Lab 4: Microservices Decomposition & Communication

Bus Ticket Management System
System Name: Bus Ticket Management System

Introduction

This lab focuses on transitioning from the monolithic layered architecture (Lab 3) to a microservices architecture. The goal is to logically decompose the Bus Ticket Management System into independent, loosely coupled services based on business capabilities, define clear service contracts, and establish communication strategies between services.

The microservices architecture will enable:

- **Independent deployment** - Each service can be deployed independently
- **Technology diversity** - Different services can use different technologies
- **Scalability** - Services can be scaled independently based on demand
- **Fault isolation** - Failure in one service doesn't cascade to others
- **Team autonomy** - Different teams can own different services

Activity Practice 1: Decomposition by Business Capability

1.1 Core Business Capabilities Identified

Based on the functional requirements identified in Lab 1 and analysis of the existing codebase, the following core business capabilities have been identified for the system:

1. **User Management** - Managing customer accounts, authentication, and profiles
2. **Catalog Management** - Managing buses, companies, routes, and stations information
3. **Booking & Ticketing** - Handling ticket reservations and seat selections
4. **Inventory Management** - Managing seat availability and schedule inventory
5. **Payment Processing** - Coordinating payment transactions
6. **Notification Services** - Sending emails and SMS confirmations
7. **Search & Discovery** - Semantic search capabilities using embeddings
8. **Analytics & Reporting** - Revenue reports and trip statistics

1.2 Proposed Microservices

The following table maps each business capability to a dedicated microservice with clearly defined data ownership:

Business Capability	Proposed Microservice	Data Owned (Entities)	Primary Responsibilities

User Management	**User Service**	- User Profile - Authentication Credentials - User Roles	- User registration - Authentication (JWT) - Profile management - Password management
Catalog Management	**Catalog Service**	- Bus Companies - Vehicles (Cars) - Routes - Stations - Bus Reviews	- CRUD operations for catalog data - Vehicle specifications - Route definitions - Company information - Review management
Booking & Ticketing	**Booking Service**	- Tickets - Booking History - Passenger Information	- Ticket creation - Booking confirmation - Cancellation processing - Ticket lookup - Invoice generation
Inventory Management	**Inventory Service**	- Seats - Seat Configurations - Schedules - Availability Status	- Real-time seat availability - Schedule management - Seat reservation/release - Inventory locking
Payment Processing	**Payment Service**	- Payment Transactions - Payment Methods - Payment Status	- Payment gateway integration - Transaction processing - Payment status tracking - QR code generation
Notification	**Notification Service**	- Email Templates - Notification Logs - Delivery Status	- Send booking confirmations - Send payment receipts - Password reset emails - Trip reminders
Search & Discovery	**Search Service**	- Embedding Vectors - Search Indexes - Query Cache	- Semantic search using embeddings - Trip search - Route recommendations - Chatbot query processing

1.3 Service Boundaries & Coupling Analysis

Key Principles Applied:

- **Data Encapsulation**: Each service owns its database tables exclusively
- **Loose Coupling**: Services interact only through well-defined APIs
- **High Cohesion**: Related functionalities are grouped within the same service
- **Independent Deployment**: Services can be deployed without affecting others

Dependencies Matrix:

Service	Depends On
Inventory Service	Catalog Service (read-only: bus capacity)
Booking Service	User Service, Inventory Service, Catalog Service, Payment Service
Payment Service	Booking Service
Notification Service	Booking Service, User Service
Search Service	Catalog Service, Inventory Service

1.4 External Dependencies

The system interacts with the following external systems:

