PHENIKAA UNIVERSITY

SCHOOL OF COMPUTING

COURSE MODULE: SOFTWARE ARCHITECTURE

PROJECT TITLE: BUILDING A BUS TICKET BOOKING SYSTEM

Instructor:	TS. Vũ Quang Dũng	
Group:	Nguyễn Minh Nguyệt	(23010408)
Class:	CSE703110-1-2-25(N02)	

Ha Noi, January 30, 2026

CONTENTS

IMAGE CATALOG

PREFACE

In the context of the rapid development of information technology, the application of software systems to management and service delivery has become an inevitable trend across many areas of modern society. Particularly in the passenger transportation sector, the increasing demand for fast, convenient, and transparent ticket booking requires software systems not only to provide correct functionality but also to ensure scalability, reliability, high performance, and long-term maintainability.

The Software Architecture course plays an important role in equipping students with systematic thinking at the architectural level, including decomposing systems into independent components, selecting appropriate architectural styles, defining communication mechanisms among services, and optimizing quality attributes such as scalability, availability, and maintainability. Through this course, students are exposed to modern architectural approaches, especially Microservices Architecture, which enables the development of flexible and sustainable large-scale systems in real-world environments.

Based on these practical needs, our team selected the topic "**BUILDING A BUS TICKET BOOKING SYSTEM**" for this course project. The project focuses on requirement analysis, overall architectural design, service decomposition, inter-service communication mechanisms, and system deployment models. Through this process, the team has the opportunity to apply theoretical knowledge to solve real-world problems while strengthening architectural thinking and system design skills.

Although we have made considerable efforts to complete this project, due to limitations in time, experience, and research scope, shortcomings are unavoidable. Therefore, we sincerely look forward to receiving valuable feedback and suggestions from our instructor to further improve the quality of this project.

ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to TS. Vũ Quang Dũng, our lecturer for the Software Architecture course, for his dedicated guidance, valuable feedback, and professional support throughout the completion of this project. His knowledge and practical experience have greatly helped us develop a deeper understanding of architectural design principles and successfully accomplish this work.

We would also like to thank the School of Information Technology – Phenikaa University for providing favorable learning conditions, facilities, and academic resources that enabled us to conduct research and complete this project effectively.

Finally, we would like to thank our friends and classmates for their support, discussions, and constructive suggestions during the learning and development process. Their encouragement has been a significant motivation for us to complete our tasks.

Despite our best efforts, the project may still contain limitations and mistakes. We highly appreciate any comments and recommendations from our instructor to help us improve and gain better experience for future studies and projects.

1. Executive Summary

This project aims to design and implement a Bus Ticket Booking System utilizing a Microservices Architecture with Event-Driven Communication patterns. The system enables passengers to search for bus routes, view available schedules, select seats, and complete ticket bookings with integrated payment processing.

The architectural approach decomposes the monolithic application into six distinct microservices: Identity Service, Fleet Service, Route Service, Schedule Service, Ticket Service, and an API Gateway. These services communicate through RESTful APIs for synchronous operations and Apache Kafka for asynchronous event-driven messaging. Redis is employed for distributed seat locking to handle concurrent booking scenarios.

The final achievement is a fully functional bus ticket booking platform that demonstrates:

- High Scalability through service decomposition and horizontal scaling capabilities
- Loose Coupling via event-driven communication patterns
- Real-time Seat Availability using Redis distributed locks
- Reliable Payment Processing with webhook integration (SePay)
- Responsive User Experience through two client applications (Customer Portal and Admin Dashboard)

2. Project Requirements & Goals

This section defines the scope and the key architectural drivers for the project.

