E-Commerce Microservices - Extension & Enhancement Plan

Document Version: 1.0
Date: February 17, 2026
Author: System Architecture Team
Status: Planning Phase

📋 Table of Contents

---

1. Executive Summary

Current State

- ✅ **2 Microservices**: User Auth (8081), Product Catalog (8080)
- ✅ **Technologies**: Spring Boot 3.5.4, Java 21, MySQL, MongoDB, Redis, Kafka
- ✅ **Communication**: Feign (sync), Kafka (async)
- ✅ **Security**: JWT authentication with RBAC
- ✅ **Additional**: GCP Pub/Sub POC for vehicle messaging

Proposed Enhancements

- 🎯 **6 New Services**: Orders, Payments, Inventory, Notifications, Gateway, Service Registry
- 🎯 **Advanced Patterns**: Saga, Event Sourcing, CQRS, Circuit Breakers
- 🎯 **Observability**: Distributed tracing, centralized logging, metrics
- 🎯 **Search**: Elasticsearch integration
- 🎯 **Deployment**: Kubernetes-ready with auto-scaling

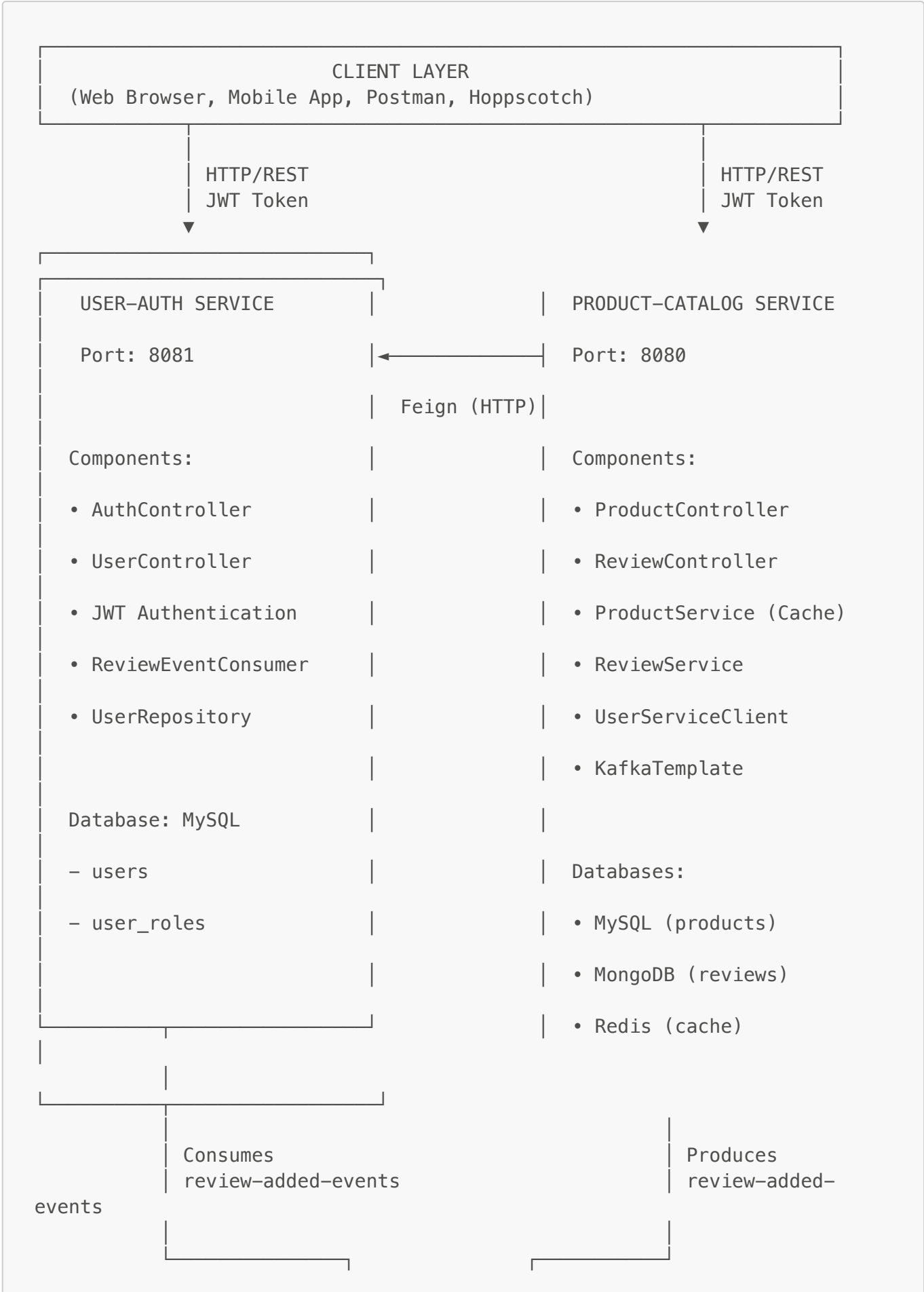
Business Value

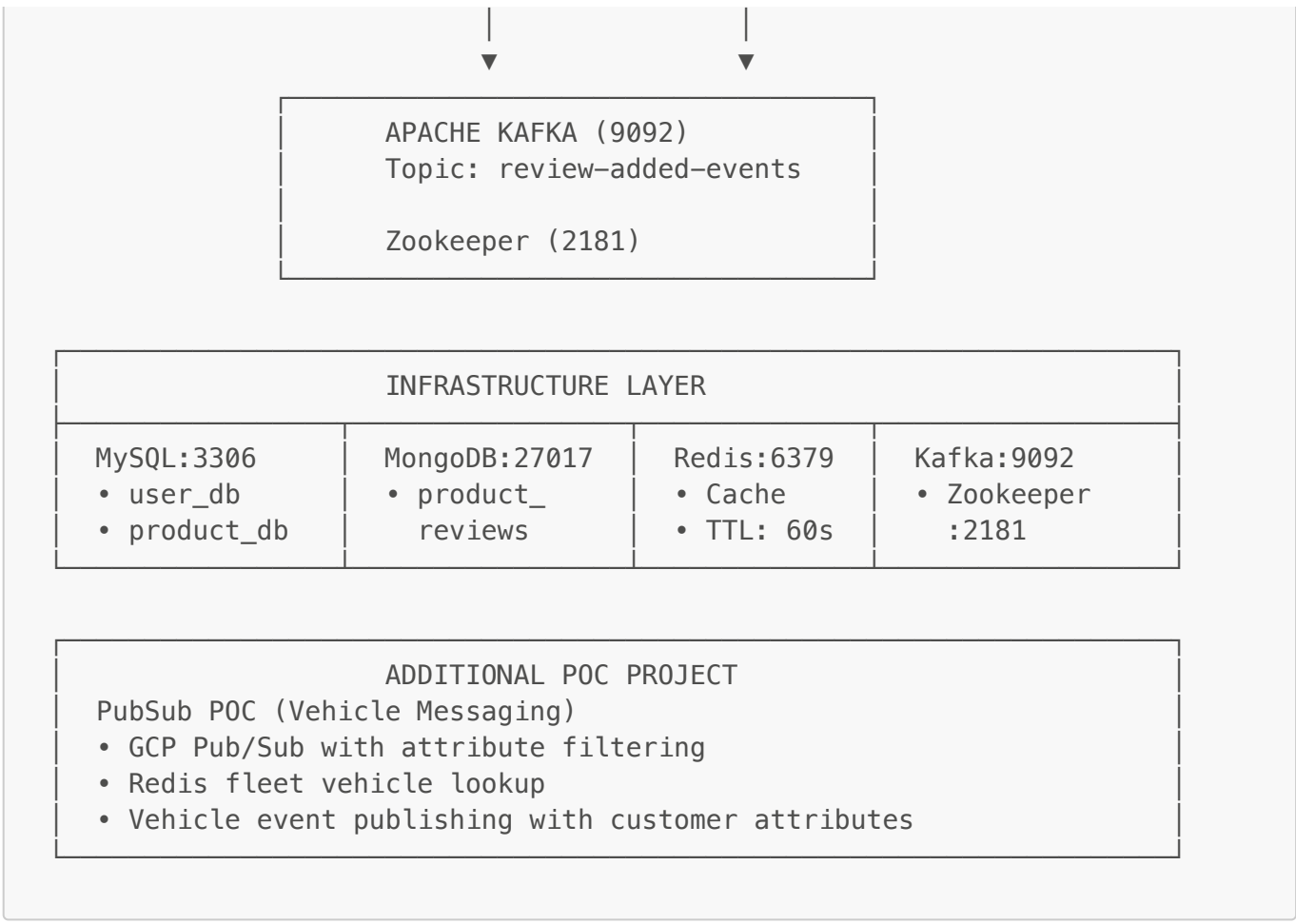
- **Complete E-commerce Flow**: Cart → Order → Payment → Fulfillment
- **Improved Reliability**: 99.9% uptime with circuit breakers and retries
- **Better Performance**: Sub-100ms response times with multi-level caching
- **Enhanced UX**: Real-time notifications, advanced search, recommendations

