Mastering System Design

Design an E-Commerce Marketplace Platform (aka Amazon)

What Are We Building?

- A multi-vendor e-commerce platform for buyers and sellers.
- Core functionalities:
    - Browse and search products
    - Manage inventory and product listings
    - Handle shopping cart and secure checkout
    - Process payments and detect fraud
- Must be scalable, secure, and consistent.

Functional Requirements

- User Registration & Authentication (Buyers/Sellers)
- Product Catalog Management
- Inventory Consistency during Checkout
- Shopping Cart and Order Management
- Secure Payment Processing
- Basic Fraud Detection and Alerts
- Seller Dashboard (sales, products, payouts)
- Users:
    - Buyers: Search, browse, add to cart, pay, track orders.
    - Sellers: List products, manage stock, handle orders, track earnings.
    - Administrators: Moderate content, monitor platform health, resolve disputes.

Non-Functional Requirements

- ✅ Performance:
    - Fast product search and browsing (< 300ms latency for common queries)
- ✅ Scalability:
    - Handle initial load of ~10,000 concurrent users
    - Elastic growth as user base expands
- ✅ Availability:
    - 99.9% uptime target
    - Fault-tolerant core services (catalog, cart, checkout)
- ✅ Security:
    - Secure user authentication (OAuth2/JWT)
    - Secure payment processing (PCI-DSS compliance best practices)
- ✅ Consistency:
    - Strong consistency for inventory updates during checkout
- ✅ Maintainability:
    - Modular service design (e.g., catalog service, payment service)
    - Easy onboarding of new sellers and products

Key System Design Challenges

- Inventory Accuracy: Prevent overselling at high scale.
- High Availability: Handle millions of buyers and sellers online.
- Secure Payments: Ensure data security and prevent fraud.
- System Scalability: Grow seamlessly as user base increases.
- Efficient Search: Support quick product browsing across huge catalogs.

Assumptions and Constraints

- ✅ Assumptions:
    - Users have stable internet access (no offline mode required).
    - External payment gateway (e.g., Stripe, Razorpay) will handle payment processing.
    - No warehouse or logistics management initially — sellers handle shipping.
    - Product catalog size is manageable (~500K products initially).
    - Fraud detection is basic, rule-based (no ML-based advanced detection yet).
- ✅ Constraints:
    - MVP must launch within 3–4 months.
    - Budget constraints limit infrastructure choices (e.g., cloud services preferred over heavy on-prem setups).
    - Minimal DevOps/SRE support — systems must be simple to operate.
    - Limited personalization (search is keyword-based, no ML recommendations).
    - Regulatory compliance must be ensured for payments (PCI DSS basics, GDPR for user data).

Scale Estimations

- Users: ~1 Million registered users; ~10,000 daily active users (DAUs) at launch
- Traffic: Peak ~100 requests/second during sales
- Product Catalog: ~500,000 products
- Orders: ~1,000 orders/day initially
- Sellers: ~5,000 active sellers

Critical System Bottlenecks

- Search and Browsing Latency: Need fast, scalable product search
- Inventory Consistency: Prevent overselling during flash sales
- Checkout and Payment Flows: Ensure reliability under peak traffic
- Database Hotspots: Popular products causing heavy reads/writes
- Fraud/Payment Errors: Detect and handle abnormal transactions quickly

Core Services Breakdown

- **User Service** – Manage registration, login, authentication
- **Product Catalog Service** – Manage product listings, search, categories
- **Inventory Service** – Track stock levels, handle stock reservations
- **Order Management Service** – Manage cart, checkout, order lifecycle
- **Payment Service** – Handle payment processing, retries
- **Notification Service** – Send email/SMS notifications
- **Admin Service** – Seller/product moderation, platform controls

Service APIs and Responsibilities

- User Service APIs:
  a. POST /register, POST /login, GET /profile
- Catalog APIs:
  a. GET /products, GET /products/{id}, POST /products
- Inventory APIs:
  a. GET /inventory/{productId}, POST /reserve-stock
- Order APIs:
  a. POST /cart/add, POST /checkout, GET /order/{orderId}
- Payment APIs:
  a. POST /payments/initiate, POST /payments/verify

✅ APIs should be RESTful, stateless, and versioned (v1).

Service Communication Approach

- Synchronous (HTTP REST):
  - User, Product Catalog, Inventory, Orders, Payments
- Asynchronous (Events via Message Queue):
  - Order Placed → Inventory Update → Notification
  - Payment Success → Order Confirmation → Notify Seller

✅ Use event-driven async flows for critical but non-blocking operations (e.g., stock updates, sending emails).

Database and Storage Choices

- User Data
  - Relational DB (e.g., PostgreSQL, for strong consistency)
- Product Catalog
  - Search-optimized DB (e.g., Elasticsearch + PostgreSQL backup)
- Inventory Data
  - Relational DB (e.g., PostgreSQL, strict consistency on stock)
- Order Data
  - Relational DB (PostgreSQL, transactional)
- Payment Records
  - Relational DB (PostgreSQL, encrypted fields)
- Logs and Events
  - Object Storage (e.g., AWS S3 or similar)

✅ Relational databases for critical business data.

✅ ElasticSearch or Redisearch for fast product search indexing.