2.1 Core Functional Requirements

ID	Description
FR-01	The system must allow users to register and authenticate using email/username and password with JWT-based security.
FR-02	The system must allow users to search for available bus routes by departure station, arrival station, and travel date.
FR-03	The system must display available bus schedules for a selected route, including departure time, vehicle information, and pricing.
FR-04	The system must show real-time seat availability for each vehicle schedule, with seat types (Standard, VIP, Luxury) and pricing.
FR-05	The system must allow users to select and temporarily lock seats during the booking process to prevent double booking.
FR-06	The system must integrate with payment gateway (SePay) for processing ticket payments via bank transfer/QR code.
FR-07	The system must confirm bookings and update ticket status upon successful payment verification via webhook.
FR-08	The system must allow users to view their booking history and cancel tickets with valid reasons.
FR-09	The system must provide an admin dashboard for managing bus companies, vehicles, routes, stations, and schedules.
FR-10	The system must send notifications to users upon booking confirmation or cancellation.

2.2 Key Quality Attributes (Architectural Goals – Non-functional Requirements)

The following quality attributes serve as the primary architectural drivers for the system design:

Quality Attribute	Description	Target Metric
Scalability	The system must handle a high number of concurrent users during peak booking hours.	Support 10,000+ concurrent active users
Availability	The system must maintain high uptime and handle service failures gracefully.	99.5% uptime with graceful degradation
Performance	API response times must be optimized for user experience.	< 500ms for seat availability queries
Consistency	Seat booking operations must be consistent to prevent double bookings.	Zero double-booking incidents
Modifiability	The architecture must allow adding new features (e.g., chatbot, reviews) without major refactoring.	New service deployment within hours
Security	User authentication and payment data must be protected.	JWT-based auth, secure webhook validation

2.3 Propose model of ASR

a. ASR: Real-Time Seat Locking (Consistency)

Component	**Details**
Statement	When a user initiates booking for a seat, the seat must be temporarily locked for 5 minutes to prevent other users from booking the same seat.
Impact	This drives the implementation of Redis-based distributed locking mechanism. The lock ensures atomicity across multiple service instances and provides TTL-based automatic release.

b. ASR: Payment Integration Reliability

Component	**Details**
Statement	Payment confirmation must be processed reliably through external payment gateway webhooks, with idempotent processing to handle duplicate notifications.
Impact	This requires implementing idempotent webhook handlers, storing payment transaction IDs (SePay ID), and using Kafka events to decouple payment processing from ticket confirmation.

