ADR-1: Containerize Retail App with Docker Compose for Refunds, Metrics, and Reproducible Demos

Status
Accepted

Context
The Retail App has evolved into a multi-component system that includes:

- Node.js/Express backend with Prisma
- React frontend
- Aiven PostgreSQL database
- Supabase for storage and public URLs
- A new end-to-end returns/refund (RMA) workflow
- Observability features (logs, refund metrics, cycle time tracking)

Running this stack manually across developers and graders produced repeated issues:

- **Node.js/Prisma incompatibilities** leading to “Prisma client version mismatch” or migration failures.
- **Inconsistent Postgres schemas and seed data**, breaking critical flows (business verification, catalog upload, flash sales, refunds).
- **Slow onboarding** (3–4 hours) due to manual installs, DB setup, environment files, and migrations.
- **Unreliable refund demos** during Checkpoint 3 because the refund lifecycle requires:
    - seeded RefundRequest examples
    - Supabase photo URLs

    - a functional admin refund dashboard
    - metrics (RMA rate, refund cycle time)
    - logs for each state transition

Checkpoint 3 requires graders to run the **entire refund lifecycle** on their machines; achieving this reliably requires removing all environment drift.

Decision
Adopt a Docker + Docker Compose setup that runs the entire Retail App stack with one command, including the new returns/refund (RMA) and observability features.
The Compose topology includes:

1. **db** – PostgreSQL 14 with persistent volume `postgres_data`
2. **backend** – Node/Express API with Prisma and refund/metrics modules
3. **frontend** – React SPA served on port 3000

Standardize configuration via environment variables:
`DATABASE_URL`, Auth0 configs, Supabase keys/URLs, JWT secrets, and refund module settings.

On backend container startup:

- Run Prisma migrations automatically
- Seed the database with:
    - Admin user
    - Sample products
    - Test sales
    - At least one RefundRequest with Supabase URLs
    - Data for refund metrics (e.g., IN_INSPECTION count, approval rate baseline)

Integrate the RMA subsystem so refunds can be:

- created with Supabase photo evidence,
- transitioned across states in the admin dashboard,
- logged and measured via the metrics endpoint,
- demonstrated identically across machines.

Consequences

Positive

- **Reproducibility:** Every developer and grader uses identical Node, Prisma, and PostgreSQL versions—no more installation mismatch bugs.
- **Rapid onboarding:** A fully working system (refunds, metrics, catalog, flash sales) now starts in under 10 minutes via `docker compose up`.
- **Reliable refund demos:**
  Seeded refund data + containerized DB guarantee refund workflows behave the same across machines.
- **Unified observability:**
  Logs and refund metrics (cycle time, defective frequency, RMA rate) appear consistently in the backend container.
- **Improved testability:**
  Flash-sale tests, catalog upload tests, and refund reliability tests all run against a consistent, isolated database.

Negative

- Requires contributors and graders to understand Docker basics.
- Containers consume more memory/CPU than a minimal host-only setup.
- Must manage image rebuilds and caching carefully.
- External services (Supabase, Auth0) still sit outside Docker.

Alternatives Considered

- **Host-only execution (no Docker)**
  Rejected: maintains environment drift and breaks reproducibility, especially for Prisma migrations and refund features.

- **Partial containerization (backend or db only)**
  Rejected: still leaves fragile boundaries (local Node versions, local React builds), and fails the goal of unified end-to-end demos.

- **Local Kubernetes**
  Rejected: unnecessarily complex for a course project and difficult to support for graders.

Mapped Pattern and how it is applied

- **Environment Parity / Containerization Pattern**
  Backend, frontend, and database run in containers with the same configuration on every machine—preventing drift and ensuring refund demos are identical for graders.

- **Infrastructure-as-Code Pattern**
  Dockerfiles and docker-compose.yml define the environment in code (migrations, seeding, ports, Supabase URLs), making it version-controlled, repeatable, and testable.

- **Observability Pattern Integration**
  By containerizing the backend with refund logging/metrics, refund SLOs (RMA rate, cycle time) can be measured reliably at runtime using the exact same setup used during development and grading.

