Digital Commerce Platform - Microservices Architecture Design

---

1. Architecture Evolution

Monolith → Modular Monolith → Microservices

Phase 1: Monolithic Architecture

- **Single Deployment Unit**: Entire system deployed as one application with tightly coupled components (UI, business logic, data access) sharing a single database
- **Challenges**: Scaling bottlenecks, long deployment cycles, technology lock-in, and difficult to maintain as codebase grows

Phase 2: Modular Monolith

- **Logical Separation**: Refactor codebase into distinct modules (Customer, Orders, Payments, Inventory) with well-defined interfaces and clear boundaries, but still deployed as single unit
- **Database Schema Separation**: Create separate schemas within the same database for each domain to establish data ownership
- **API Layer Introduction**: Implement internal API contracts between modules to prepare for future service extraction
- **Benefits**: Reduced coupling, easier testing, team ownership of modules, while maintaining deployment simplicity

Phase 3: Microservices Architecture

- **Service Extraction**: Extract high-value modules (Orders, Payments) as independent services with their own databases (Database-per-Service pattern)
- **Distributed Communication**: Implement synchronous communication (REST/gRPC) for request-response patterns and asynchronous messaging (Kafka/RabbitMQ) for event-driven workflows
- **Independent Deployment**: Each service can be deployed, scaled, and updated independently with CI/CD pipelines per service
- **Data Ownership**: Each microservice owns its data exclusively - no direct database access across services, only through APIs or events
- **Integration Strategy**: API Gateway for external clients, Service Mesh for internal service-to-service communication, Event Mesh for asynchronous event distribution
- **Operational Complexity**: Requires distributed tracing, centralized logging, service discovery, and robust monitoring to manage the distributed nature

---

2. Service Boundaries Definition

Selected Domains: Customer, Orders, and Payments

2.1 Customer Service

Responsibilities

- Manage customer profile, authentication, and preferences
- Handle customer registration, login, and profile updates
- Store customer addresses, contact information etc
- **Entities**: Customer, Address, ContactInfo
- **Data Ownership**: Customer database (PostgreSQL)

APIs (Synchronous - REST)

1. Create Customer

POST /api/v1/customers

Content-Type: application/json

Request Body:

```
{
  "email": "",
  "firstName": "",
  "lastName": "",
  "phone": ",
```

```
  "address": {
    "street": "",
    "city": "",
    "state": "",
    "zipCode": "",
    "country": ""
  }
}
```

Response: 201 Created

```
{
  "customerId": "cust_123abc",
}
```

2. Get Customer

GET /api/v1/customers/{customerId}

Response: 200 OK

```
{
  "customerId": "cust_123abc",
  "email": "",
  "firstName": "",
  "lastName": "",
  "phone": "",
  "addresses": [...]
}
```

3. Update Customer

PUT /api/v1/customers/{customerId}

Content-Type: application/json

Request:

```
{
  "phone": ""
}
```

Response: 200 OK

Events Published (Asynchronous - Kafka)

1. CustomerCreated

Topic: customer.events

Event: CustomerCreated

```
{
  "eventId": "evt_123xyz",
  "eventType": "CustomerCreated",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "customerId": "cust_123abc",
    "email": "",
    "firstName": "",
    "lastName": ""
  }
}
```

2. CustomerUpdated

Topic: customer.events

Event: CustomerUpdated

```
{
  "eventId": "evt_124xyz",
  "eventType": "CustomerUpdated",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "customerId": "cust_123abc",
    "updatedFields": ["phone", "email"]
  }
}
```

Communication Pattern

- **Synchronous (REST)**: Customer profile operations require immediate validation and response (registration, login, profile retrieval) - users expect instant feedback
- **Asynchronous (Events)**: Customer lifecycle events (created, updated) are published for other services (Orders, Notifications) to maintain eventual consistency without tight coupling

---

2.2 Orders Service

Responsibilities

- Manage order lifecycle (creation, updates, cancellation)

- Orchestrate order fulfillment workflow
- Track order status and history
- Coordinate with Payments, Inventory, and Shipping services
- **Entities**: Order, OrderItem, OrderStatus, OrderHistory
- **Data Ownership**: Orders database (PostgreSQL)

APIs (Synchronous - REST)