---

2. Current Architecture

2.1 Architecture Diagram (As-Is)





2.2 Current Data Flow

```
┌─────────────────────────────────────────────────────────────┐
│  FLOW 1: USER REGISTRATION & LOGIN                          │
└─────────────────────────────────────────────────────────────┘

User → POST /api/auth/register
       ↓
   AuthController
       ↓
   1. Check username exists
   2. Hash password (BCrypt)
   3. Save to MySQL
   4. Assign ROLE_USER
       ↓
   Response: "User registered successfully"

User → POST /api/auth/login
       ↓
   AuthController
       ↓
   1. Authenticate (AuthenticationManager)
   2. Load UserDetails
   3. Generate JWT token (HS256)
       ↓
   Response: { token: "eyJhbGc..." }
```

```
┌──────────────────────────────────────────────────────────────────┐
│  FLOW 2: PRODUCT MANAGEMENT WITH CACHING                         │
└──────────────────────────────────────────────────────────────────┘

GET /api/products/{id} + JWT
       ↓
   JWT Filter validates token
       ↓
   ProductController
       ↓
   ProductService.getById()
       ↓
   Check Redis cache
       ├─ Cache HIT → Return cached data (5ms)
       └─ Cache MISS → Query MySQL (50ms)
                    → Store in Redis
                    → Return data


┌──────────────────────────────────────────────────────────────────┐
│  FLOW 3: ADD REVIEW (CROSS-SERVICE + KAFKA EVENT)                │
└──────────────────────────────────────────────────────────────────┘

POST /api/reviews + JWT
{
  "productId": 1,
  "userId": 101,
  "rating": 5,
  "comment": "Excellent product!"
}
       ↓
   JWT Filter (extract roles)
       ↓
   ReviewController (@PreAuthorize)
       ↓
   ReviewService.addReview()
       ↓
    ┌──────────────────────────────────┐
    │ 1. Validate User (Feign Call)    │
    │    GET http://localhost:8081/    │
    │         api/users/101            │
    │    ↓                             │
    │    User Service → MySQL query    │
    │    ↓                             │
    │    Return UserDto or 404         │
    └──────────────────────────────────┘
       ↓ User exists ✓
   2. Save review to MongoDB
   3. Publish to Kafka
       ↓
   KafkaTemplate.send("review-added-events", event)
       ↓
```

```
┌─────────────────────────────────┐
│   KAFKA BROKER                  │
│   Topic: review-added-events    │
└─────────────────────────────────┘
    ↓
User Service (ReviewEventConsumer)
    ↓
@KafkaListener
    ↓
Log event (TODO: Send email/SMS)
```

2.3 Technology Stack (Current)

Component	Technology	Version	Purpose
Language	Java	21	Modern language features
Framework	Spring Boot	3.5.4	Microservices foundation
Security	Spring Security	6.5.2	Authentication/Authorization
JWT	jjwt	0.12.6	Token generation/validation
Database (Relational)	MySQL	8.0	Users, Products
Database (Document)	MongoDB	5.0	Reviews
Cache	Redis	7.0	Product caching
Messaging	Apache Kafka	3.0	Event streaming
Service Communication	OpenFeign	4.3.0	REST client
Build Tool	Gradle	8.10	Dependency management
Testing	JUnit 5 + PITest	-	Unit + Mutation testing
Documentation	OpenAPI/Swagger	3.0	API documentation
Cloud (POC)	GCP Pub/Sub	-	Vehicle messaging

2.4 Current Limitations

Architectural Gaps

1. ❌ **No Service Discovery**: Hardcoded URLs in Feign client
2. ❌ **No Circuit Breakers**: Service failures cascade
3. ❌ **No API Gateway**: Multiple ports, no centralized security
4. ❌ **No Distributed Tracing**: Hard to debug cross-service issues
5. ❌ **No Centralized Logging**: Logs scattered across services

Functional Gaps

1. ❌ **No Order Management**: Can't complete purchases

2. ❌ **No Payment Integration**: No revenue generation
3. ❌ **No Inventory Tracking**: Overselling risk
4. ❌ **No Notifications**: No email/SMS alerts
5. ❌ **No Search**: Basic SQL queries, no full-text search