Caching and Queueing Strategy

- ✅ Caching (using Redis):
  - Frequently accessed product data (popular products, categories)
  - User session tokens (if server-side sessions)
  - Stock levels (read-heavy optimization)
- ✅ Queues (using Kafka/SQS/RabbitMQ):
  - Order placement events → trigger inventory update
  - Payment success events → trigger order confirmation and notifications
  - Asynchronous notification dispatch (emails/SMS)

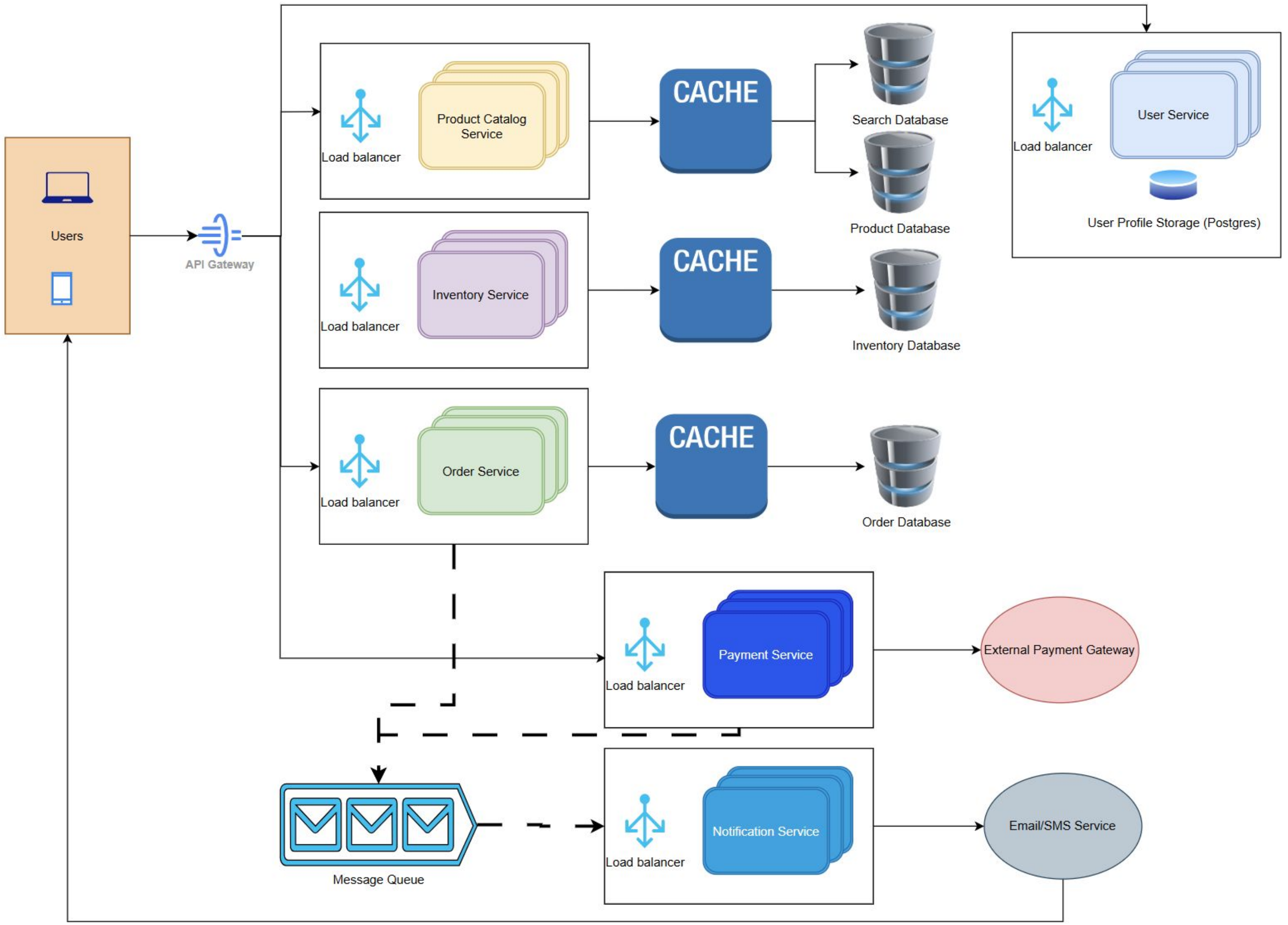
✅ Caching = Speed up reads;

✅ Queues = Decouple heavy async tasks from real-time flows.

Strategic Tech & Infra Decisions

- ✅ Database Choice:
  - PostgreSQL for core transactional data
  - Elasticsearch for product search indexing
- ✅ Caching Layer:
  - Redis for product caching, session management, and stock-level fast reads
- ✅ Message Queues:
  - Kafka (preferred) or AWS SQS for async event-driven flows
- ✅ Hosting/Cloud:
  - AWS / GCP / Azure managed services (focus on managed DBs, object storage, serverless queues)
- ✅ Authentication & Authorization:
  - OAuth 2.0 / OpenID Connect; JWT tokens for user sessions
- ✅ Payment Integration:
  - External gateway like Stripe or Razorpay (focus on PCI compliance)
- ✅ System Communication:
  - REST APIs for synchronous flows
  - Event-driven architecture for critical async processes

The Final Design - ECommerce Platform



Users
API Gateway
Load balancer
Product Catalog Service
CACHE
Search Database
Product Database
Load balancer
User Service
User Profile Storage (Postgres)
Load balancer
Inventory Service
CACHE
Inventory Database
Load balancer
Order Service
CACHE
Order Database
Load balancer
Payment Service
External Payment Gateway
Message Queue
Load balancer
Notification Service
Email/SMS Service