1. Create Order

POST /api/v1/orders

Content-Type: application/json

Authorization: Bearer {token}

Request:

```
{
  "customerId": "cust_123abc",
  "items": [
    {
      "productId": "prod_456def",
      "quantity": 2,
      "price": 100.00
    }
  ],
  "shippingAddress": {
    "street": "",
    "city": "",
    "state": "",
```

```
    "zipCode": ""
  },
  "paymentMethodId": "pm_123ghi"
}

Response: 201 Created
{
  "orderId": "ord_abc123",
  "status": "pending_payment",
  "totalAmount": 200.00,
  "createdAt": "",
  "estimatedDelivery": ""
}
```

2. Get Order Details

```
GET /api/v1/orders/{orderId}
Authorization: Bearer {token}

Response: 200 OK
{
  "orderId": "ord_abc123",
  "customerId": "cust_123abc",
  "status": "confirmed",
  "items": [...],
  "totalAmount": 200.00,
  "paymentStatus": "paid",
```

```
  "shippingStatus": "processing",
  "createdAt": "",
  "updatedAt": ""
}
```

3. Cancel Order

```
POST /api/v1/orders/{orderId}/cancel
Authorization: Bearer {token}

Response: 200 OK
{
  "orderId": "ord_abc123",
  "status": "cancelled"
}
```

4. List Customer Orders

```
GET /api/v1/customers/{customerId}/orders?page=1&limit=20
Authorization: Bearer {token}

Response: 200 OK
{
  "orders": [...],
  "pagination": {
    "page": 1,
    "limit": 10,
    "total": 15
```

```
  }
}
```

Events Published (Asynchronous - Kafka)

1. OrderCreated

Topic: orders.events

Event: OrderCreated

```
{
  "eventId": "evt_order_001",
  "eventType": "OrderCreated",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "orderId": "ord_abc123",
    "customerId": "cust_123abc",
    "items": [
      {
        "productId": "prod_456def",
        "quantity": 2,
        "price": 100.00
      }
    ],
    "totalAmount": 200.00,
    "status": "pending_payment"
  }
}
```

2. OrderStatusChanged

Topic: orders.events

Event: OrderStatusChanged


```
{
  "eventId": "evt_order_002",
  "eventType": "OrderStatusChanged",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "orderId": "ord_abc123",
    "previousStatus": "pending_payment",
    "newStatus": "confirmed",
    "reason": "payment_successful"
  }
}
```


3. OrderCancelled

Topic: orders.events

Event: OrderCancelled


```
{
  "eventId": "evt_order_003",
  "eventType": "OrderCancelled",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "orderId": "ord_abc123",
    "customerId": "cust_123abc",
```

```
    "reason": "",
    "refundAmount": 200.00
  }
}
```


Events Consumed (Asynchronous - Kafka)

1. PaymentCompleted (from Payments Service)

2. PaymentFailed (from Payments Service)

3. InventoryReserved (from Inventory Service)

Communication Pattern

- **Synchronous (REST)**: Order creation and retrieval require immediate response to users - customers need instant order confirmation and tracking
- **Synchronous to Payments**: Payment authorization requires immediate validation before order confirmation (strong consistency requirement)
- **Asynchronous (Events)**: Order status changes are broadcast to multiple consumers (Notifications, Analytics, Shipping) - decouples services
- **Event-Driven Core**: Orders service acts as orchestrator, reacting to payment and inventory events to progress order state

---

2.3 Payments Service

Responsibilities

- Process payment transactions (authorize, capture, refund)
- Integrate with payment gateways (Stripe, PayPal, etc.)
- Maintain payment history
- Handle payment method management
- **Entities**: Payment, PaymentMethod, Transaction, Refund
- **Data Ownership**: Payments database (PostgreSQL with encryption)

APIs (Synchronous - REST)

1. Authorize Payment

POST /api/v1/payments/authorize

Content-Type: application/json

Authorization: Bearer {token}

Request:

```
{
  "orderId": "ord_abc123",
  "customerId": "cust_123abc",
  "amount": 200.00,
  "paymentMethodId": "pm_123ghi"
}
```

Response: 200 OK

```
{
  "paymentId": "pay_def456",
  "status": "authorized",
```

```
  "amount": 200.00,
  "expiresAt": ""
}
```

2. Capture Payment

POST /api/v1/payments/{paymentId}/capture

Content-Type: application/json

Request:

```
{
  "amount": 200.00
}
```

Response: 200 OK

```
{
  "paymentId": "pay_def456",
  "status": "captured",
  "capturedAmount": 200.00,
  "capturedAt": ""
}
```

3. Refund Payment

POST /api/v1/payments/{paymentId}/refund

Content-Type: application/json

Request:

```
{
  "amount": 200.00,
  "reason": ""
}
```

Response: 200 OK

```
{
  "refundId": "ref_ghi789",
  "paymentId": "pay_def456",
  "status": "refunded",
  "refundedAmount": 200.00,
  "refundedAt": ""
}
```