ADR-2: End-to-End Observability for Refunds, Metrics, and Business Ops

Status
Accepted

Context
The Retail App now includes complex return/refund (RMA) flows, multi-step admin verification, Supabase image uploads, payment/credit issuance, and time-based catalog features (flash sales, business pricing windows). Without observability, debugging refund failures, business catalog mismatches, and flash-sale anomalies becomes unpredictable.

During development of the returns module and metrics dashboard, we repeatedly encountered problems such as:

- Refunds stuck in “IN_INSPECTION” with no trace of what failed.
- Missing logs for failures in Supabase upload callbacks, leading to “orphaned” refund requests with no attached product image.
- Inconsistent refund metrics caused by missing event logging during approval or credit issuance.
- Admins unable to diagnose why a refund remained in APPROVED_AWAITING_SHIPMENT without cycle-time metrics.
- Difficulty validating SLOs for Checkpoint 3 (refund success rate, cycle time < X seconds, business upload accuracy).

Checkpoint 3 also requires **runtime demonstrations of refund transitions, RMA rate, and item-level logs**, which are impossible to show without a structured observability layer.

Decision
Implement a unified observability layer for the backend subsystem that provides:

1. **Structured JSON logging** across all controllers (refunds, catalog, checkout) with fields such as `refundId`, `orderId`, `businessId`, `stateTransition`, and `actorType`.
2. **Domain-level metrics** surfaced through a lightweight `/metrics` endpoint:
    - RMA rate (refunds / purchases)

    - Refund cycle time (request → closed)
    - Number of refunds in each state
    - Business catalog upload success rate

3. **Application health endpoints** (`/health/live`, `/health/ready`) for container orchestration and grader validation.
4. **Correlation IDs** automatically attached to each request to link logs, metrics, and error traces.
5. **Admin Dashboard integration** displaying refund metrics (approval rate, defective rate, cycle time) in real time.

RefundService, MetricsService, and MonitoringService now emit structured events during each of these steps:

- Create Refund → log + increment “refund_created”
- Approve Refund → log + increment “refund_approved"
- Issue Credit → log + increment “credit_issued”
- Error → log structured exception with correlationId

Consequences
Positive

- **Debuggable refund lifecycle:** Every transition (IN_INSPECTION → APPROVED_AWAITING_SHIPMENT → SHIPPED → COMPLETED) is logged and traceable.
- **Accurate refund KPIs:** Metrics like cycle time and RMA rate become visible and reproducible for Checkpoint 3 demonstrations.
- **Drastically easier testing:** CI can assert RMA counts, average approval time, and failure events directly.
- **Supports SLO-based grading:** Graders can see logs/metrics from inside the Dockerized backend exactly as written in the spec.

Negative

- Requires adding log/metrics calls to all refund and payment code paths.
- Developers must understand structured logging conventions.
- Requires consistent tagging for correlation IDs.

Alternatives Considered

- **Console.log everywhere**
  Rejected: ambiguous, unstructured, and impossible to trace refund state transitions.
- **External monitoring system (Prometheus/Grafana)**
  Rejected: too heavy for course constraints; increased operational complexity.
- **Tracing only, no metrics**
  Rejected: metrics (not just logs) required for RMA cycle-time measurement.

Mapped Pattern and how it is applied

- **Observability pattern**:
  Logs + metrics + health endpoints instrument the refund module, enabling measurable SLOs for refund reliability and cycle time.
- **Event logging pattern**:
  Each refund state change emits a structured event, enabling reliable refund analytics.
- **Correlation / Request-ID pattern**:
  Every refund operation becomes traceable through all involved services.

ADR-3: Resilience for Refunds, Catalog Uploads, and External Dependencies

Status
Accepted

Context
Returns and refunds rely on multiple external systems: Supabase (file storage), Aiven PostgreSQL (DB), Auth0 (authentication), and external payment logic (credit issuance via PaymentService).

Common issues observed during development:

- Refund approvals failing mid-transition, leaving refunds in inconsistent states (e.g., APPROVED but no credit issued).
- Supabase upload failures causing incomplete refund records.
- Network hiccups to Aiven causing temporary read/write delays.
- Admin actions (approve, deny, issue credit) breaking due to silent upstream timeouts.

