ADR 1. Documenting Design Choices for Three Lightweight Features

The following Architectural Decision Records (ADRs) outline the tactics and patterns for implementing the three lightweight features in Checkpoint 4, optimizing for performance, modifiability, and decoupled communication.

Feature 2.1: Order History Filtering & Search

Title	Implement Database Indexing for Order History Filtering & Search
Status	Proposed
Context (problem & background)	The Order History feature requires filtering by status, date range, and keyword search across large historical tables (`Sale`, `ReturnRequest`, etc.). Given the system's focus on high traffic volumes during peak load, searching large historical datasets using unindexed fields risks performing full table scans, leading to unpredictable latency and inefficient resource usage.
Decision	Implement **Database Indexing** and **Query Optimization** (Performance Tactic: Increase efficiency of resource usage). Indexes will be created on frequently filtered and sorted columns, such as `Sale.status`, `Sale._sale_date`, `ReturnRequest.status`, and foreign keys used for joins.
Consequences (pros/cons, trade-offs)	**Pros:** Significantly increases the speed and predictability of read operations, ensuring the user interface remains responsive even as history grows. It optimizes resource consumption by avoiding resource-intensive full table scans. **Cons:** Indexing adds slight computational overhead to write operations (inserts and updates).
Alternatives Considered	**Increase Resources (Vertical Scaling):** Rejected, as adding faster CPUs/memory is generally more expensive than optimizing processing strategies. Indexing provides targeted performance gain for read queries.

Title	Implement Data Caching for Order History Read Efficiency
Status	Proposed
Context (problem & background)	Historical data, once recorded, rarely changes. Repeated requests for recent order history (e.g., the last 5 completed sales shown on the index page or frequently filtered views) unnecessarily load the central PostgreSQL database. This contention for shared data negatively impacts the performance of other critical processes, such as concurrent stock updates (Performance goal P.2).

Decision	Implement **Data Caching** (Performance/Availability Tactic: Maintain multiple copies of data). Results from frequently executed, time-intensive historical read queries (like filtered order lists) will be stored in a local or distributed cache (e.g., Redis).
Consequences (pros/cons, trade-offs)	**Pros:** Reduces blocked time due to database contention, increasing overall system performance and throughput. By serving reads from the cache, it can maintain availability even if the database experiences transient latency issues. **Cons:** Requires a mechanism to manage cache memory and potential invalidation (though low risk for immutable historical data).
Alternatives Considered	**Data Replication:** Rejected, as data replication focuses on reducing write contention or improving redundancy for mutable data. Caching is more efficient and appropriate for immutable historical data reads.

Title	Implement Layered Service Abstraction (Layers Pattern) for History Logic
Status	Proposed
Context (problem & background)	The Order History and Returns History functionality spans multiple data entities and complex joining/filtering logic (Sale, SaleItem, ReturnRequest, Product). Integrating this complex logic directly into the application controller layer or mixing it with persistence details increases coupling, making future modifications (e.g., changing search providers or adding new return statuses) costly and risky.
Decision	Implement a **Layered Service Abstraction** using the **Layers Pattern** (Modifiability Pattern). A dedicated `HistoryService` layer will encapsulate all business logic related to retrieving, filtering, and preparing historical records, decoupling the API controllers from the underlying database schema.
Consequences (pros/cons, trade-offs)	**Pros:** Greatly enhances **modifiability**, allowing the logic within the lower layer (`HistoryService`) to change (e.g., query optimization or adding new data sources) without affecting the higher layers (UI controllers). This structural decision reduces the number of interfaces any team must understand. **Cons:** Layering may introduce a small performance penalty due to call traversal.
Alternatives Considered	**Direct Integration:** Rejected, as mixing database queries and presentation logic violates separation of concerns and leads to high coupling, making changes costly.

Feature 2.2: Low Stock Alerts

Title	Implement Publish-Subscribe Pattern for Low Stock Alerts
Status	Proposed
Context (problem & background)	The low stock alert feature requires the Inventory logic to notify the Admin Dashboard whenever a product's stock drops below a predefined threshold. Direct invocation creates tight coupling between the `InventoryService` (data producer) and the `Dashboard` (consumer), reducing modifiability. The system already relies on an intermediary message bus for decoupling reporting services (Integrability I.2).
Decision	Implement the **Publish-Subscribe Pattern** using the existing Message Queue infrastructure. The Inventory logic will act as a **Publisher**, broadcasting a generic "InventoryUpdated" event whenever stock changes. A dedicated Alert Listener (Subscriber) consumes these events, checks the low-stock threshold, and updates the UI notification state.
Consequences (pros/cons, trade-offs)	**Pros:** Achieves extreme **loose coupling** (publishers have no knowledge of subscribers). This increases modifiability, as adding a new consumer (e.g., an automated re-order service) requires zero changes to the core inventory module. **Cons:** Introducing asynchronous communication can negatively impact performance (latency) and makes system behavior harder to reason about, as implementation is implicitly invoked.
Alternatives Considered	**Client-Server Pattern:** Rejected, as the consumer (Dashboard/Alerting Service) would need prior knowledge of the producer (Inventory Service), resulting in high coupling.

Feature 2.3: Return/Refund Notifications

Title	Implement Publish-Subscribe Pattern for RMA Status Notifications
Status	Proposed
Context (problem & background)	The Returns & Refunds workflow includes multiple status changes (e.g., `AUTHORIZED`, `REFUNDED`) which must trigger immediate notifications for the end user. Directly updating the UI notification panel from the `ReturnsService` creates coupling, limiting the ability to add external notification channels (like email/SMS) later.
Decision	Implement the **Publish-Subscribe Pattern** for status changes. The `ReturnsService` (Publisher) broadcasts events like "RMAStatusChanged" via the Message Queue. A lightweight UI service (Subscriber) consumes these events to refresh the notification panel, while external services could subscribe later for additional notification types.

Consequences (pros/cons, trade-offs)	**Pros:** Decouples the core RMA/Refund logic from the consumer-specific communication mechanism (notification UI). This supports high **integrability** for future external notification services. Events are easily logged, supporting system auditing. **Cons:** Asynchronous delivery adds a small latency overhead before the notification is received by the user.
Alternatives Considered	**Direct Database Polling:** Rejected, as constantly polling the database for status updates wastes resources and adds computational overhead, violating performance goals.

ADR 2. Documenting Documentation or Repository Organization Improvements