External System	Purpose	Integration Type	Services Using It
Payment Gateway (e.g., SePay, VNPay)	Process online payments, generate QR codes for bank transfers	REST API / Webhook	Payment Service
Email Provider (e.g., SMTP/Nodemailer)	Send transactional emails (confirmations, receipts, notifications)	SMTP Protocol	Notification Service
SMS Gateway (Optional)	Send SMS confirmations and reminders	REST API	Notification Service
Cloud Storage (e.g., Cloudinary)	Store and serve images (bus photos, company logos, station images)	REST API	Catalog Service
AI/ML Service (Embedding API)	Generate embeddings for semantic search	REST API / gRPC	Search Service, Catalog Service

Activity Practice 2: Defining Service Contracts

2.1 Overview of Service Contracts

Service contracts define the public API interfaces that each microservice exposes. These contracts serve as the **contract of communication** between services and with external clients. All services follow RESTful API design principles with JSON payloads.

2.2 Catalog Service API Contract

The Catalog Service manages all catalog-related data including buses, companies, routes, and stations.

Endpoint: Get Bus/Vehicle Details

Property	Value
Endpoint	/api/catalog/vehicles/{id}
HTTP Method	GET
Description	Retrieve detailed information about a specific bus/vehicle
Path Parameters	id (number) - Unique vehicle identifier
Query Parameters	*(none)*
Request Body	*(none)*

Response (200 OK):


```
{
  "success": true,
  "message": "Vehicle found",
  "data": {
    "id": 123,
    "name": "Mercedes Sprinter Luxury",
    "description": "40-seat luxury bus with reclining seats and WiFi",
    "license_plate": "29A-12345",
    "capacity": 40,
    "company_id": 5,
    "company_name": "Phuong Trang Express",
    "featured_image": "https://cloudinary.com/.../bus-image.jpg",
    "amenities": ["WiFi", "AC", "Reclining Seats", "USB Charging"],
    "created_at": "2025-12-01T10:30:00Z",
    "updated_at": "2026-01-05T08:15:00Z"
  }
}
```


Endpoint: Search Vehicles

Property	Value
Endpoint	/api/catalog/vehicles
HTTP Method	GET
Description	Search and list vehicles based on filter criteria
Query Parameters	- company_id (optional): Filter by bus company - min_capacity (optional): Minimum seat capacity

	- page (default: 1): Page number - limit (default: 10): Items per page

Response (200 OK):


```
{
  "success": true,
  "message": "Vehicles retrieved successfully",
  "data": {
    "results": [
      {
        "id": 123,
        "name": "Mercedes Sprinter Luxury",
        "license_plate": "29A-12345",
        "capacity": 40,
        "company_id": 5,
        "company_name": "Phuong Trang Express",
        "featured_image": "https://cloudinary.com/.../bus-image.jpg"
      }
    ],
    "pagination": {
      "page": 1,
      "limit": 10,
      "total": 45,
      "totalPages": 5
    }
  }
}
```


Endpoint: Create Vehicle (Admin)

Property	**Value**
Endpoint	/api/catalog/vehicles
HTTP Method	POST
Description	Add a new vehicle to the system (Admin only)
Authentication	Required (JWT with admin role)

Request Body:


```
{
  "name": "Mercedes Sprinter Luxury",
  "description": "40-seat luxury bus",
  "license_plate": "29A-12345",
  "capacity": 40,
  "company_id": 5,
  "featured_image": "https://cloudinary.com/.../image.jpg"
}
```


Response (201 Created):


```
{
  "success": true,
  "message": "Vehicle created successfully",
  "data": {
    "id": 124,
    "name": "Mercedes Sprinter Luxury",
    "license_plate": "29A-12345",
    "capacity": 40,
    "company_id": 5,
    "created_at": "2026-01-10T09:00:00Z"
  }
}
```


Endpoint: Get Route Details

Property	Value
Endpoint	/api/catalog/routes/{id}
HTTP Method	GET
Description	Retrieve detailed information about a specific route

Response (200 OK):


```
{
  "success": true,
  "message": "Route found",
  "data": {
```

```
    "id": 15,
    "departure_station": {
      "id": 1,
      "name": "Bến xe Miền Đông",
      "city": "Ho Chi Minh",
      "location": "292 Đinh Bộ Lĩnh, Bình Thạnh"
    },
    "arrival_station": {
      "id": 8,
      "name": "Bến xe Đà Lạt",
      "city": "Da Lat",
      "location": "1 Tô Hiến Thành, Phường 3"
    },
    "distance_km": 308.5,
    "estimated_duration_hours": 7.5,
    "is_active": true
  }
}
```


2.3 Booking Service API Contract

The Booking Service manages ticket reservations and booking lifecycle.

