



Scalable Event Ticketing & Seat Allocation System Design

[Abstract](#)

**A Technical Report Submitted in Fulfillment of Academic
Requirements**

A Ramcharan
12413985
K24BX
09

ramcharanambur@gmail.com

Contents

1 Introduction and System Context

- 1.1 Design Goals and Non-Functional Requirements (NFRs)
- 1.2 System Decomposition and Service Map
- 1.3 Stakeholder Requirements

2 Data Model and Consistency

- 2.1 Seat State Management and Consistency Model
 - 2.1.1 Seat State Transition Flowchart
 - 2.1.2 Entity-Relationship (ER) Schema Outline

3 Scalability, Caching, and Transaction Flow

- 3.1 Capacity Planning and Estimates
- 3.2 API Contracts (Seat Service)
- 3.3 Checkout Read/Write Path Flowchart
- 3.4 Payment Service Integration and Idempotency

4 Data Sharding and Scaling

- 4.1 Seat Database Scaling Strategy
 - 4.1.1 Sharding Key Selection
 - 4.1.2 Hot Shard Management (Flash Sale)

5 Asynchronous Processing and Queuing

- 5.1 Queuing Strategy (Kafka)
- 5.2 Reservation Cleanup Strategy

6 Resilience and Failure Management

- 6.1 Oversell Prevention and Lock Management
- 6.2 Failure Scenarios and Mitigation
- 6.3 Rate Limiting Implementation

7 Monitoring and Observability

- 7.1 Key Observability Targets
 - 7.1.1 Metrics and Alerting (Prometheus/Grafana)
- 7.2 Distributed Tracing (Jaeger)

8 Future Work and Maintenance

- 8.1 Maintainability and Operations
- 8.2 Future Enhancements

Chapter 1

Introduction and System Context

This report outlines the architecture for a highly available and scalable Event Ticketing System, specifically designed to withstand high-volume, spiky traffic during flash sales while maintaining strong consistency in seat allocation to prevent oversells. The system relies on a distributed microservices pattern.

The Scalable Event Ticketing & Seat Allocation System is engineered to manage extreme traffic spikes, such as those during flash sales, while ensuring no overselling of seats through strong consistency mechanisms. Built on a microservices architecture, the system decouples concerns like event browsing, seat reservation, payment processing, and order fulfillment to achieve high availability and scalability. The core pathway for seat commitment prioritizes atomic transactions in a distributed database to prevent race conditions under concurrent access.

Design goals emphasize handling over 100,000 requests per second (RPS) at peak, maintaining P99 checkout latency under 2 seconds, and achieving 99.95% uptime. Scalability is realized through stateless services and database sharding, while performance relies on aggressive caching for reads. Strong consistency is enforced only for seat status updates to guarantee zero oversells, whereas event metadata allows eventual consistency for faster reads. Maintainability is supported by independent service deployments and clear boundaries.

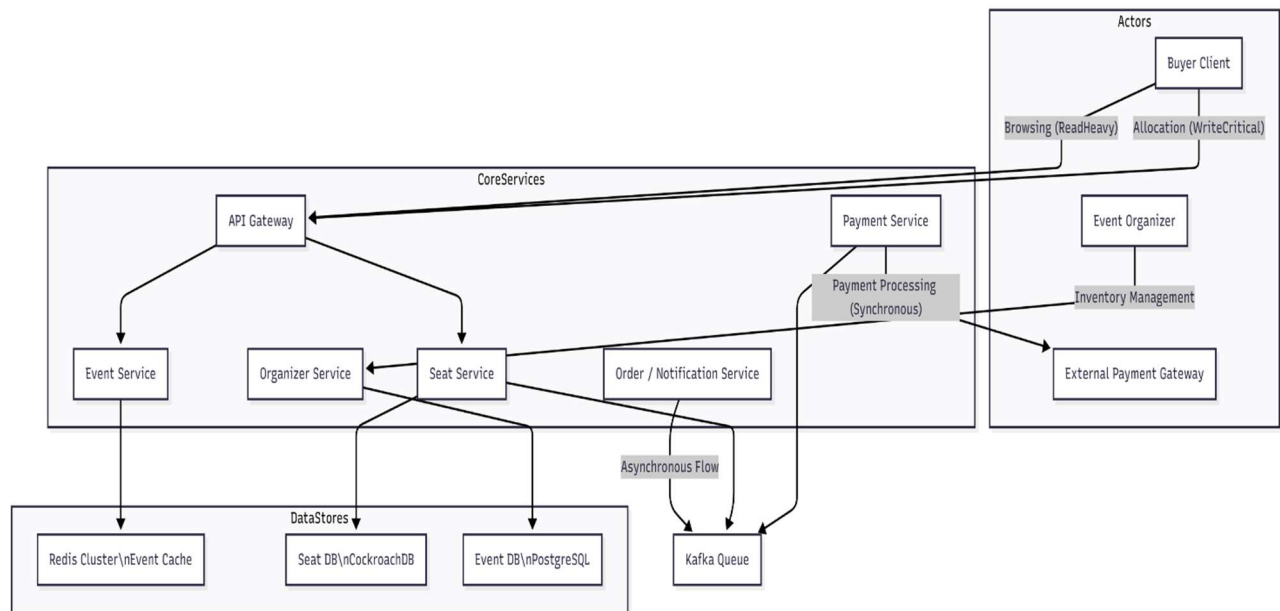
The system decomposes into key services: API Gateway for routing, Event Service for metadata, Seat Service for critical allocations, Payment Service for transactions, and Order/Notification Service for post-commit actions. Interactions include read-heavy browsing via caches and write-critical paths through the Seat DB. Stakeholders benefit from seamless buyer experiences, real-time organizer dashboards, idempotent payments, and support tools for manual interventions.

