

Introduction

This document explains the design of a scalable, medium-complexity Event Ticketing & Seat Allocation system.

It focuses on ensuring fairness, zero overselling, and smooth performance during high-demand event launches.

The system is designed for real-world constraints and is suitable for a college-level project submission.

Problem Overview

Ticket systems experience flash-sale spikes with millions of requests in seconds. This creates challenges:

- Overselling seats
- Slow checkout due to DB locks
- Payment and reservation sync issues
- High concurrency read/write load

This system addresses these issues through modular services and strong consistency models.

Stakeholders

Primary stakeholders:

- Buyers – browse and purchase tickets
- Event Organizers – create events and review sales
- Finance Teams – handle payments, refunds, reports
- Support Teams – troubleshoot booking issues

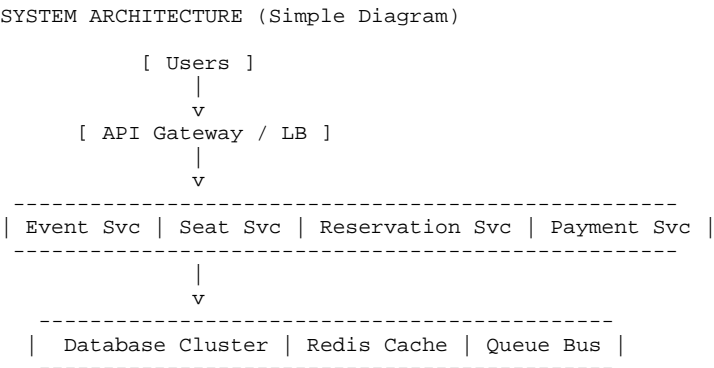
Functional Requirements

- Users can browse events
- Users can view real-time seating availability
- Seats can be reserved with strict locking
- Payments are processed securely
- Reservations expire automatically
- Users receive digital tickets

Non-Functional Requirements

- p99 checkout time < 2s
- 200k+ concurrent users supported
- 99.95% availability
- Strong consistency on seat commits
- Eventual consistency on event listings
- High maintainability via microservices

Architecture Diagram



Seat Reservation Flow

SEAT RESERVATION FLOW (Simple Diagram)

```
User → Select Seat
      → API Gateway
      → Reservation Service
          → Check Seat
          → Lock Seat (Atomic)
          → Create Temporary Hold (TTL)
User → Payment
Payment Service → Verify Payment
Reservation Service → Commit Seat
```

ER Diagram

DATA MODEL (Simple ERD)

[Event]

- EventID
- Name
- Time
- VenueID

|

1-to-many

v

[Seat]

- SeatID
- EventID
- Status

[Reservation]

- ReservationID
- UserID
- Expiry
- SeatIDs (list)

[Payment]

- PaymentID
- Amount
- Status

Detailed Architecture Explanation

The system follows a microservice architecture with an API Gateway acting as the entry point.

Each service handles a well-defined domain:

- Event Service – manages event data
- Seat Service – tracks seat inventory
- Reservation Service – locks and reserves seats
- Payment Service – handles transaction lifecycle

Database:

- SQL used for structured entities (events, seats)
- Redis used for seat availability caching
- Queues handle asynchronous tasks such as reservation expiry, email notifications

Reservation Logic Deep Dive

The reservation workflow:

1. User requests seat lock
2. Service checks availability from cache
3. If available → lock seat atomically
4. Create reservation with TTL (2–5 minutes)
5. User proceeds to payment
6. On payment success → commit seat permanently
7. If payment fails or TTL expires → lock is released automatically

This ensures zero overselling even during concurrency spikes.

Caching Strategy

Redis cache:

- Stores hot data (seat status, event metadata)
- Reduces DB load significantly
- Enables microsecond-level reads

Cache invalidation uses:

- Write-through updates
- TTL-based clearing for expired events

Failure Handling

- Payment success but commit failure: retry using idempotency keys
- Reservation service crash: uncommitted locks restored via DB + TTL
- Network issues: exponential backoff and retries
- Full system outage: multi-region failover supported through stateless services

Conclusion

This architecture is a balanced, scalable, and fault-tolerant system. It prevents overselling,

supports high concurrency, ensures fast checkout, and stays maintainable with clear service boundaries.

This version is suitable for academic submission and demonstrates strong system design principles.