Endpoint: Create Booking

Property	**Value**
Endpoint	/api/bookings
HTTP Method	POST
Description	Create a new ticket booking
Authentication	Required (JWT)

Request Body:


```
{
  "user_id": 42,
  "schedule_id": 158,
  "seat_ids": [234, 235],
  "passenger_name": "Nguyen Van A",
```

```
  "passenger_phone": "0901234567",
  "passenger_email": "nguyenvana@email.com"
}
```


Response (201 Created):


```
{
  "success": true,
  "message": "Booking created successfully",
  "data": {
    "ticket_id": 5678,
    "ticket_number": "VVTD20260110-5678",
    "status": "PENDING",
    "total_price": 450000,
    "seats": ["S12", "S13"],
    "created_at": "2026-01-10T10:30:00Z"
  }
}
```


Endpoint: Get Booking Details

Property	Value
Endpoint	/api/bookings/{ticketId}
HTTP Method	GET
Description	Retrieve complete booking information
Authentication	Required (JWT)

Response (200 OK):


```
{
  "success": true,
  "data": {
    "ticket_id": 5678,
    "ticket_number": "VVTD20260110-5678",
    "status": "BOOKED",
    "passenger_name": "Nguyen Van A",
    "passenger_phone": "0901234567",
    "schedule": {
```

```
      "id": 158,
      "departure_time": "2026-01-15T08:00:00Z",
      "route": {
        "from": "Bến xe Miền Đông",
        "to": "Bến xe Đà Lạt"
      }
    },
    "seats": ["S12", "S13"],
    "total_price": 450000,
    "payment_status": "PAID"
  }
}
```


Endpoint: Cancel Booking

Property	Value
Endpoint	/api/bookings/{ticketId}/cancel
HTTP Method	POST
Description	Cancel an existing booking
Authentication	Required (JWT)

Request Body:


```
{
  "reason": "Change of travel plans"
}
```


Response (200 OK):


```
{
  "success": true,
  "message": "Booking cancelled successfully",
  "data": {
    "ticket_id": 5678,
    "status": "CANCELED",
    "refund_amount": 450000
  }
}
```

2.4 Inventory Service API Contract

The Inventory Service manages seat availability and reservations.

Endpoint: Check Seat Availability

Property	Value
Endpoint	/api/inventory/schedules/{scheduleId}/seats
HTTP Method	GET
Description	Get available seats for a specific schedule

Response (200 OK):


```
{
  "success": true,
  "data": {
    "schedule_id": 158,
    "total_seats": 40,
    "available_seats": 15,
    "seats": [
      {
        "id": 234,
        "seat_number": "S12",
        "seat_type": "VIP",
        "status": "AVAILABLE",
        "price": 250000
      },
      {
        "id": 235,
        "seat_number": "S13",
        "seat_type": "VIP",
        "status": "AVAILABLE",
        "price": 250000
      }
    ]
  }
}
```

Endpoint: Reserve Seats (Internal API)

Property	Value
Endpoint	/api/inventory/seats/reserve
HTTP Method	POST
Description	Reserve seats temporarily (called by Booking Service)
Authentication	Service-to-service authentication

Request Body:


```
{
  "seat_ids": [234, 235],
  "reservation_timeout": 300
}
```


Response (200 OK):


```
{
  "success": true,
  "message": "Seats reserved successfully",
  "data": {
    "reservation_id": "res-abc123",
    "expires_at": "2026-01-10T10:35:00Z"
  }
}
```