2.4 Use Case Modeling

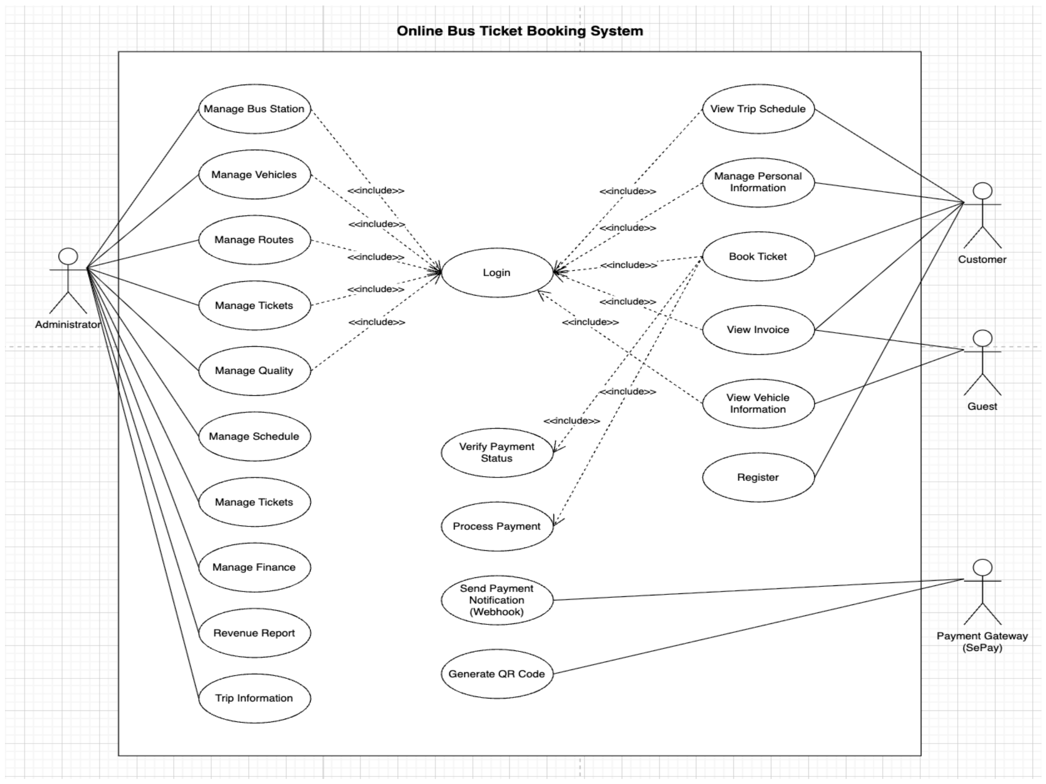

Figure 1. Overall Use Case Diagram

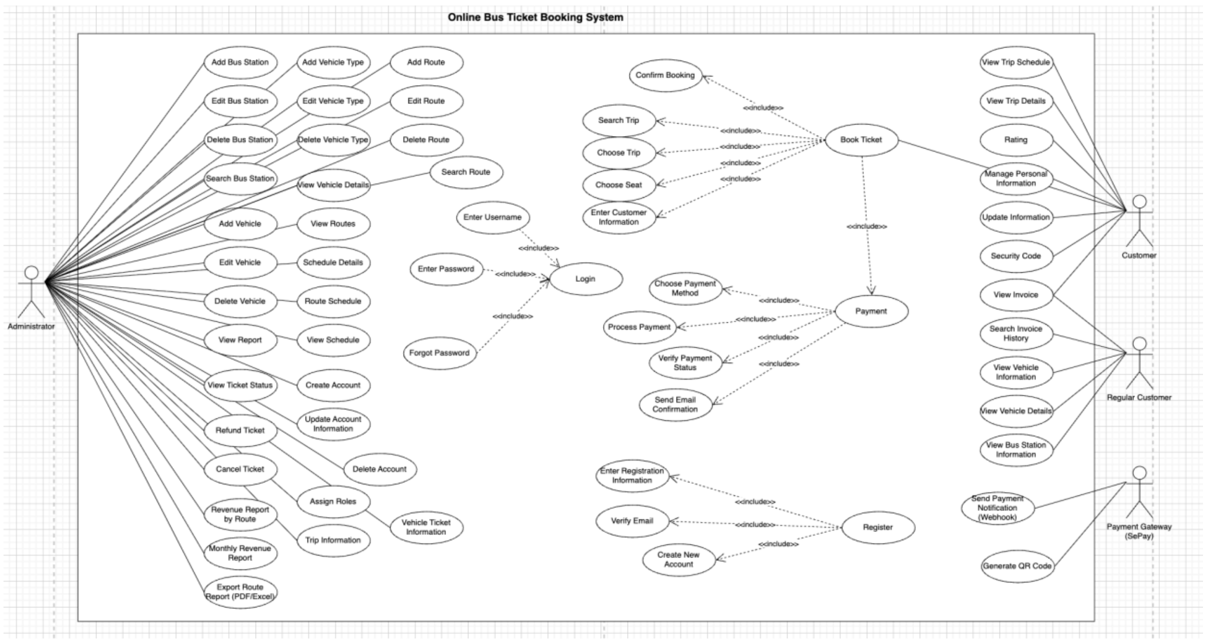

Figure 2. Detailed Use Case Diagram

Analysis Diagrams

This section uses UML analysis diagrams to visually represent the system's requirements, behavior, structure, and component interactions in the online bus ticket booking system, supporting better understanding and guiding the design process.Architectural Design & Implementation.