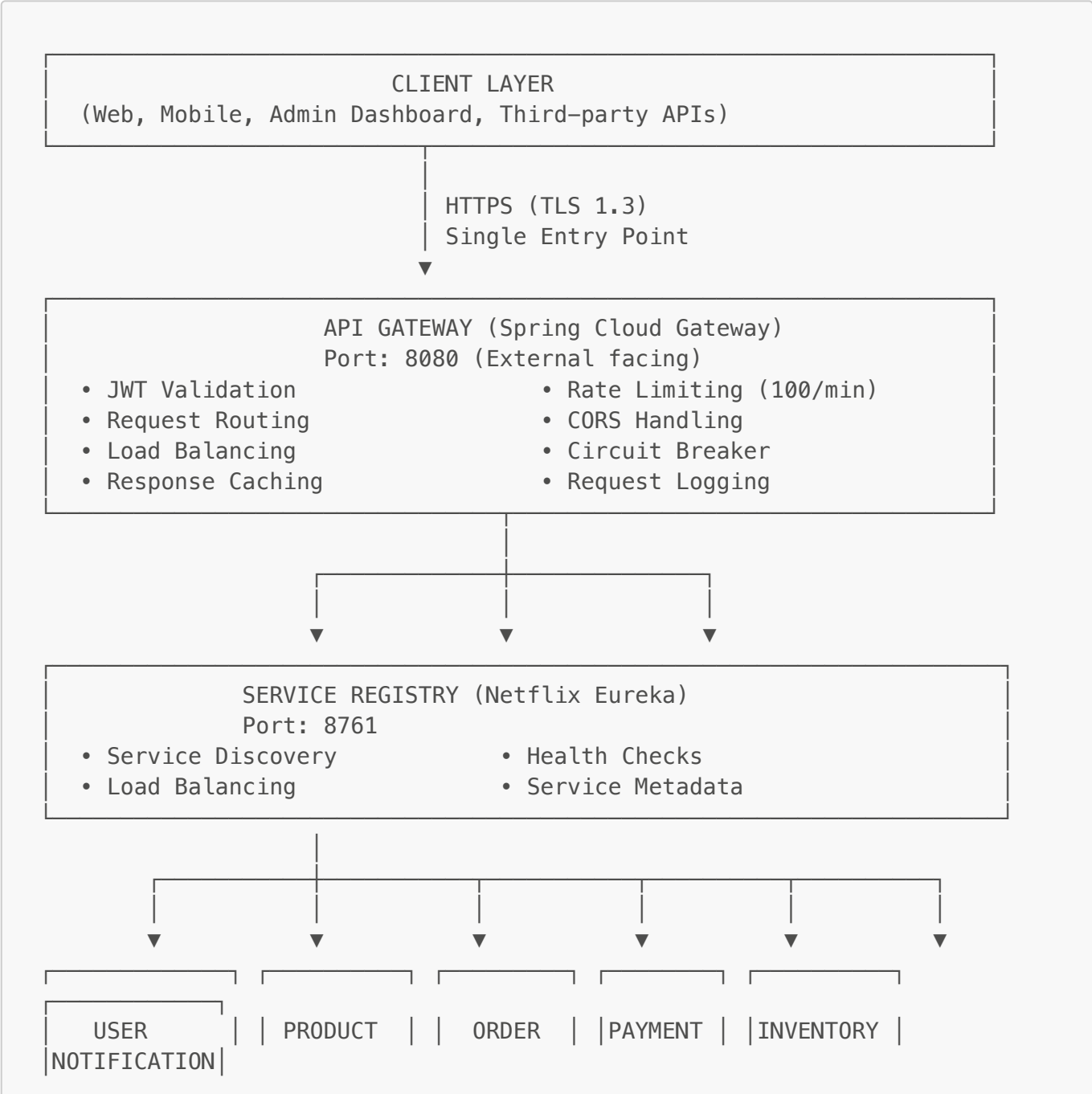
Operational Gaps

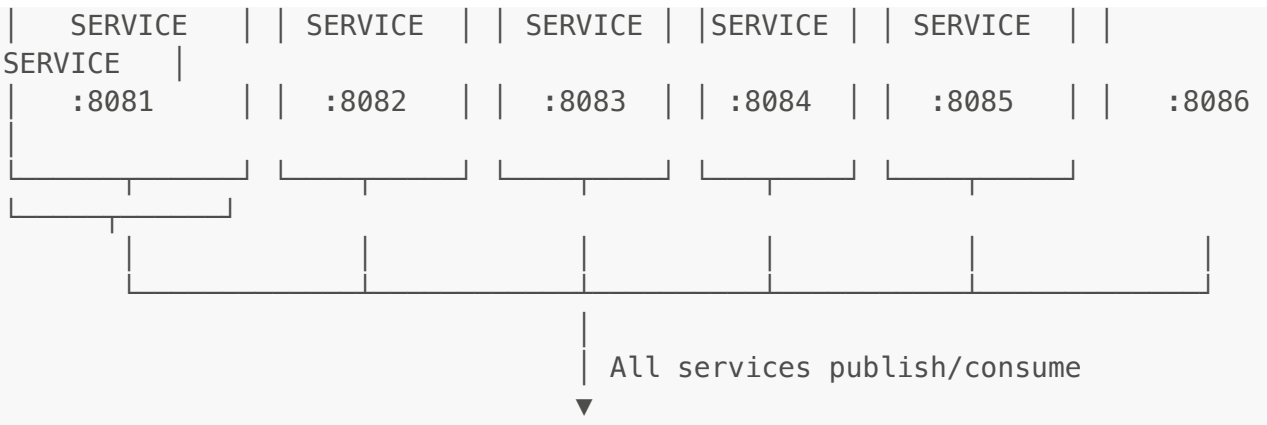
1. ❌ **Manual Scaling**: No auto-scaling capability
2. ❌ **No Health Monitoring**: Limited observability
3. ❌ **No Deployment Automation**: Manual deployments
4. ❌ **No Rollback Strategy**: Risky releases

---

3. Proposed Architecture

3.1 Target Architecture Diagram (To-Be)





EVENT BUS (Apache Kafka)
Port: 9092

Topics:
- review-added-events
- order-created-events
- order-paid-events
- order-shipped-events
- inventory-reserved-events
- payment-completed-events
- notification-events
- inventory-updated-events

DATA LAYER (Polyglot Persistence)

MySQL	MongoDB	PostgreSQL	Redis
• Users	• Reviews	• Orders	• Cache (L2)
• Products	• Logs	• Events	• Session
			• Rate Limiter

SEARCH LAYER

Elasticsearch :9200
- Product Full-Text Search
- Faceted Navigation
- Auto-complete
- Search Analytics

OBSERVABILITY STACK

Zipkin :9411	ELK Stack	Prometheus + Grafana
• Distributed Tracing	• Centralized Logging	• Metrics
• Performance Analysis	• Search	• Dashboards
	• Visualization	• Alerts

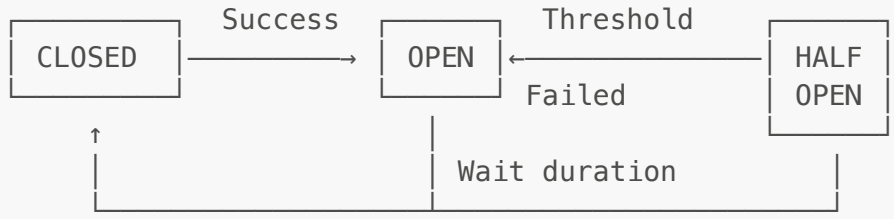
EXTERNAL INTEGRATIONS

Payment Gateway	Email Service	SMS Service
• Stripe	• AWS SES	• Twilio
• PayPal	• SendGrid	• AWS SNS

3.2 Service Interaction Patterns

```
┌─────────────────────────────────────────────────────────────────┐
│  PATTERN 1: SYNCHRONOUS (Request/Response via Feign + Eureka)   │
└─────────────────────────────────────────────────────────────────┘

Order Service needs user info:
    OrderService
        ↓ (Feign Client)
    Eureka Discovery (finds User Service instances)
        ↓ (Load balanced)
    User Service → Returns user data
        ↓
    Order Service continues processing


┌─────────────────────────────────────────────────────────────────┐
│  PATTERN 2: ASYNCHRONOUS (Event-Driven via Kafka)               │
└─────────────────────────────────────────────────────────────────┘

Order Saga Flow (Happy Path):

1. Order Service
   ↓ Publish: OrderCreatedEvent
   Kafka Topic: order-created-events

2. Inventory Service (Listener)
   ↓ Consumes: OrderCreatedEvent
   ↓ Reserve stock
   ↓ Publish: InventoryReservedEvent

3. Payment Service (Listener)
   ↓ Consumes: InventoryReservedEvent
   ↓ Process payment (Stripe API)
   ↓ Publish: PaymentCompletedEvent

4. Order Service (Listener)
   ↓ Consumes: PaymentCompletedEvent
   ↓ Update order status: PAID
   ↓ Publish: OrderPaidEvent

5. Notification Service (Listener)
   ↓ Consumes: OrderPaidEvent
   ↓ Send email + SMS confirmation


┌─────────────────────────────────────────────────────────────────┐
│  PATTERN 3: SAGA PATTERN (Compensating Transactions)            │
└─────────────────────────────────────────────────────────────────┘
```

```
Order Saga Flow (Payment Failure):

1. Order Created → Inventory Reserved
2. Payment Processing... FAILED ❌
3. Compensation Actions:
   ↓ Publish: PaymentFailedEvent
   ↓ Inventory Service: Release reserved stock
   ↓ Order Service: Update status → CANCELLED
   ↓ Notification Service: Send failure email


┌──────────────────────────────────────────────────────────────────┐
│  PATTERN 4: CIRCUIT BREAKER (Resilience)                         │
└──────────────────────────────────────────────────────────────────┘

Review Service calls User Service via Feign:

@CircuitBreaker(name = "userService", fallbackMethod = "getUserFallback")
public UserDto getUser(Long userId) {
    return userServiceClient.getUserById(userId);
}

States:
```