2.5 Payment Service API Contract

Endpoint: Initiate Payment

Property	Value
Endpoint	/api/payments/initiate
HTTP Method	POST
Description	Initiate payment process for a booking

Request Body:


```
{
  "ticket_id": 5678,
  "amount": 450000,
  "payment_method": "ONLINE"
```

}

Response (200 OK):

{
 "success": true,
 "data": {
  "payment_id": "pay-xyz789",
  "qr_code_url": "https://img.vietqr.io/...",
  "bank_transfer_info": {
   "bank_name": "VCB",
   "account_number": "1234567890",
   "amount": 450000,
   "content": "VVTD20260110-5678"
  },
  "expires_at": "2026-01-10T11:00:00Z"
 }
}

Endpoint: Payment Status

Property	Value
Endpoint	/api/payments/{paymentId}/status
HTTP Method	GET
Description	Check current payment status

Response (200 OK):

{
 "success": true,
 "data": {
  "payment_id": "pay-xyz789",
  "status": "PAID",
  "paid_at": "2026-01-10T10:45:00Z"
 }
}

2.6 Service Interaction: Booking Flow

This demonstrates how services interact when a user books a ticket:

Scenario: User adds a ticket to cart and proceeds to payment

1. **Frontend → Inventory Service**
   GET /api/inventory/schedules/158/seats
   → Frontend displays available seats
2. **Frontend → Booking Service**
   POST /api/bookings
   → Booking Service creates PENDING ticket
3. **Booking Service → Inventory Service** (Internal)
   POST /api/inventory/seats/reserve (with seat_ids: [234, 235])
   → Inventory Service reserves seats temporarily
4. **Booking Service → Payment Service** (Internal)
   POST /api/payments/initiate
   → Payment Service generates QR code and bank details
5. **Payment Service → External Payment Gateway**
   Webhook listens for payment confirmation
6. **Payment Gateway → Payment Service** (Webhook)
   Sends payment confirmation
7. **Payment Service → Booking Service** (Internal Event/API)
   Notifies payment success
8. **Booking Service → Inventory Service** (Internal)
   POST /api/inventory/seats/confirm
   → Confirms seat reservation (status: BOOKED)
9. **Booking Service → Notification Service** (Event)
   Publishes BookingConfirmed event
10. **Notification Service → Email Provider**
    Sends confirmation email to customer

Key Principle: The Booking Service never directly accesses the Inventory Service's database. All interactions happen through defined API endpoints.

Activity Practice 3: Communication Strategy Design

3.1 Synchronous vs Asynchronous Communication

The system employs both synchronous (HTTP/REST) and asynchronous (Message Queue/Events) communication patterns based on the nature of the interaction.

Communication Strategy Table

Interaction	Services Involved	Communication Type	Protocol/Technology	Rationale

User Login	API Gateway → User Service	Synchronous	HTTP/REST	Immediate response required for authentication
Seat Availability Lookup	Booking Service → Inventory Service	Synchronous	HTTP/REST	Real-time data needed before checkout
Vehicle Information	Booking Service → Catalog Service	Synchronous	HTTP/REST	Data needed immediately for display
Payment Initiation	Booking Service → Payment Service	Synchronous	HTTP/REST	Immediate payment details required
Booking Confirmation Email	Booking Service → Notification Service	Asynchronous	Message Queue (RabbitMQ/Kafka)	Email can be sent in background; doesn't block booking
Payment Status Update	Payment Gateway → Payment Service	Asynchronous	Webhook	External system pushes updates asynchronously
Trip Reminder	Scheduler → Notification Service	Asynchronous	Message Queue	Batch operation, not time-critical
Analytics Data Collection	All Services → Analytics Service	Asynchronous	Event Stream (Kafka)	Reporting doesn't require real-time processing
Search Index Update	Catalog Service → Search Service	Asynchronous	Event Stream	Index can be updated eventually

