E-commerce Platform System Design

System Architecture

Service Decomposition Strategy

I opted for a domain-driven microservices architecture over a monolith for the following reasons:

- **Scalability**: Independent scaling of services based on demand
- **Resilience**: Isolated failure domains prevent system-wide outages
- **Development velocity**: Smaller, focused teams can work independently
- **Technology flexibility**: Different services can use appropriate technologies

Key Architecture Layers

1. **Client Layer**
    - Web Applications, Mobile Apps, Admin Portal, Third-party Integrations, API consumers
    - Each client interacts through the API Gateway
2. **API Management Layer**
    - API Gateway: Entry point for all client requests, handling routing, composition, and protocol translation
    - Rate Limiter: Prevents abuse and ensures fair resource allocation
    - Auth Service: Centralizes authentication and authorization
3. **Core Services Layer**
    - User Service: Account management, preferences, permissions
    - Product Service: Catalog management, pricing, availability
    - Catalog Service: Categories, search indexes, product relationships
    - Order Service: Order processing, status tracking
    - Payment Service: Payment processing, refunds, financial records
    - Cart Service: Shopping cart management
    - Review Service: Product reviews and ratings
    - Flash Sales Service: Limited-time promotion management
    - Inventory Service: Stock tracking and availability
    - Recommendation: Personalized product suggestions

4. **Infrastructure Services Layer**

    - Service Registry: Service discovery and registration
    - Config Server: Centralized configuration management
    - Logging Service: Aggregated logging and analysis
    - Monitoring Service: Health checks and metrics collection
    - Circuit Breaker: Fault tolerance and service isolation
5. **Asynchronous Processing Layer**

    - Message Queue: Reliable message delivery (Kafka/RabbitMQ)
    - Event Bus: Real-time event distribution
    - Task Scheduler: Delayed and scheduled tasks
6. **Data Storage Layer**

    - Relational Databases: Transactional data (Users, Orders, Payments)
    - NoSQL Databases: Product catalog, reviews, flexible schema data
    - Caching Layer: Multi-level caching strategy
    - Data warehousing and analytics storage

Database Design & Scaling

Data Distribution Strategy

- **Relational data**: PostgreSQL for primary transactional data, with MySQL read replicas
- **Document data**: MongoDB for product catalog and reviews
- **Key-value data**: DynamoDB for session data and feature flags

Scaling Strategies

- **Horizontal partitioning**: Sharding by user ID for user-specific data
- **Vertical partitioning**: Database-per-service pattern
- **Read replicas**: Scale read capacity independently
- **Connection pooling**: Efficient database connection management
- **CQRS pattern**: Separate read and write models for high-traffic services

Data Consistency

- **Strong consistency**: For financial transactions and inventory updates
- **Eventual consistency**: For product catalog and user preferences
- **Transaction patterns**: Saga pattern for distributed transactions

Caching Architecture

Multi-layer Caching

- **Client-side cache**: Browser and mobile app caching
- **CDN cache**: Static assets, images, and product media
- **API Gateway cache**: API response caching
- **Distributed cache layer**: Redis clusters for session data, product catalog, user preferences

Cache Policies

- **LRU eviction**: For general-purpose caches
- **TTL-based**: For time-sensitive data
- **Cache-aside pattern**: For database query results
- **Write-through**: For data that must be persisted
- **Cache invalidation**: Event-based cache clearing
- **Flash sales priority**: Special handling for high-demand events

Asynchronous Processing

Event-driven Architecture

- Message Queue for high-throughput events (Kafka)
- RabbitMQ for task management
- Consumer services process events asynchronously

Use Cases

- Inventory updates
- Order processing
- Email notifications
- Analytics events
- User action tracking
- Real-time updates (flash sales)

Fault Tolerance & High Availability

Resilience Patterns

- Circuit breakers to prevent cascading failures
- Bulkhead pattern for resource isolation
- Retry with exponential backoff
- Fallback mechanisms for degraded functionality

Service Health Management

- Health check APIs for all services
- Automated service restarts
- Self-healing deployment with Kubernetes

Multi-region Deployment

Geographic Distribution

- Primary and secondary regions with active-active configuration
- Cross-region replication for databases and caches
- Global load balancing and DNS services

Failover Strategy

- Automated failover for critical services
- Data replication with minimal lag
- Global CDN with edge caching

API Rate Limiting

Protection Mechanisms

- Token bucket algorithm implementation
- User-based, IP-based, and service-based limits
- Burst handling for legitimate traffic spikes
- Distributed rate limit counters using Redis

Security Considerations

Authentication & Authorization

- OAuth 2.0 with JWT tokens
- Role-based access control
- Fine-grained permissions

Data Protection

- Encryption in transit and at rest
- PCI DSS compliance for payment processing
- Data minimization and retention policies

API Security

- Request validation and sanitization

- Protection against common attack vectors
- Monitoring for suspicious activities

Monitoring & Observability

Observability Stack

- Distributed tracing across services
- Centralized logging with correlation IDs
- Real-time metrics and dashboards
- Alerting based on business-critical thresholds

Performance Measurement

- Key performance indicators (KPIs) tracking
- User experience metrics
- Business impact metrics (conversion rate, cart abandonment)