4. Get Payment Status

GET /api/v1/payments/{paymentId}

Authorization: Bearer {token}

Response: 200 OK

```
{
  "paymentId": "pay_def456",
  "orderId": "ord_abc123",
  "status": "captured",
  "amount": 200.00,
  "paymentMethod": {
    "type": "card",
```

```
    "last4": "4242"
  },
  "createdAt": ""
}
```

Events Published (Asynchronous - Kafka)

1. PaymentAuthorized

Topic: payments.events

Event: PaymentAuthorized

```
{
  "eventId": "evt_pay_001",
  "eventType": "PaymentAuthorized",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "paymentId": "pay_def456",
    "orderId": "ord_abc123",
    "customerId": "cust_123abc",
    "amount": 200.00,
    "status": "authorized"
  }
}
```

2. PaymentCompleted

Topic: payments.events

Event: PaymentCompleted

```
{
```

```
  "eventId": "evt_pay_002",
  "eventType": "PaymentCompleted",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "paymentId": "pay_def456",
    "orderId": "ord_abc123",
    "customerId": "cust_123abc",
    "amount": 200.00,
    "status": "captured"
  }
}
```


3. PaymentFailed

Topic: payments.events

Event: PaymentFailed


```
{
  "eventId": "evt_pay_003",
  "eventType": "PaymentFailed",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "paymentId": "pay_def456",
    "orderId": "ord_abc123",
    "customerId": "cust_123abc",
    "amount": 200.00,
    "errorCode": "insufficient_funds",
```

```
    "errorMessage": "Card declined - insufficient funds"
  }
}
```

4. PaymentRefunded

Topic: payments.events

Event: PaymentRefunded

```
{
  "eventId": "evt_pay_004",
  "eventType": "PaymentRefunded",
  "timestamp": "",
  "version": "1.0",
  "data": {
    "refundId": "ref_ghi789",
    "paymentId": "pay_def456",
    "orderId": "ord_abc123",
    "amount": 200.00,
    "reason": ""
  }
}
```

Communication Pattern

- **Synchronous (REST)**: Payment operations (authorize, capture, refund) require immediate response for strong consistency - Orders service needs instant confirmation to proceed

- **Asynchronous (Events)**: Payment lifecycle events are published for downstream consumers (Orders, Notifications, Analytics) to react without blocking payment processing
- **Security**: All payment APIs require authentication, use HTTPS, and sensitive data is encrypted at rest and in transit

---

3. Integration Patterns Selection

3.1 API Gateway Pattern

Services Using This Pattern

- **All External-Facing APIs**: Customer Service, Orders Service, Payments Service

1. Centralized Entry Point

- Single point of entry for all client applications (web, mobile, partners)
- Simplifies client-side logic by providing unified API interface

2. Cross-Cutting Concerns

- **Authentication & Authorization**: Validates JWT tokens, enforces OAuth2 flows before routing to backend services
- **Rate Limiting**: Prevents abuse by limiting requests per customer (e.g., 1000 req/hour per API key)
- **Request/Response Transformation**: Converts external API contracts to internal service formats
- **API Composition**: Aggregates data from multiple services (e.g., order details + customer info + payment status in single response)

3. Security Enforcement

- SSL/TLS termination at gateway
- API key management and validation

4. Observability

- Centralized logging of all API requests
- Metrics collection (latency, error rates, throughput)
- Request tracing with correlation IDs

Capabilities Provided

- **Routing**: Intelligent request routing based on path, headers, or query parameters

- **Load Balancing**: Distributes traffic across service instances
- **Caching**: Response caching for frequently accessed data (e.g., product catalog)
- **Protocol Translation**: REST to gRPC conversion for internal services
- **API Versioning**: Supports multiple API versions (v1, v2) simultaneously

Limitations

1. Single Point of Failure

- If API Gateway goes down, all external traffic is blocked
- Mitigation: Deploy in high-availability mode with multiple instances across availability zones

2. Performance Bottleneck

- All requests pass through gateway, can become bottleneck under extreme load if not properly scaled

3. Increased Complexity

- Additional infrastructure component to manage, configure, and monitor
- Requires expertise in gateway-specific configuration (Kong, AWS API Gateway, Apigee)

3.2 Service Mesh Pattern

Services Using This Pattern

- **All Internal Service-to-Service Communication**: Customer ↔ Orders, Orders ↔ Payments, Orders ↔ Inventory