3.2 Synchronous Communication Details

Technology: HTTP/REST with JSON payloads

Use Cases:

- Request-response operations requiring immediate data
- User-facing operations where latency matters

- Transactional operations requiring strong consistency

Advantages:

- Simple to implement and debug
- Immediate feedback to users
- Strong consistency guarantees

Challenges:

- Tight coupling between services
- Cascading failures if dependent service is down
- Increased latency for chained calls

Mitigation Strategies:

- Circuit breaker pattern (e.g., using Resilience4j, Hystrix)
- Timeout configuration
- Retry logic with exponential backoff
- Fallback responses

3.3 Asynchronous Communication Details

Technology: Message Queue (RabbitMQ/Apache Kafka) or Event Streaming

Use Cases:

- Notifications (email, SMS)
- Background processing
- Event-driven workflows
- Eventual consistency scenarios

Advantages:

- Loose coupling between services
- Better fault tolerance (messages can be retried)
- Improved scalability
- Non-blocking operations

Challenges:

- Complexity in debugging distributed flows
- Eventual consistency (data might not be immediately synchronized)
- Message ordering and duplicate handling

Event Examples:

// Event: BookingConfirmed

```
{
  "event_type": "BookingConfirmed",
  "timestamp": "2026-01-10T10:45:00Z",
  "data": {
    "ticket_id": 5678,
    "ticket_number": "VVTD20260110-5678",
    "user_id": 42,
    "email": "nguyenvana@email.com",
    "schedule_id": 158,
    "total_price": 450000
  }
}
// Event: PaymentCompleted
{
  "event_type": "PaymentCompleted",
  "timestamp": "2026-01-10T10:45:00Z",
  "data": {
    "payment_id": "pay-xyz789",
    "ticket_id": 5678,
    "amount": 450000,
    "payment_method": "ONLINE"
  }
}
```


3.4 API Gateway Pattern

All client requests (web, mobile) go through an **API Gateway** which:

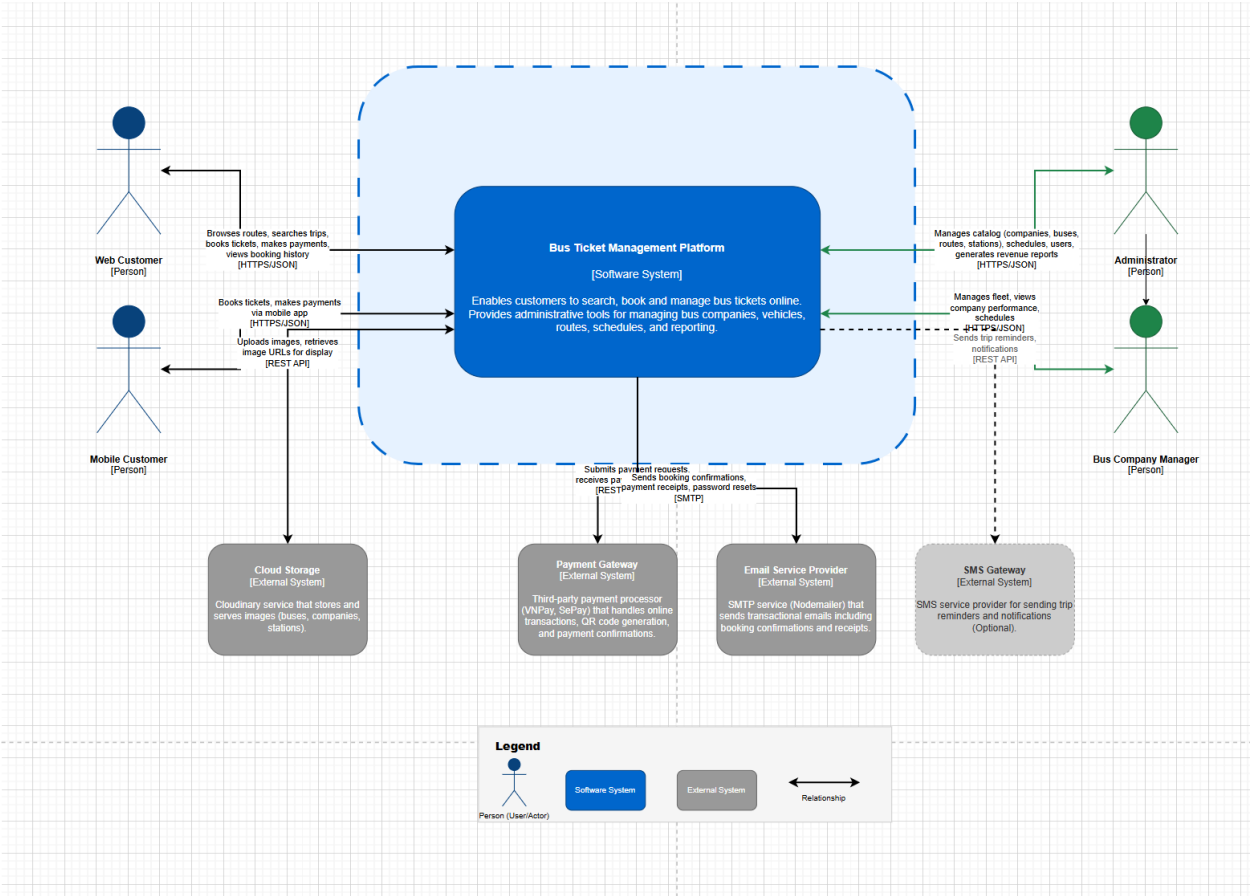
- Routes requests to appropriate microservices
- Handles authentication/authorization (JWT validation)
- Implements rate limiting
- Aggregates responses from multiple services
- Provides unified error handling

Benefits:

- Single entry point for clients

- Simplified client logic
- Centralized cross-cutting concerns (auth, logging, monitoring)

Activity Practice 4: C4 Model (Level 1: System Context)



4.1 System Context Diagram Overview

The C4 System Context diagram provides a high-level view of the Bus Ticket Management System, showing:

- The system boundary
- External actors (users)
- External systems (third-party integrations)
- High-level interactions

4.2 Actors and External Systems

Actors:

1. **Web Customer** - End users booking tickets through web interface
2. **Mobile Customer** - End users booking tickets through mobile app
3. **Administrator** - Staff managing catalog, schedules, and operations
4. **Bus Company Manager** - Company representatives managing their fleet

External Systems:

1. **Payment Gateway** (VNPay/SePay) - Processes online payments
2. **Email Service Provider** (SMTP) - Sends transactional emails
3. **Cloud Storage** (Cloudinary) - Stores and serves images
4. **SMS Gateway** (Optional) - Sends SMS notifications

4.3 System Interactions

Customer Interactions:

- Browses available bus routes and schedules
- Searches for trips using semantic search
- Books tickets and selects seats
- Makes online payments
- Views booking history and invoices
- Receives email confirmations

Administrator Interactions:

- Manages bus companies and vehicles
- Configures routes and stations
- Creates and manages schedules
- Views revenue reports and trip statistics
- Manages user accounts

System to External System:

- → **Payment Gateway**: Submits payment requests, receives confirmations
- → **Email Provider**: Sends booking confirmations, receipts, password resets
- → **Cloud Storage**: Uploads images, retrieves image URLs
- → **SMS Gateway**: Sends trip reminders and notifications

4.4 C4 Context Diagram (Level 1)

The complete C4 System Context diagram has been created and saved at:

File Location: architecture/C4_model.drawio

The diagram includes:

- Large container representing the **[System] Bus Ticket Management Platform**
- Actors positioned outside system boundary (Web Customer, Administrator, Bus Company Manager)
- External systems with clear integration points

