Server-side validation & idempotency**

* Backend: implement checkout mutation checkout(cart, idempotencyKey):
  + Validate coupon(s), prices, stock.
  + Atomically decrement stock using UPDATE products SET stock = stock - $qty WHERE id = $id AND stock >= $qty and check affected rows.
  + Use Postgres transaction; write order record; publish order.created.
* Implement idempotency check table to ignore duplicate checkouts by idempotencyKey.

**Day 14 — Coupon engine (simple)**

* Implement coupon filtering service:
  + Apply basic rules (min order value, applicable SKUs, expiry).
  + Return best coupon for an order (rule-based for demo).
* Add applyCoupons resolver that returns best coupons and discount amounts.

**Day 15 — Sync testing & QA**

* Test offline add → reconnect → sync → checkout end-to-end.
* Test conflict scenarios: price changed, insufficient stock on reconnect — ensure server authoritative resolution and user notification.
* Add UI conflict messages (e.g., “2 items reduced due to stock change”).
* Deliverable: offline cart sync + successful atomic checkout.

**Week-3 Deliverables**

* Realm-based offline cart in KMM
* checkout mutation with atomic stock decrement + idempotency
* Coupon engine + UI integration
* End-to-end offline → checkout demo scripts

**Week 4 — Recommendations, deploy & polish (goal: production-like demo deployed + monitoring + resume-ready artifacts)**

**Day 16 — Simple recommender & vector storage**

* For demo speed, use **pgvector** extension on Postgres or a small in-memory ANN (Faiss) locally.
* Build a small training pipeline (script) that:
  + Consumes orders (historical demo CSV or synthetic), trains simple embeddings (TF Recommenders or matrix factorization) OR computes simple item co-occurrence scores.
  + Stores item vectors in pgvector or a simple vector store.
* Expose GraphQL recommendedItems(userId, limit) which:
  + Retrieves user vector (or aggregates recent buys), queries nearest neighbors, returns items + score.
  + Apply business-rule re-ranking (stock, margin).

**Day 17 — Client integration**

* KMM: RecommendationRepository calling recommendedItems.
* UI: Home feed with recommended cards and fallback popular items if insufficient data.

**Day 18 — Deployment prep**

* Dockerize backend (backend, indexer worker).
* Docker Compose for local dev (Postgres + OpenSearch + Redis + backend).
* Create GitHub Actions workflows to build & push images.
* Prepare Render/Heroku deploy scripts or a simple Kubernetes manifest for GKE.

**Day 19 — Observability & CI**

* Integrate Sentry for backend and mobile (DSN in secret manager).
* Add Prometheus metrics endpoint and a simple Grafana dashboard (or use provider monitoring).
* Ensure CI runs: lint, tests, build artifacts.

**Day 20 — Final polish, README, demo & resume assets**

* Polish UI (small touches): animations, avatar, product images placeholder.
* Write README with:
  + how to run locally (docker-compose)
  + deploy instructions (Render/Heroku)
  + demo script (step-by-step)
  + architecture diagram + decisions
* Prepare resume bullets & portfolio text.
* Final full demo run — record a 5-minute screencast (optional).
* Deliverable: deployed backend URL + mobile app APK/IPA debug builds (or TestFlight link) and demo script.

**Week-4 Deliverables**

* Recommendation endpoint + UI
* CI/CD pipeline + deployed backend
* Observability + Sentry + README + demo script
* Resume-ready artifacts and recorded demo (optional)

**4 — Acceptance criteria / demo script (5 minute demo)**

Demo script to show recruiters/internally (5–7 minutes):

1. Admin demo (30s): Open Android app → Admin login → Scan barcode → Upsert price & stock → Save. (Show logs of mutation).
2. Search demo (30s): On iOS, type product name → autocomplete results within 50–150ms → open product.
3. Real-time sync demo (30s): After upsert, refresh search → new product appears instantly (shows OpenSearch index latency).
4. Offline cart demo (60s): Turn off network → add few items to cart → show local cart state → reconnect → show sync log and server validation (if any conflict).
5. Checkout demo (60s): Checkout with coupon applied → show backend logs of atomic order creation and stock decrement.
6. Recommendations demo (30s): Log in as different users → show different recommended feed items.
7. Deployment & monitoring (30s): Show deployed backend URL, Grafana dashboard, and Sentry event for a simulated error.

Acceptance: all flows work end-to-end on the demo environment; backend logs + monitoring show events; README includes run & deploy steps.

**5 — Deliverables you’ll get at the end**

* Monorepo with:
  + shared KMM module (Apollo Kotlin + Realm + repositories)
  + ios-app and android-app shells with connected flows
  + backend Node TS GraphQL with Postgres/OpenSearch integration
  + infra Dockerfiles and docker-compose for local dev
* CI workflow (GitHub Actions) for build/test
* Deployment instructions and a deployed backend (Render/Heroku or chosen provider)
* README, demo script, and 3–5 resume bullets tailored to this project
* Optional: short recorded demo video and sample dataset/scripts for recommendations

**6 — Risks & mitigations**

* Risk: **Realm Kotlin quirks / platform differences.**  
  Mitigation: If Realm sync proves tricky, fallback to SQLDelight + explicit sync manager (more code, more control).
* Risk: **OpenSearch infra complexity** (memory, JVM tuning).  
  Mitigation: Use managed OpenSearch or run very small index for demo; stub search if needed.
* Risk: **Vector DB complexity (Milvus) for recommendations.**  
  Mitigation: Use pgvector or Faiss for demo; upgrade later to Milvus if needed.
* Risk: **Mobile native camera integrations time-consuming.**  
  Mitigation: Implement camera stubs first (paste SKU), add camera scanning second. Use ML Kit for quick cross-platform decoder.

**7 — Tests to include**

* Unit tests: repositories (KMM), backend resolvers.
* Integration tests: backend GraphQL mutation → Postgres → OpenSearch indexing (run in CI with services).
* E2E manual checklist: scan → upsert → search → offline add → sync → checkout.
* Load test (optional): benchmark OpenSearch autocomplete latency with a script.

**8 — Resume / portfolio copy (paste-ready)**

* Built native iOS (SwiftUI) and Android (Jetpack Compose) commerce apps with a **Kotlin Multiplatform (KMM)**shared business layer (Apollo Kotlin + Realm Kotlin), enabling a single codebase for networking, cart logic, and offline sync.
* Designed & implemented a GraphQL backend (Node.js + TypeScript) with Postgres for ACID transactions, OpenSearch for sub-100ms product search, and idempotent checkout transactions ensuring correct stock decrements.
* Implemented personalized recommendations via embeddings (pgvector) and demonstrated real-time product indexing, offline-first cart sync, and deployable CI/CD pipeline (Docker + GitHub Actions).

**9 — Next steps (pick one and I’ll scaffold immediately)**

* a) Generate the **monorepo scaffold** (KMM shared + iOS/Android shells + Node TS backend + docker-compose).
* b) Create the **backend GraphQL schema + Postgres migrations + example resolvers** for product upsert, search, checkout.
* c) Produce the **KMM shared module** with Apollo Kotlin setup, ProductRepository, CartRepository skeleton, and Realm schema.
* d) Build a **demo dataset + recommender script** that trains simple embeddings and populates pgvector.

Tell me which artifact to create now (a / b / c / d) and I’ll scaffold it in this session.