1. Service Discovery

- Automatic service registration and discovery without hardcoded endpoints
- Services communicate via logical names (e.g., payments-service) instead of IP addresses
- Handles dynamic scaling and instance replacement transparently

2. Traffic Management

- **Load Balancing**: Intelligent load balancing with health checks (round-robin, least connections, weighted)
- **Circuit Breaking**: Prevents cascading failures by stopping requests to unhealthy services
- **Retry Logic**: Automatic retries with exponential backoff for transient failures
- **Timeout Management**: Enforces timeouts to prevent hanging requests

3. Security (mTLS)

- Automatic mutual TLS encryption for all service-to-service communication
- Certificate management and rotation
- Zero-trust security model - every service verifies identity of caller

4. Observability

- Distributed tracing with automatic span creation (Jaeger/Zipkin integration)

- Detailed metrics per service pair (latency percentiles, error rates, request volume)
- Service dependency graphs and traffic flow visualization

Capabilities Provided

- **Resilience**: Circuit breakers, retries, timeouts etc
- **Security**: mTLS, authorization policies, certificate management
- **Observability**: Metrics, logs, traces without application code changes
- **Traffic Control**: Canary deployments, A/B testing, traffic splitting

Limitations

1. Operational Complexity

- Requires running sidecar proxies alongside every service instance, increasing resource consumption (CPU, memory)
- Complex control plane to manage (Istio control plane)

2. Performance Overhead

- Sidecar proxy adds latency due to additional network traversal
- Increased memory footprint can be significant for large deployments

3. Debugging Difficulty

- Network issues become harder to diagnose with additional proxy layer
- Requires specialized tools and expertise to troubleshoot

4. Platform Lock-In

- Tight coupling to specific service mesh implementation (Istio, Consul Connect)

- Migration between mesh providers is complex and risky

3.3 Event Mesh Pattern (Message Broker - Kafka)

Services Using This Pattern

- **Asynchronous Communication**: Orders Service, Payments Service, Customer Service, Notifications Service, Analytics Service

1. Event-Driven Architecture

- Decouples producers and consumers - services don't need to know about each other
- Enables pub-sub model where multiple services can react to same event (e.g., OrderCreated → Notifications, Analytics, Inventory)
- Supports event sourcing and CQRS patterns for audit trails

2. High Throughput & Scalability

- Kafka handles millions of messages per second with horizontal scaling
- Partitioning enables parallel processing across consumer groups
- Ideal for high-volume scenarios (50k orders/hour, 1M price updates/hour)

3. Durability & Reliability

- Messages persisted to disk with configurable retention (days/weeks)
- Replication across brokers ensures no data loss

- Consumers can replay events from any point in time for recovery or reprocessing

4. Temporal Decoupling

- Producers and consumers don't need to be online simultaneously
- Consumers can process events at their own pace without blocking producers

Capabilities Provided

- **Message Persistence**: Durable storage with configurable retention policies
- **Ordering Guarantees**: Per-partition ordering for related events
- **Consumer Groups**: Load balancing across multiple consumer instances
- **Stream Processing**: Real-time event processing with Kafka Streams

Limitations

1. Eventual Consistency

- Asynchronous nature means data is eventually consistent across services, not immediately
- Not suitable for operations requiring strong consistency (e.g., payment authorization)
- Requires cautious design to handle out-of-order events and duplicate processing

2. Operational Complexity

- Kafka cluster management requires specialized expertise (ZooKeeper, broker tuning)
- Monitoring and troubleshooting distributed system issues (partition rebalancing, consumer lag)
- Requires infrastructure for Schema Registry, Kafka Connect, monitoring tools

3. Message Ordering Challenges

- Global ordering not guaranteed across partitions
- Requires cautious partition key selection to maintain ordering for related events

4. Increased Latency

- Asynchronous processing introduces latency
- Not suitable for real-time user-facing operations requiring immediate feedback
- Debugging event flows across multiple services is complex with distributed tracing

---

4. High Traffic Scenario Design

Traffic Requirements

- **Price Updates**: 1M updates/hour (~278 updates/second)
- **Orders**: 50k orders/hour (~14 orders/second)
- **Inventory Adjustments**: Real-time updates with order placements

---

4.1 Integration Pattern Selection for High Load

1. Orders Service

- **Pattern**: API Gateway (external) + Service Mesh (internal) + Kafka (primary integration)
- **Load**: 50k orders/hour = ~14 orders/second
- Synchronous order creation API with immediate response
- Kafka for order events distribution (OrderCreated, OrderStatusChanged)
- Event-driven orchestration for order fulfillment workflow
- Database sharding by customer ID for write scalability