3. Architectural Design & Implementation

3.1 Architectural Pattern: Event-Driven Architecture (EDA)

The system implements Microservices Architecture combined with Event-Driven Architecture (EDA) patterns. This hybrid approach provides:

- Service Decomposition: The domain is decomposed into bounded contexts, each implemented as an independent microservice with its own database (Database-per-Service pattern).
- API Gateway Pattern: A centralized gateway handles request routing, authentication, and cross-cutting concerns.
- Event-Driven Communication: Services communicate asynchronously via Kafka for operations that don't require immediate response (e.g., payment confirmation, seat status updates).
- Distributed Locking: Redis provides distributed locks for critical sections (seat reservation) across service instances.

Architectural Rationale:

Pattern	Justification
Microservices	Enables independent deployment, scaling, and technology choices per service. Supports team autonomy and fault isolation.
API Gateway	Single entry point simplifies client integration, enables centralized security, and provides routing flexibility.
Event-Driven	Decouples services temporally, improving resilience. Payment and notification operations benefit from async processing.
Redis Locking	Essential for maintaining consistency in concurrent seat booking scenarios across multiple service instances.
Database-per-Service	Each service owns its data, enabling independent schema evolution and reducing coupling.

3.2 System Context Diagram (C4 Model)

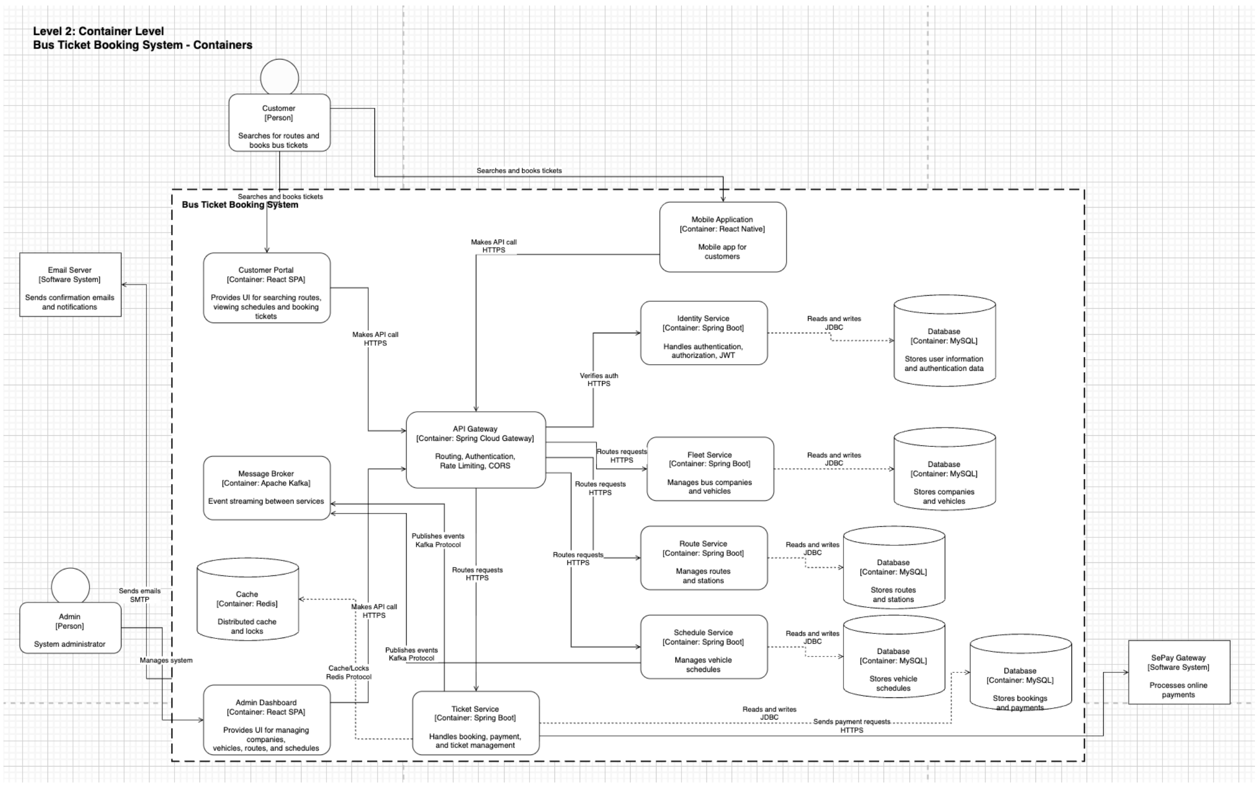
Level 2: Container Level
Bus Ticket Booking System - Containers
Customer [Person] Searches for routes and books bus tickets
Searches and books tickets
Searches and books tickets
Bus Ticket Booking System