Checkpoint 3 requires **demonstrating resilience under failure**, including:

- Stable refund state machine
- PROCESSING_FAILED fallback
- No orphaned refunds
- Safe credit issuance

Decision
Introduce a resilience layer across the refund, upload, and payment workflows:

1. **Timeouts** for all external calls (Supabase, Auth0, PaymentService, DB queries).
2. **Bounded retries with exponential backoff** for transient failures.
3. **Refund state-machine protection:**

    - Illegal or partial transitions automatically revert to PROCESSING_FAILED.

4. **Compensation logic** when payment reversal or credit issuance fails.

5. **Graceful degradation**:
   Metrics and logging failures do not block refunds.

6. **Atomic refund operations** via Prisma `$transaction` where possible.

Consequences
Positive

- Refund lifecycle remains consistent even during partial failures.
- No more “half-approved” refunds or broken credit issuance.
- Catalog uploads recover cleanly from intermittent Supabase outages.
- SLOs for refund reliability in Checkpoint 3 (≥95% successful transitions) achievable.

Negative

- More code complexity around retries, timeouts, and compensation.
- Requires careful tuning of retry limits and backoff intervals.

Alternatives Considered

- **No retries, rely on defaults**
  Rejected: unacceptable for refund consistency.
- **Saga orchestration / microservices**
  Rejected: too heavy for this stage; adds unnecessary operational overhead.
- **Circuit breakers on all external calls**
  Rejected for now: adds extra layers not required until real-world traffic.

Mapped Pattern and how it is applied

- **Resilience pattern (timeouts + retries)**:
  Ensures stable refund operations despite transient dependency failures.

- **Compensation (undo) pattern**:
  Rolls back refund steps when downstream processes fail.

- **Graceful degradation**:
  Non-critical services (metrics/logging) never block the refund pipeline.

ADR-4: Returns / Refunds (RMA) Module Design

Status
Accepted

Context
Checkpoint 3 introduces a full RMA (Return Merchandise Authorization) workflow including:

- Refund creation
- Supabase photo evidence
- IN_INSPECTION → APPROVED_AWAITING_SHIPMENT → SHIPPED → COMPLETED
- Defective item flagging
- Credit issuance
- Admin review dashboard
- RMA metrics (cycle time, approval rate)

Originally, refund logic was scattered across OrderService, PaymentService, and scattered controllers, causing:

- Inconsistent refund states
- Hard-to-debug refund failures
- Missing authorization boundaries
- No shared metrics or observability
- Broken flows during Supabase/DB latency

Decision
Introduce a **dedicated Returns module**, composed of:

- **RefundController + AdminRefundController** (presentation)
- **RefundService + RefundProcessor + MetricsService + MonitoringService** (business)
- **RefundRepository, DefectiveItemRepository, PaymentRepository** (persistence)
- A unified **RefundStatus enum** and strict **state machine**
- Supabase URL linking for refund images
- End-to-end metrics for RMA rate and cycle time
- A seeded refund record for demos via Docker Compose startup

Refund transitions now flow through one place—`RefundService`—ensuring atomic updates, compensations, and logging.

Consequences
Positive

- Centralized refund logic → predictable refund lifecycle
- Supports user-facing 3-step guided return UI
- Supports admin dashboard with reliable metrics
- Integrates cleanly with catalog, stock, payment, and Supabase
- Enables Checkpoint 3 SLO demonstrations (cycle time, success rate)

Negative

- Requires more boilerplate (new controllers/services/repos)
- Requires migrations for refund tables, defective items, refund transactions

Alternatives Considered

- **Embedding refunds into the Order controller**
  Rejected: leads to tangled logic and duplicated code.

- **Microservice for RMA**
  Rejected: too heavy at this stage, adds operational cost.

- **Minimal refund API without states**
  Rejected: insufficient for flash-sale returns, defective item flows, and admin tooling.

Mapped Pattern and how it is applied

- **Modularization / Separation of concerns**:
  Refund module isolated from checkout, catalog, and auth logic.

- **State machine pattern**:
  Ensures legal transitions, protects against partial failures.

- **Domain service pattern**:
  Orchestrates refunds, credits, uploads, and logs.