- Directional arrows showing data flow
- Labels describing the nature of each interaction

Key Relationships:

- Web Customer ← → : "Browses routes, books tickets, makes payments"
- Administrator → : "Manages catalog, schedules, and reports"
- → Payment Gateway: "Submits payment for processing"
- → Email Provider: "Sends booking confirmations and notifications"
- → Cloud Storage: "Uploads/retrieves images"

Microservices Deployment Considerations

5.1 Database Strategy

Approach: Database per Service (Logical Separation)

Each microservice owns its database schema/tables:

- **User Service**: users table
- **Catalog Service**: bus_companies, cars, routes, stations, bus_reviews tables
- **Booking Service**: tickets table
- **Inventory Service**: seats, schedules tables
- **Payment Service**: Separate payments table (to be created)

Rationale:

- Enables independent scaling
- Prevents tight coupling through shared databases
- Allows technology diversity (e.g., using PostgreSQL for some services, MongoDB for others)

Challenge:

- Distributed transactions (solved using Saga pattern or eventual consistency)
- Data duplication (acceptable trade-off for autonomy)

5.2 Service Discovery & Load Balancing

Technology Options:

- **Netflix Eureka** - Service registry
- **Consul** - Service mesh with health checking
- **Kubernetes** - Built-in service discovery and load balancing

Benefits:

- Dynamic service registration
- Health monitoring
- Automatic failover

5.3 Containerization & Orchestration

Docker Containers: Each microservice packaged as a Docker container with:

- Application code
- Runtime dependencies
- Environment configuration

Kubernetes Orchestration:

- Manages container deployment
- Auto-scaling based on metrics
- Rolling updates with zero downtime
- Self-healing (restarts failed containers)

Transition Plan from Monolith to Microservices

6.1 Phased Approach (Strangler Fig Pattern)

Rather than a "big bang" migration, we use an incremental approach:

Phase 1: Extract High-Value Services

- Extract **Payment Service** first (clear boundary, high business value)
- Extract **Notification Service** (minimal dependencies)

Phase 2: Core Domain Services

- Extract **Catalog Service** (foundational data)
- Extract **Inventory Service** (enables independent scaling)

Phase 3: Orchestration Services

- Extract **Booking Service** (orchestrates multiple services)
- Extract **Search Service** (specialized functionality)

Phase 4: Decommission Monolith

- Migrate remaining logic
- Retire monolithic application

6.2 API Gateway Implementation

Implement API Gateway to:

- Route requests to monolith OR microservices based on routing rules
- Gradually shift traffic to microservices
- Maintain backward compatibility during migration

Conclusion

This lab has successfully decomposed the Bus Ticket Management System into **7 core microservices** based on business capabilities:

1. **User Service** - Authentication and profile management
2. **Catalog Service** - Buses, companies, routes, stations
3. **Booking Service** - Ticket reservations and lifecycle
4. **Inventory Service** - Seat availability and schedules
5. **Payment Service** - Payment processing and tracking
6. **Notification Service** - Email and SMS communications
7. **Search Service** - Semantic search and recommendations

Each service has:

- **Clear boundaries** - Well-defined data ownership
- **Defined contracts** - RESTful API specifications
- **Communication strategy** - Synchronous (HTTP) or Asynchronous (Events)
- **Independence** - Can be deployed and scaled independently

The **C4 System Context diagram** provides a high-level architectural view showing how the system interacts with external actors and systems.

The microservices architecture enables **scalability**, **fault isolation**, **technology diversity**, and **team autonomy**, addressing the Architecturally Significant Requirements (ASRs) identified in Lab 1:

- **ASR-1**: Real-time seat inventory (Inventory Service scales independently)
- **ASR-2**: Multi-tenant management (Catalog Service handles tenant isolation)
- **ASR-3**: High availability (Services scale horizontally with load balancers)