Mobile Application [Container: React Native] Mobile app for customers
Makes API call HTTPS
Email Server [Software System] Sends confirmation emails and notifications
Customer Portal [Container: React SPA] Provides UI for searching routes, viewing schedules and booking tickets
Makes API call HTTPS
Identity Service [Container: Spring Boot] Handles authentication, authorization, JWT
Reads and writes JDBC
Database [Container: MySQL] Stores user information and authentication data
Verifies auth HTTPS
API Gateway [Container: Spring Cloud Gateway] Routing, Authentication, Rate Limiting, CORS
Routes requests HTTPS
Fleet Service [Container: Spring Boot] Manages bus companies and vehicles
Reads and writes JDBC
Database [Container: MySQL] Stores companies and vehicles
Message Broker [Container: Apache Kafka] Event streaming between services
Routes requests HTTPS
Publishes events Kafka Protocol
Routes requests HTTPS
Routes requests HTTPS
Route Service [Container: Spring Boot] Manages routes and stations
Reads and writes JDBC
Database [Container: MySQL] Stores routes and stations
Cache [Container: Redis] Distributed cache and locks
Sends emails SMTP
Makes API call HTTPS
Admin [Person] System administrator
Schedule Service [Container: Spring Boot] Manages vehicle schedules
Reads and writes JDBC
Database [Container: MySQL] Stores vehicle schedules
Publishes events Kafka Protocol
Manages system
Cache/Locks Redis Protocol
Database [Container: MySQL] Stores bookings and payments
SePay Gateway [Software System] Processes online payments
Admin Dashboard [Container: React SPA] Provides UI for managing companies, vehicles, routes, and schedules
Ticket Service [Container: Spring Boot] Handles booking, payment, and ticket management
Reads and writes JDBC
Sends payment requests HTTPS