3.3 New Services Detailed

Service: Order Service (Port 8083)

Purpose: Manage customer orders and order lifecycle

Endpoints:

- `POST /api/orders` - Create new order
- `GET /api/orders/{id}` - Get order details
- `GET /api/orders/user/{userId}` - Get user's orders
- `PUT /api/orders/{id}/cancel` - Cancel order

Database: PostgreSQL (orders, order_items, order_events)

Events Published:

- `OrderCreatedEvent`
- `OrderPaidEvent`
- `OrderShippedEvent`
- `OrderCancelledEvent`

Events Consumed:

- `PaymentCompletedEvent`
- `PaymentFailedEvent`
- `InventoryReservedEvent`
- `InventoryReservationFailedEvent`

Dependencies:

- Product Service (get product details via Feign)
- User Service (validate user via Feign)
- Inventory Service (check stock via events)
- Payment Service (process payment via events)

Service: Payment Service (Port 8084)

Purpose: Handle payment processing and transaction management

Endpoints:

- `POST /api/payments` - Process payment
- `GET /api/payments/{id}` - Get payment status
- `POST /api/payments/{id}/refund` - Refund payment

Database: PostgreSQL (payments, transactions)

External Integrations:

- Stripe API
- PayPal SDK

Events Published:

- `PaymentCompletedEvent`
- `PaymentFailedEvent`
- `RefundCompletedEvent`

Events Consumed:

- `InventoryReservedEvent`

Service: Inventory Service (Port 8085)

Purpose: Manage product stock and reservations

Endpoints:

- `GET /api/inventory/{productId}` - Get stock level
- `POST /api/inventory/reserve` - Reserve stock
- `POST /api/inventory/release` - Release reservation

- `PUT /api/inventory/{productId}/stock` - Update stock

Database: MySQL (inventory, reservations)

Events Published:

- `InventoryReservedEvent`
- `InventoryReservationFailedEvent`
- `InventoryUpdatedEvent`

Events Consumed:

- `OrderCreatedEvent`
- `OrderCancelledEvent`
- `PaymentFailedEvent`

Service: Notification Service (Port 8086)

Purpose: Send email, SMS, and push notifications

Endpoints:

- `POST /api/notifications/email` - Send email
- `POST /api/notifications/sms` - Send SMS
- `GET /api/notifications/{userId}` - Get notification history

Database: MongoDB (notification_logs)

External Integrations:

- AWS SES / SendGrid (Email)
- Twilio / AWS SNS (SMS)
- Firebase Cloud Messaging (Push)

Events Consumed:

- `ReviewAddedEvent`
- `OrderCreatedEvent`
- `OrderPaidEvent`
- `OrderShippedEvent`
- `PaymentCompletedEvent`
- `PaymentFailedEvent`

Templates:

- Review confirmation
- Order confirmation
- Payment receipt
- Shipping notification
- Order cancellation

---

4. Detailed Component Analysis

4.1 Changes to Existing Services

User Service (e-com-user-auth)

Current State:

```
// Basic user model
@Entity
public class User {
    private Long id;
    private String username;
    private String email;
    private String password;
    private Set<String> roles;
}
```

Proposed Changes:

1. Add Contact Information:

```
@Entity
public class User {
    private Long id;
    private String username;
    private String email;
    private String phone;          // NEW: For SMS notifications
    private String firstName;       // NEW: For personalization
    private String lastName;        // NEW: For personalization
    private String password;
    private Set<String> roles;

    // NEW: Preferences
    @Embedded
    private NotificationPreferences preferences;

    // NEW: Timestamps
    private LocalDateTime createdAt;
    private LocalDateTime lastLogin;
}

@Embeddable
public class NotificationPreferences {
    private boolean emailEnabled = true;
    private boolean smsEnabled = false;
    private boolean pushEnabled = false;
}
```

2. Add Profile Management Endpoints:

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    // NEW ENDPOINTS
    @GetMapping("/profile")
    public UserDto getCurrentUserProfile();

    @PutMapping("/profile")
    public UserDto updateProfile(@RequestBody UpdateProfileRequest
request);

    @PutMapping("/preferences")
    public void updateNotificationPreferences(@RequestBody
NotificationPreferences prefs);
}
```

3. Enhanced Security:

```
// Add refresh token support
@Entity
public class RefreshToken {
    private String token;
    private Long userId;
    private LocalDateTime expiryDate;
}
```

Build.gradle additions:

```
// Add notification dependencies
implementation 'org.springframework.boot:spring-boot-starter-mail'
// Optional: implementation 'com.twilio.sdk:twilio:9.14.1'
```

Product Service (e-com-product-catalog)

Current State:

- Basic CRUD for products
- Redis caching
- MongoDB for reviews

Proposed Changes:

1. Add Inventory Integration:

```
@Service
public class ProductService {
    @Autowired
    private InventoryServiceClient inventoryClient;

    public ProductWithStock getProductWithStock(Long id) {
        Product product = getProductById(id);
        Integer stock = inventoryClient.getStock(id);
        return new ProductWithStock(product, stock);
    }
}
```

2. Add Product Rating Cache:

```
// Compute average rating from reviews
public ProductRating calculateRating(Long productId) {
    List<Review> reviews = reviewRepository.findByProductId(productId);
    double avg = reviews.stream()
        .mapToInt(Review::getRating)
        .average()
        .orElse(0.0);
    return new ProductRating(productId, avg, reviews.size());
}
```

3. Event Publishing for Stock Changes:

```
@KafkaListener(topics = "inventory-updated")
public void handleInventoryUpdate(InventoryUpdatedEvent event) {
    // Invalidate product cache
    cacheManager.evict("products", event.getProductId());
}
```

Build.gradle additions:

```
// Add Elasticsearch for search
implementation 'org.springframework.boot:spring-boot-starter-data-
elasticsearch'
```

4.2 Infrastructure Components

API Gateway Configuration

application.yml:

```yaml
spring:
  cloud:
    gateway:
      routes:
        # User Service Routes
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/api/auth/**, /api/users/**
          filters:
            - name: RateLimiter
              args:
                redis-rate-limiter.replenishRate: 100
                redis-rate-limiter.burstCapacity: 200
            - name: CircuitBreaker
              args:
                name: userServiceBreaker
                fallbackUri: forward:/fallback/user

        # Product Service Routes
        - id: product-service
          uri: lb://product-service
          predicates:
            - Path=/api/products/**, /api/reviews/**
          filters:
            - ResponseCache
            - name: Retry
              args:
                retries: 3
                statuses: BAD_GATEWAY, SERVICE_UNAVAILABLE

