# 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

## Implement Database Indexing for Order History Filtering & Search

- **Status:** Accepted
- **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.

## Implement Data Caching for Order History Read Efficiency

- **Status:** Accepted
- **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.

## Implement Layered Service Abstraction (Layers Pattern) for History Logic

- **Status:** Accepted
- **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 History Service 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

## Implement Publish-Subscribe Pattern for Low Stock Alerts

- **Status:** Accepted
- **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 threshol. Direct invocation creates tight coupling between the Inventory Service (data producer) and the Dashboard (consumer), reducing modifiability. The system already relies on an intermediary message bus for decoupling reporting services (Integrability 1.2).

- **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

## Implement Publish-Subscribe Pattern for RMA Status Notifications

- **Status:** Accepted
- **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 Returns Service 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

The following Architectural Decision Records document improvements to documentation structure, repository organization, and observability practices implemented during Checkpoint 4.

## 2.1 Structured Service Layer Organization

- **Status:** Accepted
- **Context (problem & background):** As the application grew through checkpoints, services were added ad-hoc without a clear organizational pattern. The `src/services/` directory contained a mix of domain services (ReturnsService, RefundService), infrastructure services (InventoryService), and tactical implementations. This made it difficult for developers to locate functionality and understand service dependencies.
- **Decision:** Organize services into distinct categories based on their architectural role:
  - **Domain Services:** Business logic services (ReturnsService, RefundService, HistoryService)
  - **Infrastructure Services:** Cross-cutting concerns (InventoryService, NotificationService, LowStockAlertService)
  - **Integration Services:** External system adapters (PartnerCatalogService, FlashSaleService)
- **Consequences (pros/cons, trade-offs):**
  - **Pros:** Clearer mental model for developers. Easier to identify where new functionality should be added. Reduces cognitive load when navigating the codebase.
  - **Cons:** Requires refactoring imports when reorganizing. Minor overhead in maintaining the organizational pattern.
- **Alternatives Considered:** Feature-based organization (grouping by business feature like returns/, orders/). Rejected because it can lead to code duplication for shared infrastructure and makes cross-cutting concerns harder to implement.

## 2.2 Consistent ADR Documentation Format

- **Status:** Accepted
- **Context (problem & background):** Previous checkpoints used varying formats for Architectural Decision Records, making it difficult to compare decisions across checkpoints or understand the rationale for existing implementations.
- **Decision:** Standardize all ADRs to follow the format:
  - **Status:** (Proposed, Accepted, Deprecated, Superseded)
  - **Context (problem & background):** Clear problem statement
  - **Decision:** Specific architectural choice with pattern/tactic reference
  - **Consequences (pros/cons, trade-offs):** Balanced analysis
  - **Alternatives Considered:** At least one alternative with rejection reasoning
- **Consequences (pros/cons, trade-offs):**

    - **Pros:** Consistent format improves searchability and comprehension. Explicit alternatives help future developers understand why certain paths were not taken.
    - **Cons:** More upfront documentation effort per decision.
- **Alternatives Considered:** Lightweight ADRs with only decision and date. Rejected because the context and alternatives are critical for future maintainability and onboarding.

# 2.3 Enhanced Observability Integration in New Features

- **Status:** Accepted
- **Context (problem & background):** Checkpoint 3 established observability patterns (metrics, structured logging, events) but new features in CP4 needed consistent integration with these patterns. Without explicit guidance, new features might bypass observability, creating blind spots in system monitoring.
- **Decision:** All new CP4 features must integrate with the existing observability stack:
    - **Counters:** For tracking feature usage (e.g., `low_stock_alerts_total`, `notifications_created_total`)
    - **Events:** For significant state changes (e.g., `rma_status_changed`, `inventory_updated`)
    - **Structured Logging:** For debugging and audit trails
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Unified monitoring across all features. Easier to debug issues in production. Events can be consumed by multiple downstream systems (dashboards, alerting).
    - **Cons:** Slight performance overhead from metric collection. Increased code complexity in each feature.
- **Alternatives Considered:** Per-feature custom logging. Rejected because it would fragment the observability model and make dashboard creation more complex.

# 2.4 Configuration-Driven Feature Behavior

- **Status:** Accepted
- **Context (problem & background):** Hard-coded values for feature behavior (e.g., low stock thresholds, page sizes) make it difficult to tune system behavior without code changes and redeployment.
- **Decision:** All configurable parameters for CP4 features are exposed via the centralized `Config` class with environment variable support:
    - `LOW_STOCK_THRESHOLD` (default: 5) - Threshold for low stock alerts
    - `ORDER_HISTORY_PAGE_SIZE` (default: 20) - Pagination size for order history
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Runtime tunability without code changes. Easy to override in different environments (dev, staging, production). Centralized configuration reduces scattered magic numbers.
    - **Cons:** Additional indirection when tracing default values.
- **Alternatives Considered:** Database-stored configuration. Rejected for lightweight features because it adds unnecessary complexity and database dependency for simple parameters.

# 2.5 Repository Cleanup and Structure Validation

- **Status:** Accepted
- **Context (problem & background):** Over multiple checkpoints, the repository accumulated unused files, outdated configurations, and inconsistent directory structures. This technical debt impedes onboarding and increases the risk of confusion.
- **Decision:** Perform repository cleanup as part of CP4:
    - Verify `deploy/dockercompose.yml` still works with new features
    - Ensure all tests run (mark known failures with clear TODOs)
    - Remove unused code and outdated configurations
    - Validate directory structure matches Checkpoint 1 layout guidelines
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Cleaner repository improves developer experience. Reduces confusion from obsolete artifacts. Docker deployment confidence.
    - **Cons:** Time investment in cleanup activities rather than new features.
- **Alternatives Considered:** Deferring cleanup to a future maintenance sprint. Rejected because accumulated technical debt compounds over time and the final checkpoint is the appropriate time to ensure repository quality.