Figure 3. Level 2 - Container Level

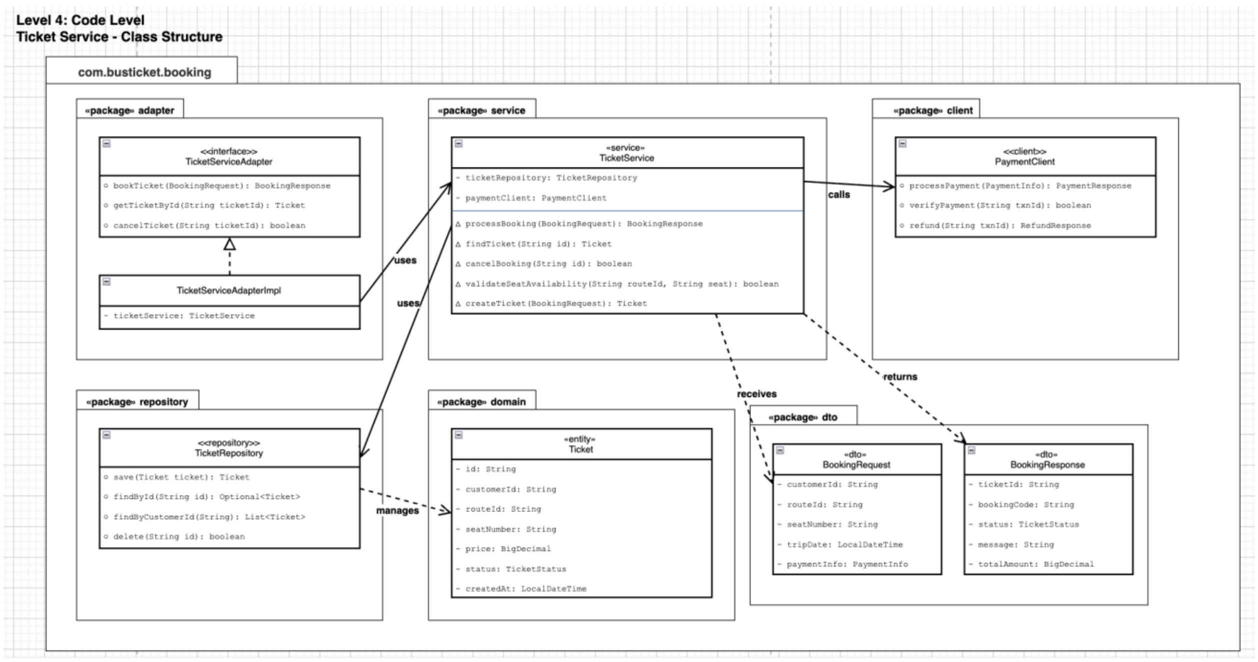
Level 4: Code Level
Ticket Service - Class Structure
com.busticket.booking
«package» adapter
<<interface>> TicketServiceAdapter
○ bookTicket(BookingRequest): BookingResponse
○ getTicketById(String ticketId): Ticket
○ cancelTicket(String ticketId): boolean
TicketServiceAdapterImpl
- ticketService: TicketService
«package» service
«service» TicketService
- ticketRepository: TicketRepository
- paymentClient: PaymentClient
△ processBooking(BookingRequest): BookingResponse
△ findTicket(String id): Ticket
△ cancelBooking(String id): boolean
△ validateSeatAvailability(String routeId, String seat): boolean
△ createTicket(BookingRequest): Ticket
uses
uses
calls
«package» client
<<client>> PaymentClient
○ processPayment(PaymentInfo): PaymentResponse
○ verifyPayment(String txnId): boolean
○ refund(String txnId): RefundResponse
receives
returns
«package» repository
<<repository>> TicketRepository
○ save(Ticket ticket): Ticket
○ findById(String id): Optional<Ticket>
○ findByCustomerId(String): List<Ticket>
○ delete(String id): boolean
manages
«package» domain
«entity» Ticket
- id: String
- customerId: String
- routeId: String
- seatNumber: String
- price: BigDecimal
- status: TicketStatus
- createdAt: LocalDateTime
«package» dto
«dto» BookingRequest
- customerId: String
- routeId: String
- seatNumber: String
- tripDate: LocalDateTime
- paymentInfo: PaymentInfo
«dto» BookingResponse
- ticketId: String
- bookingCode: String
- status: TicketStatus
- message: String
- totalAmount: BigDecimal

Figure 4. Level 4 - Code Level

Identity Service:

Handles user authentication and authorization with JWT tokens.

Endpoint	**Method**	**Description**
/auth/register	POST	User registration with validation
/auth/login	POST	Authentication, returns JWT token
/auth/logout	POST	Token invalidation
/auth/profile	GET	Retrieve current user profile

Fleet Service:

Manages bus companies, vehicles, and seat configurations.

Endpoint	**Method**	**Description**
/bus-companies	GET/POST	List/Create bus companies
/bus-companies/:id	GET/PUT/DELETE	CRUD operations on specific company
/vehicles	GET/POST	List/Create vehicles with company association
/seats/vehicle/:vehicleId	GET	Get all seats for a vehicle

Route Service:

Handles geographic routing between stations.

Endpoint	**Method**	**Description**
/stations	GET/POST	List/Create stations with location data
/routes	GET/POST	List/Create routes with departure/arrival stations

Schedule Service:

Manages vehicle departure schedules.