        # Order Service Routes
        - id: order-service
          uri: lb://order-service
          predicates:
            - Path=/api/orders/**
          filters:
            - name: RequestRateLimiter
              args:
                deny-empty-key: false
```

Eureka Server Configuration

Main Class:

```java
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {
    public static void main(String[] args) {
```

```
        SpringApplication.run(ServiceRegistryApplication.class, args);
    }
}
```

application.yml:

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
  server:
    enable-self-preservation: true
```

Observability Stack

Zipkin Configuration (Each Service):

```
spring:
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      probability: 1.0  # 100% sampling for dev
```

Prometheus Configuration:

```
# prometheus.yml
scrape_configs:
  - job_name: 'spring-actuator'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['localhost:8081', 'localhost:8082', 'localhost:8083']
```

Each Service application.yml:

```
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,prometheus
  metrics:
```

```
    export:
      prometheus:
        enabled: true
```

---

5. Implementation Phases

Phase 1: Foundation (Weeks 1-2) 🟢 START HERE

Objective: Add infrastructure services and improve reliability

Tasks:

Week 1:

1. ✅ Create Eureka Server project
   - Dependencies: `spring-cloud-starter-netflix-eureka-server`
   - Configure port 8761
   - Test service registration
2. ✅ Update existing services to register with Eureka
   - Add dependency: `spring-cloud-starter-netflix-eureka-client`
   - Configure `spring.application.name`
   - Configure `eureka.client.service-url`
3. ✅ Update Feign clients to use service discovery
   - Remove hardcoded URLs
   - Use `@FeignClient(name = "user-service")`

Week 2: 4. ✅ Add Circuit Breakers with Resilience4j

- Dependency: `resilience4j-spring-boot3`
- Configure fallback methods
- Add retry mechanism

5. ✅ Create API Gateway
   - Project: `spring-cloud-starter-gateway`
   - Configure routes for existing services
   - Add rate limiting
6. ✅ Testing
   - Test service discovery
   - Test circuit breaker behavior
   - Load test gateway

Deliverables:

- Working Eureka Server
- Services auto-discovering each other
- API Gateway routing requests
- Circuit breakers protecting services

Estimated Effort: 40-50 hours

Phase 2: Notification Service (Weeks 3-4)

Objective: Complete the TODO - send email/SMS notifications

Tasks:

Week 3:

1. ✅ Update User model
    - Add `phone`, `firstName`, `lastName`
    - Add `NotificationPreferences`
    - Database migration script
2. ✅ Create Notification Service
    - New Spring Boot project
    - Register with Eureka
    - Configure email (AWS SES or SMTP)
    - Configure SMS (Twilio)
3. ✅ Implement NotificationService class
    - Email templates (Thymeleaf)
    - SMS message builder
    - Async processing with `@Async`

Week 4: 4. ✅ Update ReviewEventConsumer

- Fetch user details
- Call NotificationService
- Handle errors gracefully

5. ✅ Create notification templates
    - HTML email templates
    - SMS message templates
    - Support for multiple languages
6. ✅ Testing
    - Unit tests for NotificationService
    - Integration tests with Kafka
    - Real email/SMS testing

Deliverables:

- Notification Service deployed
- Email notifications working
- SMS notifications working (optional)
- Template management

Estimated Effort: 35-45 hours

Phase 3: Order Management (Weeks 5-7)

Objective: Enable complete purchase flow

Tasks:

Week 5:

1. ✅ Create Order Service skeleton
    - Spring Boot project
    - PostgreSQL database setup
    - Entity models (Order, OrderItem)
    - Basic CRUD operations
2. ✅ Implement order creation
    - Validate products (call Product Service)
    - Validate user (call User Service)
    - Calculate totals
    - Save to database

Week 6: 3. ✅ Create Payment Service skeleton

- Spring Boot project
- PostgreSQL database setup
- Stripe SDK integration
- Payment entity model

4. ✅ Implement payment processing
    - Create Stripe payment intent
    - Handle webhooks
    - Publish PaymentCompletedEvent

Week 7: 5. ✅ Create Inventory Service

- Spring Boot project
- MySQL database setup
- Stock reservation logic
- Event listeners

6. ✅ Implement Saga pattern

    - Order orchestration
    - Compensating transactions
    - Event choreography

7. ✅ Integration testing
    - End-to-end order flow testing
    - Failure scenario testing
    - Performance testing

Deliverables:

- Order Service operational
- Payment integration working
- Inventory management functional
- Saga pattern implemented

Estimated Effort: 70-85 hours

Phase 4: Observability (Weeks 8-9)

Objective: Add monitoring, logging, and tracing

Tasks:

Week 8:

1. ✅ Setup Zipkin
    - Docker container
    - Configure all services
    - Test trace visualization
2. ✅ Setup ELK Stack
    - Elasticsearch, Logstash, Kibana
    - Configure log shipping
    - Create dashboards
3. ✅ Add structured logging
    - JSON log format
    - Correlation IDs
    - MDC context

Week 9: 4. ✅ Setup Prometheus + Grafana

- Prometheus for metrics collection
- Grafana dashboards
- Alert rules

5. ✅ Custom metrics

    - Business metrics (orders/hour, revenue)
    - Technical metrics (response time, error rate)
    - JVM metrics

6. ✅ Documentation

    - Runbooks for common issues
    - Dashboard user guide
    - Alert response procedures

Deliverables:

- Distributed tracing operational
- Centralized logging with search
- Metrics and dashboards
- Alert system configured

Estimated Effort: 45-55 hours

Phase 5: Search & Performance (Weeks 10-11)

Objective: Add advanced search and optimize performance

Tasks:

Week 10:

1. ✅ Setup Elasticsearch

    - Docker container
    - Create indices
    - Configure mappings

2. ✅ Sync products to Elasticsearch

    - Initial bulk import
    - Real-time updates via Kafka
    - Handle failures

3. ✅ Implement search API

    - Full-text search
    - Faceted search
    - Auto-complete

Week 11: 4. ✅ Performance optimization

- Add Caffeine cache (L1)
- Optimize database queries
- Add database indexes

5. ✅ Load testing

    - JMeter test scripts
    - Gatling scenarios
    - Performance benchmarks

6. ✅ Optimization based on results
    - Adjust cache TTLs
    - Optimize slow queries
    - Scale services

Deliverables:

- Elasticsearch search working
- Multi-level caching
- Performance benchmarks documented
- Optimized configuration

Estimated Effort: 40-50 hours

Phase 6: Advanced Features (Weeks 12-14)

Objective: Add real-time features and ML

Tasks:

Week 12:

1. ✅ Add WebSocket support
    - Configure STOMP
    - Order tracking updates
    - Stock alerts
2. ✅ Implement real-time features
    - Live order status
    - Admin dashboard updates
    - Chat support (optional)

Week 13: 3. ✅ ML Recommendation Service

- Python Flask service
- Collaborative filtering model
- Integration with Product Service

4. ✅ Data pipeline for ML
    - Export user behavior data
    - Train recommendation model
    - Deploy model

Week 14: 5. ✅ Event Sourcing for Orders

- Event store implementation
- Rebuild state from events
- Snapshot mechanism

6. ✅ CQRS pattern
    - Separate read/write models
    - Denormalized views
    - Event handlers

Deliverables:

- WebSocket real-time updates
- ML recommendation engine
- Event Sourcing implemented
- CQRS pattern applied

Estimated Effort: 75-90 hours

Phase 7: Cloud & DevOps (Weeks 15-16)

