Architecture Design Package: Cloud-Native E-Commerce Platform

1. Project Overview

1.1 Problem Summary

System Description: A scalable, microservices-based e-commerce backend that centralizes product inventory, identity management, and order fulfillment. It exposes secure, event-driven APIs to support web storefronts and administrative dashboards.

Users:

- **Customers:** Public users who browse catalogs, manage carts, and purchase goods.
- **Admins:** Internal staff who manage inventory, view business analytics, and handle support.

Problem Solved: Traditional monolithic e-commerce platforms often suffer from tight coupling, making them hard to scale during traffic spikes (e.g., Black Friday) and difficult to update without downtime. This architecture solves these issues by decoupling domains (Product vs. Order vs. User) to ensure high availability and independent scalability.

1.2 Core User Flows

1. **User Sign Up:** A new user registers; credentials are hashed and stored securely.
2. **Browse Products:** High-volume read operations where users filter and search the catalog.
3. **Place Order (Checkout):** A transactional flow where inventory is reserved, payments are processed, and an asynchronous notification sequence is triggered.
4. **Admin Update Inventory:** A privileged write operation to update stock levels.
5. **Async Order Notification:** A background process that sends email confirmations without blocking the user interface.

(See USER_FLOWS.md for detailed sequence diagrams of unique flows).

1.3 Requirements

Functional Requirements:

- Users must be able to search and view product details.
- Users must be able to add items to a persistent cart.
- The system must process payments securely via third-party gateways.
- Admins must be able to view aggregate order history.

Non-Functional Requirements:

- **Scalability:** The Product Service must handle 10k+ concurrent read requests during sales.
- **Consistency:** Inventory counts must be strictly consistent (ACID) to prevent overselling.
- **Availability:** The core browsing experience should remain available even if the Notification Service fails (Graceful Degradation).
- **Security:** All PII (Personal Identifiable Information) must be encrypted; API access must be guarded by JWTs.

2. Architecture

2.1 High-Level Architecture Diagram

2.2 Architectural Style Decision

Decision: Microservices Architecture (managed via Container Orchestration).

Justification:

- **Scalability:** E-commerce traffic is asymmetrical. Users browse (read) 100x more than they buy (write). Microservices allow us to scale the Product Service independently of the User Service to save costs.
- **Fault Isolation:** If the Notification Service crashes, users can still place orders. In a monolith, a memory leak in email generation could crash the entire checkout.
- **Team Autonomy:** Distinct domains allow separate teams to work on "Checkout" vs. "Discovery" without constant merge conflicts.

2.3 Component Breakdown

- **API Gateway:** The single entry point. Handles routing, rate limiting, and SSL termination.
- **User Service:** Manages identity, authentication (issuing JWTs), and profiles.
- **Product Service:** Manages catalog data, categories, and inventory state.
- **Order Service:** The transactional heart. Handles cart logic, order creation, and state transitions (Pending -> Paid -> Shipped).
- **Notification Service:** A worker service that consumes events to send emails/SMS.

3. API Design Package

3.1 API Paradigm Decision

Decision: REST (Representational State Transfer) for client-server communication; Async Events for internal decoupling.

Justification:

- **REST:** It is the industry standard for public-facing APIs (Frontends). It caches well (GET requests) and is easy for third-party developers to integrate with.
- **Why not GraphQL?** While GraphQL prevents over-fetching, it adds complexity to

caching and rate-limiting at the Gateway level. For an MVP, REST provides a predictable & cacheable surface area.

3.2 Representative REST Endpoints

GET /products

- *Description:* specific list of products with pagination.
- *Response:*
```
{
  "data": [
    { "id": "prod_123", "name": "Sneakers", "price": 99.00 }
  ],
  "page": 1,
  "total": 50
}
```

POST /orders

- *Description:* Creates a new order from the user's current session cart.
- *Request:* {"payment_method": "stripe_tok_abc", "shipping_address": {...}}
- *Response (201 Created):*
  { "order_id": "ord_789", "status": "PENDING" }

PATCH /admin/products/{id}

- *Description:* Admin updates inventory count.
- *Request:* {"stock_qty": 50}
- *Response:* 200 OK

3.3 Authentication & Authorization

Strategy: JWT (JSON Web Tokens) with RBAC (Role-Based Access Control).

Justification:

- **Statelessness:** JWTs are self-contained. The Order Service can verify a user's identity by decoding the token signature without asking the User Service database on every request. This reduces latency.
- **RBAC:** We define scopes in the token (role: "admin" vs role: "customer"). Middleware in the services checks these scopes to allow/deny actions (e.g., only Admins can PATCH products).

4. Data Model & Storage

4.1 Database Choice & Justification

Decision: PostgreSQL (Relational/SQL).

Justification:

- **ACID Compliance:** Financial transactions (Orders) and Inventory management require strict consistency. We cannot risk "Eventual Consistency" (common in NoSQL) where a user buys an item that is already out of stock.
- **Relational Structure:** E-commerce data is highly relational (Users have Orders, Orders have Items, Items are Products). SQL joins handle this naturally.
- **Why not MongoDB?** While MongoDB is good for unstructured catalogs, the lack of multi-document transactions (historically) and complex joining makes it risky for the core ledger of an e-commerce platform.

4.2 Data Schema

- **Boundary 1 (PostgreSQL):** Contains Users, Products, Orders, OrderItems.
- **Boundary 2 (Redis):** Contains ephemeral Sessions and Carts.

4.3 Data Access & Patterns

- **Read Pattern (Through-Cache):**
  - Requests for Products first check **Redis**.
  - *Hit:* Return data immediately (sub-millisecond).
  - *Miss:* Query PostgreSQL, store result in Redis with a 10-minute TTL (Time To Live), then return.
- **Write Pattern (Transactional):**
  - Orders write directly to PostgreSQL inside a transaction block to ensure Order creation and Inventory deduction happen atomically.
- **Syncing:**
  - The Search Service (Elasticsearch) is kept in sync via domain events. When a product is updated in SQL, a ProductUpdated event is emitted to the queue, which the Search Service consumes to update its index.

5. Implementation Team Proposals

5.1 MVP Team Structure (Speed Focus)

Total Staff: 6 Developers (2 Squads)

1. **Product Squad (4 Devs):** Full-stack responsibility. They build the React Frontend and the core Product/Order services simultaneously to minimize blocking.
2. **Platform Squad (2 Devs):** They build the "skeleton"—User Auth, Database setup, CI/CD pipelines, and Gateway configuration.

5.2 Phase I Team Structure (Scale Focus)

Total Staff: 10+ Developers (Domain Squads)
Once we move to Phase I (Release), we shift to Vertical Slicing:

1. **Discovery Team:** Owns the Frontend, Search Service, and Product Service. KPI: Conversion Rate.
2. **Checkout Team:** Owns the Order Service, Payments, and Cart. KPI: Successful Transactions per second.
3. **Core Platform Team:** Owns Infrastructure (Kubernetes), Security, and Notification systems. KPI: System Uptime (99.9%).