Endpoint	Method	Description
/vehicle-schedules	GET/POST	List/Create schedules
/vehicle-schedules?routeId=X	GET	Filter schedules by route

Ticket Service:

Orchestrates the booking lifecycle with Redis locking and Kafka events.

Endpoint	Method	Description
/tickets	POST	Create ticket (acquires Redis lock)
/tickets/user/me	GET	Get current user's tickets
/tickets/:id	PUT	Update ticket (cancel, etc.)
/tickets/webhook/sepay	POST	Receive SePay payment webhook

3.3 Inter-Service Communication

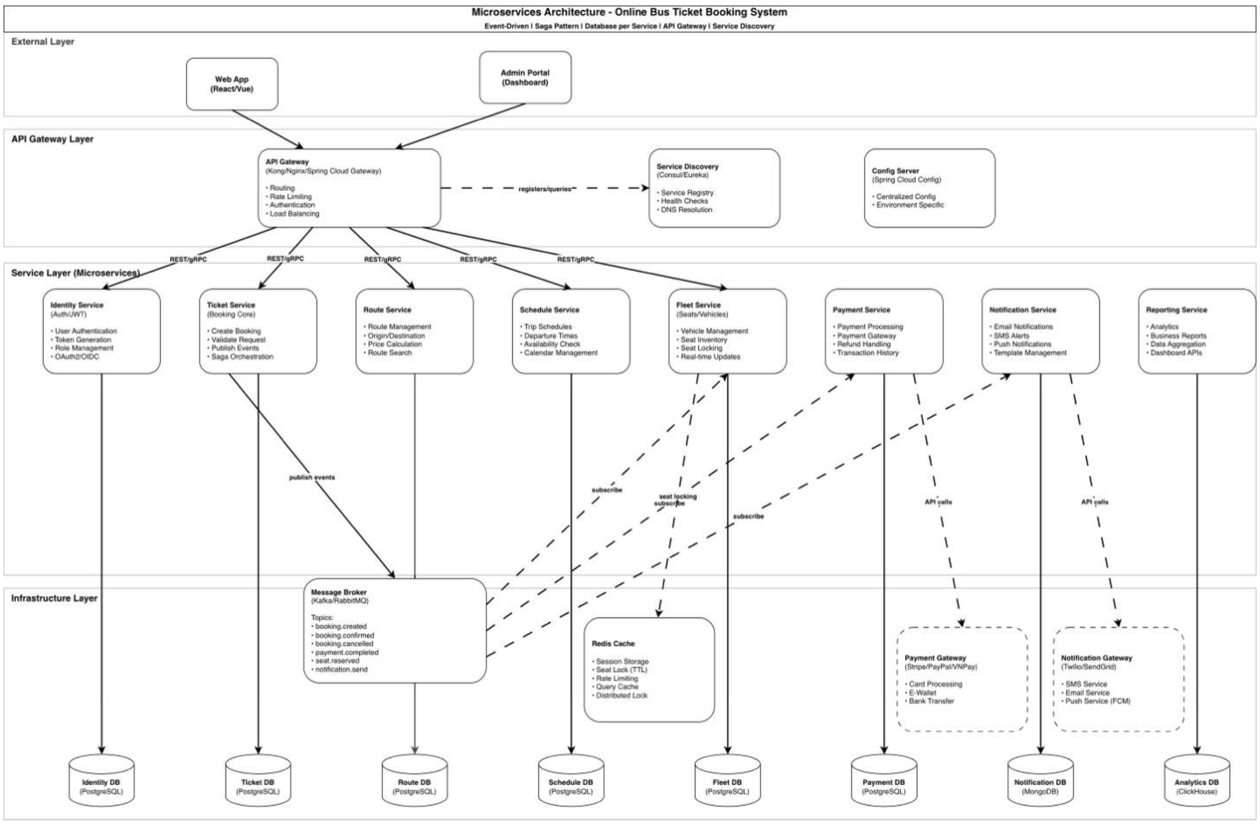

Figure 5. Event Flow Diagram

The diagram show the flow:

- The system adopts a Microservices Architecture combined with an API Gateway and Event-Driven communication. All client requests from the Web App and Admin Portal pass through the API Gateway, which manages authentication, routing, rate limiting, and load balancing before forwarding them to the appropriate services.
- Services communicate synchronously via REST/gRPC for real-time operations and asynchronously through a message broker (Kafka/RabbitMQ) for cross-service workflows, ensuring loose coupling and scalability. The ticket booking process follows the Saga pattern: the Ticket Service creates a booking, Fleet Service locks seats using Redis, Payment Service processes payment, and Notification Service sends confirmations.
- Each microservice maintains its own database, while Redis supports caching and distributed locking. This architecture enhances scalability, fault isolation, and maintainability.

4. Conclusion & Reflection

This project successfully implemented a comprehensive Bus Ticket Booking System using a Microservices Architecture with Event-Driven communication patterns. The architecture demonstrates several key software engineering principles:

Achievements:

Goal	**Achievement**
Microservices Decomposition	6 independent services with clear bounded contexts
Event-Driven Architecture	Kafka-based async messaging for payment and ticket events
Distributed Locking	Redis-based seat locking with graceful degradation
API Gateway Pattern	Centralized routing, authentication, and CORS handling
Clean Architecture	Layered service structure with separation of concerns
Database Per Service	Independent MySQL databases per service domain

4.1 Lessons Learned

Microservices trade-off: Service decomposition increases operational complexity but enables independent scaling and deployment.
Event-driven resilience: Kafka decouples services and allows workflows to operate asynchronously and reliably.
Distributed consistency: Redis locking with TTL effectively prevents concurrent double bookings.
API Gateway security: Centralized JWT validation simplifies services and ensures consistent protection.
Graceful degradation: Fallback mechanisms improve overall system reliability during failures.

4.2 Future Improvements

Improvement	Description	Priority
Service Discovery	Implement Eureka or Consul for dynamic service registration and discovery instead of hardcoded URLs	High
Circuit Breaker	Add Resilience4j circuit breakers for fault tolerance in inter-service calls	High
Centralized Logging	Implement ELK Stack (Elasticsearch, Logstash, Kibana) for aggregated log analysis	Medium
Distributed Tracing	Add Zipkin/Jaeger for request tracing across services	Medium
Kubernetes Deployment	Containerize and deploy to Kubernetes for production-grade orchestration	High
Notification Service	Implement dedicated microservice for email/SMS notifications via RabbitMQ	Medium
Review Service	Add customer review and rating functionality for bus companies	Low
Chatbot Integration	Implement AI-powered chatbot for customer support	Low
Payment Retry	Add retry mechanism for failed webhook deliveries	Medium
Caching Layer	Implement Redis caching for frequently accessed data (routes, schedules)	Medium

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Boston, MA, USA: Addison-Wesley, 2021.

[2] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O’Reilly Media, 2020.

[3] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2021.

[4] C. Richardson, *Microservices Patterns: With Examples in Java*. Shelter Island, NY, USA: Manning Publications, 2018.

[5] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley, 2004.

[6] M. Fowler, “Microservices Architecture,” martinfowler.com, 2014. [Online]. Available: https://martinfowler.com/microservices/architecture.html

[7] S. Newman, “API Gateway Pattern,” *microservices.io*. [Online]. Available: https://microservices.io/patterns/apigateway.html

[8] Apache Software Foundation, “Apache Kafka Documentation,” 2025. [Online]. Available: https://kafka.apache.org/documentation/

[9] Redis Ltd., “Redis Documentation – Distributed Locks and Caching,” 2025. [Online]. Available: https://redis.io/docs/

[10] S. Brown, “The C4 Model for Visualising Software Architecture,” 2023. [Online]. Available: https://c4model.com/

[11] OWASP Foundation, “JSON Web Token (JWT) Security Best Practices,” 2024. [Online]. Available: https://owasp.org/