Objective: Prepare for production deployment

Tasks:

Week 15:

1. ✅ Kubernetes manifests
    - Deployments for all services
    - Services and Ingress
    - ConfigMaps and Secrets
2. ✅ Helm charts
    - Chart for each service
    - Values for dev/staging/prod
    - Deployment scripts
3. ✅ CI/CD pipeline
    - GitHub Actions workflows
    - Docker image building
    - Automated testing

Week 16: 4. ✅ Production checklist

- HTTPS configuration
- Secret management (Vault)
- Backup strategy
- Disaster recovery plan

5. ✅ Performance testing
    - Production-like environment
    - Load testing
    - Chaos engineering
6. ✅ Documentation
    - Architecture documentation
    - Deployment guide
    - Operations manual

Deliverables:

- Kubernetes deployment ready
- CI/CD pipeline operational
- Production checklist completed
- Complete documentation

Estimated Effort: 50-60 hours

---

6. Technical Specifications

6.1 Database Schema Changes

User Service - users table

Current Schema:

```
CREATE TABLE users (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE user_roles (
    user_id BIGINT,
    role VARCHAR(50),
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Proposed Schema:

```
CREATE TABLE users (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(255) UNIQUE NOT NULL,
```

```
    email VARCHAR(255) NOT NULL,
    phone VARCHAR(20),                          -- NEW
    first_name VARCHAR(100),                    -- NEW
    last_name VARCHAR(100),                     -- NEW
    password VARCHAR(255) NOT NULL,
    email_notifications BOOLEAN DEFAULT TRUE,   -- NEW
    sms_notifications BOOLEAN DEFAULT FALSE,    -- NEW
    push_notifications BOOLEAN DEFAULT FALSE,   -- NEW
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP,                       -- NEW
    INDEX idx_email (email),
    INDEX idx_username (username)
);

CREATE TABLE user_roles (
    user_id BIGINT,
    role VARCHAR(50),
    FOREIGN KEY (user_id) REFERENCES users(id),
    INDEX idx_user_id (user_id)
);

CREATE TABLE refresh_tokens (                   -- NEW
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    token VARCHAR(255) UNIQUE NOT NULL,
    user_id BIGINT NOT NULL,
    expiry_date TIMESTAMP NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id),
    INDEX idx_token (token),
    INDEX idx_user_id (user_id)
);
```

Migration Script:

```
-- Add new columns to existing users table
ALTER TABLE users
    ADD COLUMN phone VARCHAR(20),
    ADD COLUMN first_name VARCHAR(100),
    ADD COLUMN last_name VARCHAR(100),
    ADD COLUMN email_notifications BOOLEAN DEFAULT TRUE,
    ADD COLUMN sms_notifications BOOLEAN DEFAULT FALSE,
    ADD COLUMN push_notifications BOOLEAN DEFAULT FALSE,
    ADD COLUMN last_login TIMESTAMP;

-- Add indexes
CREATE INDEX idx_email ON users(email);
CREATE INDEX idx_username ON users(username);

-- Create new table
CREATE TABLE refresh_tokens (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    token VARCHAR(255) UNIQUE NOT NULL,
```

```
    user_id BIGINT NOT NULL,
    expiry_date TIMESTAMP NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id),
    INDEX idx_token (token),
    INDEX idx_user_id (user_id)
);
```

Order Service - PostgreSQL

```
CREATE TABLE orders (
    id BIGSERIAL PRIMARY KEY,
    order_number VARCHAR(50) UNIQUE NOT NULL,
    user_id BIGINT NOT NULL,
    total_amount DECIMAL(10,2) NOT NULL,
    status VARCHAR(50) NOT NULL,  -- PENDING, PAID, SHIPPED, DELIVERED,
CANCELLED
    payment_id VARCHAR(255),
    shipping_address TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    INDEX idx_user_id (user_id),
    INDEX idx_status (status),
    INDEX idx_created_at (created_at)
);

CREATE TABLE order_items (
    id BIGSERIAL PRIMARY KEY,
    order_id BIGINT NOT NULL,
    product_id BIGINT NOT NULL,
    quantity INTEGER NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE,
    INDEX idx_order_id (order_id),
    INDEX idx_product_id (product_id)
);

CREATE TABLE order_events (
    id BIGSERIAL PRIMARY KEY,
    order_id BIGINT NOT NULL,
    event_type VARCHAR(100) NOT NULL,
    event_data JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (order_id) REFERENCES orders(id),
    INDEX idx_order_id (order_id),
    INDEX idx_event_type (event_type),
    INDEX idx_created_at (created_at)
);
```

Payment Service - PostgreSQL

```
CREATE TABLE payments (
    id BIGSERIAL PRIMARY KEY,
    payment_id VARCHAR(255) UNIQUE NOT NULL,
    order_id BIGINT NOT NULL,
    user_id BIGINT NOT NULL,
    amount DECIMAL(10,2) NOT NULL,
    currency VARCHAR(3) DEFAULT 'USD',
    status VARCHAR(50) NOT NULL,  -- PENDING, COMPLETED, FAILED, REFUNDED
    payment_method VARCHAR(50),   -- STRIPE, PAYPAL
    stripe_payment_intent_id VARCHAR(255),
    error_message TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    INDEX idx_order_id (order_id),
    INDEX idx_user_id (user_id),
    INDEX idx_status (status),
    INDEX idx_stripe_payment_intent_id (stripe_payment_intent_id)
);

CREATE TABLE transactions (
    id BIGSERIAL PRIMARY KEY,
    payment_id BIGINT NOT NULL,
    transaction_type VARCHAR(50) NOT NULL,  -- CHARGE, REFUND
    amount DECIMAL(10,2) NOT NULL,
    transaction_id VARCHAR(255),
    status VARCHAR(50) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (payment_id) REFERENCES payments(id),
    INDEX idx_payment_id (payment_id),
    INDEX idx_transaction_type (transaction_type)
);
```

Inventory Service - MySQL

```
CREATE TABLE inventory (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT UNIQUE NOT NULL,
    available_stock INTEGER NOT NULL DEFAULT 0,
    reserved_stock INTEGER NOT NULL DEFAULT 0,
    total_stock INTEGER GENERATED ALWAYS AS (available_stock +
reserved_stock) STORED,
    reorder_level INTEGER DEFAULT 10,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    INDEX idx_product_id (product_id),
    INDEX idx_available_stock (available_stock)
);
```

```sql
CREATE TABLE reservations (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    reservation_id VARCHAR(255) UNIQUE NOT NULL,
    product_id BIGINT NOT NULL,
    order_id BIGINT NOT NULL,
    quantity INTEGER NOT NULL,
    status VARCHAR(50) NOT NULL,  -- RESERVED, CONFIRMED, RELEASED
    expires_at TIMESTAMP NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES inventory(product_id),
    INDEX idx_order_id (order_id),
    INDEX idx_status (status),
    INDEX idx_expires_at (expires_at)
);
```

6.2 Kafka Topics Configuration

```yaml
# Kafka Topics definition
topics:
  - name: review-added-events
    partitions: 3
    replication-factor: 2

  - name: order-created-events
    partitions: 6
    replication-factor: 2

  - name: inventory-reserved-events
    partitions: 6
    replication-factor: 2

  - name: payment-completed-events
    partitions: 6
    replication-factor: 2

  - name: payment-failed-events
    partitions: 3
    replication-factor: 2

  - name: order-paid-events
    partitions: 6
    replication-factor: 2

  - name: order-shipped-events
    partitions: 3
    replication-factor: 2

  - name: notification-events
    partitions: 6
```

```
    replication-factor: 2

  - name: inventory-updated-events
    partitions: 3
    replication-factor: 2
```

6.3 API Specifications

Order Service API