3. Payments Service

- **Pattern**: API Gateway (external) + Service Mesh (internal) + Kafka (notifications) + SQS (fallback)
- **Load**: 50k payment transactions/hour = ~14 transactions/second

- Synchronous payment authorization (critical path - cannot be async)
- Kafka for payment events (PaymentCompleted, PaymentFailed)
- SQS DLQ for failed payment processing retries
- Idempotency layer to prevent duplicate charges

4. Pricing Service (Additional for 1M updates/hour)

- **Pattern**: Kafka (primary) + Service Mesh (internal queries)
- **Load**: 1M updates/hour = ~278 updates/second
- Kafka as primary ingestion mechanism for price updates
- Batch processing with Kafka Streams for aggregation
- Cache-aside pattern with Redis for price lookups
- Read-through cache to reduce database load

5. Inventory Service (Additional for real-time adjustments)

- **Pattern**: Kafka (events) + Service Mesh (synchronous checks)
- **Load**: ~14 reservations/second (aligned with orders)
- Synchronous inventory check during order creation
- Kafka for inventory adjustment events
- Optimistic locking for concurrent inventory updates

4.2 Message Broker Selection: Kafka vs Kinesis vs SQS

Selected: Apache Kafka (Primary) + Amazon SQS (Fallback)

Kafka - Primary Event Backbone

Use Cases:

- Order events (OrderCreated, OrderStatusChanged, OrderCancelled)
- Payment events (PaymentCompleted, PaymentFailed, PaymentRefunded)
- Customer events (CustomerCreated, CustomerUpdated)
- Price update streams (1M updates/hour)
- Inventory adjustment events

1. Throughput Capacity

- Handles 1M+ messages/second per cluster with proper partitioning
- Easily accommodates 1M price updates/hour (278/sec) and 50k orders/hour (14/sec)
- Linear scalability by adding brokers and partitions

2. Low Latency

3. Message Ordering

- Per-partition ordering guarantees
- Critical for order state (pending → confirmed → shipped → delivered)
- Partition by orderId or customerId to maintain event sequence

4. Durability & Replay

- Configurable retention (7-30 days typical)

- Consumers can replay events for recovery or new consumer onboarding

5. Stream Processing

- Kafka Streams for real-time aggregations (order metrics, pricing analytics)
- Ideal for real-time inventory calculations and pricing rules

Scaling Strategy:

- **Partitions**: 12 partitions per topic (orders.events, payments.events) for parallel processing
- **Replication Factor**: 3 for high availability
- **Consumer Groups**: Multiple consumer groups for different use cases (notifications, analytics, fulfillment)

Amazon SQS - Fallback & DL Queues

Use Cases:

- DL queue for failed payment processing
- Retry queue for transient failures
- Asynchronous tasks with lower throughput requirements (email notifications)

1. Simplicity

- Fully managed service with no operational overhead
- Automatic scaling without partition management
- Ideal for simple point-to-point messaging

2. Reliability

- At-least-once delivery guarantee
- Message visibility timeout for retry handling

- DL queue for poison messages

3. Cost-Effective for Low Volume

- Pay-per-request pricing suitable for error scenarios
- No need to maintain infrastructure for edge cases

Why NOT Kinesis:

- Higher latency than Kafka
- More expensive for high-throughput scenarios
- Less flexible for complex stream processing
- Shard management overhead similar to Kafka partitions

Why NOT SQS as Primary:

- No message ordering guarantees (except FIFO queues with limited throughput)
- FIFO queues limited to 3000 messages/second (insufficient for price updates)
- No replay capability - messages deleted after consumption
- Not suitable for pub-sub patterns (one message, multiple consumers)

---

5. Architecture Diagram

Diagram Components:

1. **External Layer**: Web/Mobile clients, Partner APIs
2. **API Gateway**: Kong/AWS API Gateway with authentication, rate limiting, routing
3. **Microservices Layer**: Customer Service, Orders Service, Payments Service (each with dedicated database)
4. **Service Mesh**: Istio sidecars for mTLS, observability, traffic management
5. **Event Mesh**: Kafka cluster with topics (customer.events, orders.events, payments.events)
6. **Data Layer**: PostgreSQL databases (one per service), Redis cache cluster
7. **Infrastructure**: Schema Registry, Monitoring (Prometheus/Grafana), Distributed Tracing (Jaeger)