```
openapi: 3.0.0
info:
  title: Order Service API
  version: 1.0.0

paths:
  /api/orders:
    post:
      summary: Create new order
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                items:
                  type: array
                  items:
                    type: object
                    properties:
                      productId:
                        type: integer
                      quantity:
                        type: integer
                shippingAddress:
                  type: string
      responses:
        '201':
          description: Order created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Order'
        '400':
          description: Invalid request
        '401':
          description: Unauthorized
```

```yaml
  /api/orders/{orderId}:
    get:
      summary: Get order details
      security:
        - bearerAuth: []
      parameters:
        - name: orderId
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: Order details
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Order'

components:
  schemas:
    Order:
      type: object
      properties:
        id:
          type: integer
        orderNumber:
          type: string
        userId:
          type: integer
        totalAmount:
          type: number
        status:
          type: string
          enum: [PENDING, PAID, SHIPPED, DELIVERED, CANCELLED]
        items:
          type: array
          items:
            $ref: '#/components/schemas/OrderItem'
        createdAt:
          type: string
          format: date-time

    OrderItem:
      type: object
      properties:
        productId:
          type: integer
        productName:
          type: string
        quantity:
          type: integer
```

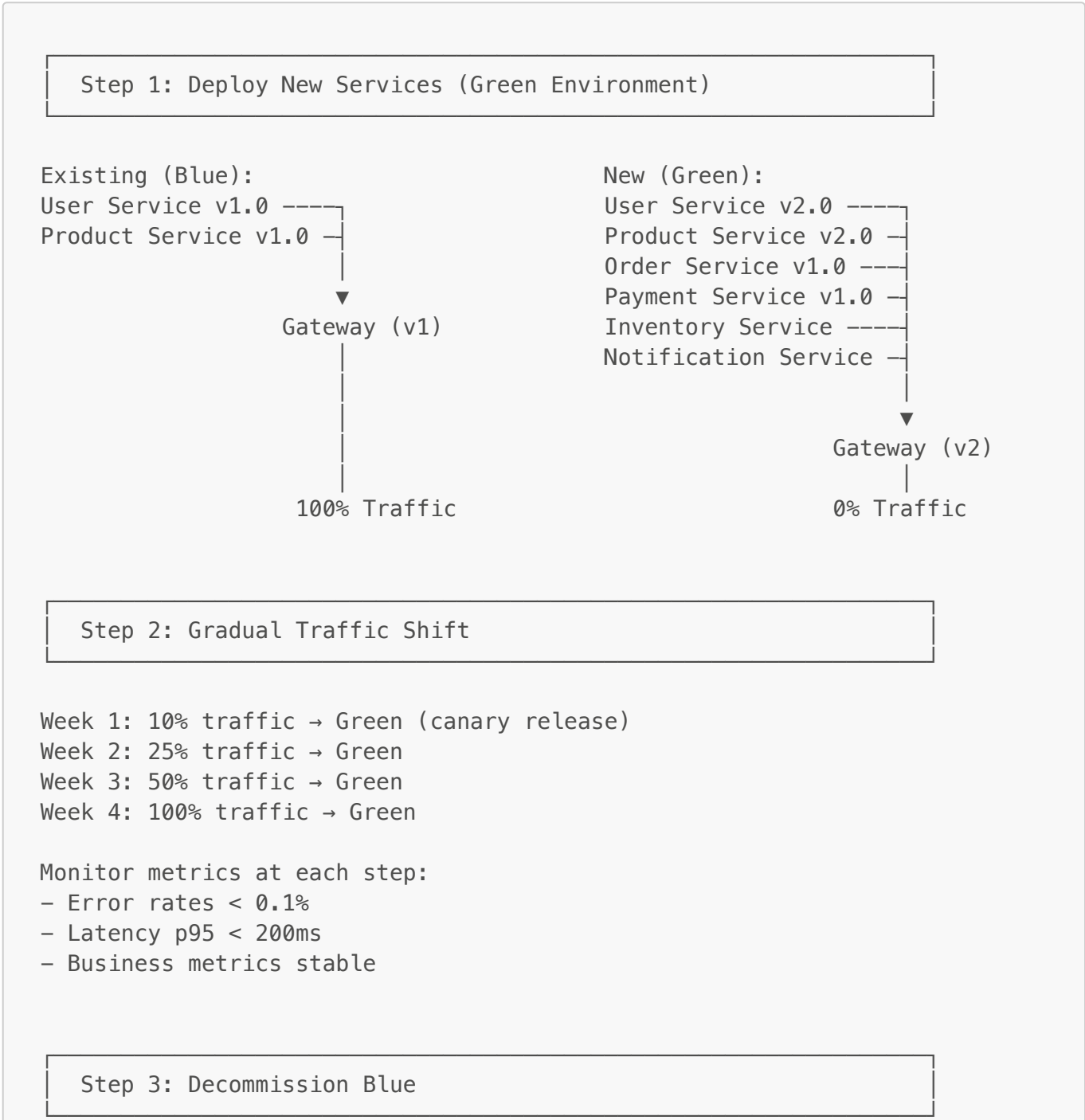
```
        price:
          type: number
```

---

7. Migration Strategy

7.1 Zero-Downtime Deployment Approach

Strategy: Blue-Green Deployment



7.2 Database Migration Strategy

Flyway Migration Scripts

```
// V1__baseline.sql (Document current state)
CREATE TABLE users (...);
CREATE TABLE products (...);

// V2__add_user_contact_info.sql
ALTER TABLE users
    ADD COLUMN phone VARCHAR(20),
    ADD COLUMN first_name VARCHAR(100),
    ADD COLUMN last_name VARCHAR(100);

// V3__add_notification_preferences.sql
ALTER TABLE users
    ADD COLUMN email_notifications BOOLEAN DEFAULT TRUE,
    ADD COLUMN sms_notifications BOOLEAN DEFAULT FALSE,
    ADD COLUMN push_notifications BOOLEAN DEFAULT FALSE;

// V4__create_orders_schema.sql
CREATE TABLE orders (...);
CREATE TABLE order_items (...);
CREATE TABLE order_events (...);
```

Configuration:

```
spring:
  flyway:
    enabled: true
    baseline-on-migrate: true
    locations: classpath:db/migration
```

7.3 Rollback Plan

Scenario: Critical Issue Discovered in Production

```
┌──────────────────────────────────────────────────────────┐
│  ROLLBACK PROCEDURE (15 minutes)                         │
└──────────────────────────────────────────────────────────┘

T+0:00 - Issue detected (alerts firing)
         └─ Incident commander declares rollback

T+0:02 - Switch traffic back to Blue environment
```

```
          └─ kubectl apply -f blue-deployment.yaml
          └─ kubectl scale deployment green --replicas=0

T+0:05 - Verify Blue environment stable
          └─ Check error rates
          └─ Check latency metrics
          └─ Verify business KPIs

T+0:10 - Database rollback (if needed)
          └─ Run rollback scripts
          └─ Restore from backup (if necessary)

T+0:15 - Post-rollback verification
          └─ All systems green
          └─ Customer impact assessment
          └─ RCA (Root Cause Analysis) initiated
```

Database Rollback Strategy:

```sql
-- Each migration has a corresponding rollback
-- V2__add_user_contact_info.sql
ALTER TABLE users ADD COLUMN phone VARCHAR(20);

-- U2__rollback_user_contact_info.sql
ALTER TABLE users DROP COLUMN phone;
```

---

8. Risk Assessment

8.1 Technical Risks

Risk	Probability	Impact	Mitigation Strategy
Service Discovery Failure	Low	High	• Fallback to hardcoded URLs • Health checks • Multiple Eureka instances
Kafka Message Loss	Medium	High	• Message persistence • Acknowledgment handling • Dead letter queue
Database Migration Issues	Medium	Medium	• Test migrations in staging • Backup before migration • Rollback scripts ready
Performance Degradation	Medium	High	• Load testing before release • Gradual rollout • Auto-scaling configured

Risk	Probability	Impact	Mitigation Strategy
Circuit Breaker False Positives	Low	Medium	• Tune thresholds • Monitor metrics • Gradual threshold adjustments
Distributed Transaction Failures	Medium	High	• Implement saga pattern • Idempotent operations • Compensating transactions
Cache Inconsistency	Medium	Low	• Short TTLs • Event-driven invalidation • Cache warming strategy

8.2 Operational Risks

Risk	Probability	Impact	Mitigation Strategy
Insufficient Monitoring	Low	High	• Comprehensive observability • Alert fatigue prevention • Runbooks for common issues
Deployment Errors	Medium	Medium	• Automated CI/CD • Deployment checklists • Automated rollback triggers
Team Knowledge Gap	Medium	Medium	• Documentation • Training sessions • Pair programming
Third-Party Service Outage	Low	High	• Circuit breakers • Fallback mechanisms • SLA monitoring
Cost Overruns	Medium	Medium	• Cost monitoring • Resource limits • Auto-scaling bounds

8.3 Business Risks

Risk	Probability	Impact	Mitigation Strategy
Feature Delays	Medium	Medium	• Phased approach • MVP first • Regular stakeholder updates
Customer Impact During Migration	Low	High	• Blue-green deployment • Feature flags • Gradual rollout

Risk	Probability	Impact	Mitigation Strategy
Incomplete Testing	Medium	High	• Comprehensive test plan • Staging environment • Beta testing program

---

9. Timeline & Resource Estimation

9.1 High-Level Timeline

```
Month 1: Foundation
├─ Week 1–2: Service Discovery & Gateway (Phase 1)
└─ Week 3–4: Notification Service (Phase 2)

Month 2: Core Features
Month 3: Core Features (continued)
├─ Week 5–7: Order Management (Phase 3)
├─ Week 8–9: Observability (Phase 4)
└─ Week 10–11: Search & Performance (Phase 5)

Month 4: Advanced Features & Deployment
├─ Week 12–14: Real-time & ML (Phase 6)
└─ Week 15–16: Cloud Deployment (Phase 7)
```

9.2 Effort Breakdown

Phase	Tasks	Hours	Resources
Phase 1	Infrastructure	40-50	1 Senior Dev
Phase 2	Notifications	35-45	1 Developer
Phase 3	Order Management	70-85	2 Developers
Phase 4	Observability	45-55	1 DevOps + 1 Dev
Phase 5	Search & Performance	40-50	1 Senior Dev
Phase 6	Advanced Features	75-90	2 Senior Devs
Phase 7	Cloud Deployment	50-60	1 DevOps + 1 Dev
Total	All Phases	**355-435 hours**	**~3 months**

9.3 Resource Requirements

Team Composition:

- **1 Senior Backend Developer** (Lead)
- **2 Backend Developers** (Implementation)

- **1 DevOps Engineer** (Infrastructure & Deployment)
- **1 QA Engineer** (Testing)
- **1 Product Owner** (Requirements & Prioritization)

Infrastructure Costs (Monthly - Development):

- AWS/GCP compute instances: $200-300
- Database services: $150-250
- Kafka cluster: $100-150
- Elasticsearch: $100-150
- Monitoring tools: $50-100
- **Total:** ~$600-950/month

Infrastructure Costs (Monthly - Production):

- Auto-scaling instances: $800-1200
- Database services (replicated): $500-800
- Kafka cluster (HA): $300-500
- Elasticsearch cluster: $400-600
- CDN & Load Balancer: $100-200
- Monitoring & Logging: $200-300
- **Total:** ~$2,300-3,600/month

9.4 Success Metrics

Technical KPIs:

- ✅ API response time p95 < 200ms
- ✅ Error rate < 0.1%
- ✅ Service availability > 99.9%
- ✅ MTTR (Mean Time To Recovery) < 15min
- ✅ Deployment frequency: Daily

Business KPIs:

- ✅ Order completion rate > 95%
- ✅ Payment success rate > 98%
- ✅ Email delivery rate > 99%
- ✅ Customer satisfaction score > 4.5/5

Operational KPIs:

- ✅ Deployment success rate > 95%
- ✅ Rollback frequency < 5%
- ✅ Alert fatigue ratio < 10%
- ✅ Documentation coverage > 90%

---

10. Appendices

Appendix A: Technology Decision Matrix

Requirement	Option 1	Option 2	Selected	Reason
Service Discovery	Consul	Eureka	**Eureka**	Better Spring integration
API Gateway	Zuul	Spring Cloud Gateway	**Spring Cloud Gateway**	Non-blocking, modern
Circuit Breaker	Hystrix	Resilience4j	**Resilience4j**	Active development, Java 8+
Tracing	Jaeger	Zipkin	**Zipkin**	Simpler setup, good UI
Logging	Splunk	ELK	**ELK**	Open source, flexible
Metrics	Datadog	Prometheus	**Prometheus**	Open source, industry standard
Search	Solr	Elasticsearch	**Elasticsearch**	Better ecosystem, easier
Payment	Stripe	PayPal	**Stripe**	Better API, documentation
Email	SendGrid	AWS SES	**AWS SES**	Cost-effective, reliable
SMS	Twilio	AWS SNS	**Twilio**	Better developer experience

Appendix B: Glossary

- **Saga Pattern**: Distributed transaction pattern using compensating transactions
- **CQRS**: Command Query Responsibility Segregation
- **Circuit Breaker**: Pattern to prevent cascade failures
- **Blue-Green Deployment**: Zero-downtime deployment strategy
- **Event Sourcing**: Storing state changes as events
- **Service Mesh**: Infrastructure layer for service-to-service communication
- **Idempotent**: Operation that produces same result regardless of repetition

---

Appendix C: Quick Start Commands

```
# Start existing services
cd e-com-user-auth && ./gradlew bootRun
cd e-com-product-catalog && ./gradlew bootRun

# Start infrastructure
docker-compose up -d

# Access services
# User Auth: http://localhost:8081
# Product Catalog: http://localhost:8080
```

```
# Swagger: http://localhost:8080/swagger-ui.html

# After Phase 1 implementation
# Eureka Dashboard: http://localhost:8761
# API Gateway: http://localhost:8080 (all services)

# After Phase 4 implementation
# Zipkin: http://localhost:9411
# Kibana: http://localhost:5601
# Grafana: http://localhost:3000
```

🎯 Next Steps

1. **Review this document** with your team
2. **Prioritize phases** based on business needs
3. **Setup development environment** for Phase 1
4. **Create GitHub project board** for tracking
5. **Schedule kickoff meeting** to align on approach
6. **Begin Phase 1 implementation**

Document Status: ✅ Ready for Review
Next Review Date: Upon Phase 1 completion
Questions/Feedback: Create GitHub issue or contact architecture team

---

End of Document