1.1 Design Goals and Non-Functional Requirements (NFRs)

The primary focus is on resilience and performance in the critical seat commitment path.

Table 1.1: Non-Functional Requirements (NFRs) Summary

Requirement	Target Metric	Rationale
Scalability	Peak RPS > 100,000	Necessary to absorb flash sale load. Achieved via stateless microservices and sharding.
Performance	P99 Checkout < 2s	Ensures a good user experience and low abandonment rate. Caching is mandatory.
Availability	99.95% Uptime	Requires multi-region deployment and robust failover strategy for all critical services.
Consistency (Seat)	Strong Consistency	Mandatory for seat status updates to guarantee zero oversells (ACID transactions).
Consistency (Listings)	Eventual Consistency	Allows aggressive caching of event and listing metadata for performance gain.
Maintainability	Modular Services	Independent deployments, clear domain boundaries, and simplified debugging.



1.2 System Decomposition and Service Map

Deployment Architecture (Kubernetes)

- All services deployed as stateless pods in Kubernetes (EKS/GKE).
- Seat Service: HPA (Horizontal Pod Autoscaler) min 10 → max 200 pods during flash sales.
- API Gateway: Ingress-NGINX with rate limiting annotations.
- Databases: CockroachDB multi-region cluster (3 AZs), Redis Sentinel for HA.

Component	Replicas (Normal)	Replicas (Peak)	CPU/Mem
Seat Service	5	50	2vCPU/4GB
Payment Service	3	10	1vCPU/2GB
Kafka Brokers	3	3	4vCPU/8GB

The following structure serves as the primary system context map, showing the decoupled service domains.

Figure 1.1: System Context and Microservices Interaction Map

Actor/System	Interaction Type	Core Service	Primary Technology
Buyer Client	Browsing (Read-Heavy)	API Gateway / Event Service	Event Cache (Redis Cluster)
Buyer Client	Allocation (Write-Critical)	API Gateway / Seat Service	Seat DB (CockroachDB)
Event Organizer	Inventory Management	Organizer Service	Event DB (PostgreSQL)
External Gateway	Payment Processing	Payment Service	Payment Gateway API (Synchronous)
Asynchronous Flow	Order Finalization	Order / Notification Service	Message Queue (Kafka)

1.3 Stakeholder Requirements

- **Buyers:** Seamless flow (browse, pick, pay), clear confirmation, quick loading times.
- **Event Organizers:** Real-time dashboard of sales, ability to manage seat sections and prices.
- **Payments/Finance:** Idempotency for all payment requests, detailed transaction logging, easy refund mechanism.
- **Support:** Ability to look up any order by user, ticket, or transaction ID, and manually release expired reservations.

Chapter 2

Data Model and Consistency

Seat state management forms the backbone of preventing oversells, with states transitioning strictly: from Available to Reserved via atomic checks, then to Sold on payment success, or back to Available on expiry or failure. A TTL-based reservation (e.g., 10 minutes) ensures seats don't lock indefinitely, cleaned asynchronously. Strong consistency is mandated here using database transactions with serializable isolation to block concurrent reserves.

The ER schema outlines core entities: Events store metadata in PostgreSQL; Seats and Reservations in CockroachDB for distributed consistency; Orders in a flexible store; IdempotencyKeys in Redis for retry safety; and Payment_Audit logs for traceability. Relationships tie seats to events via foreign keys, reservations to multiple seats, and orders to payments, enabling efficient queries while co-locating hot data.

2.1 Seat State Management and Consistency Model

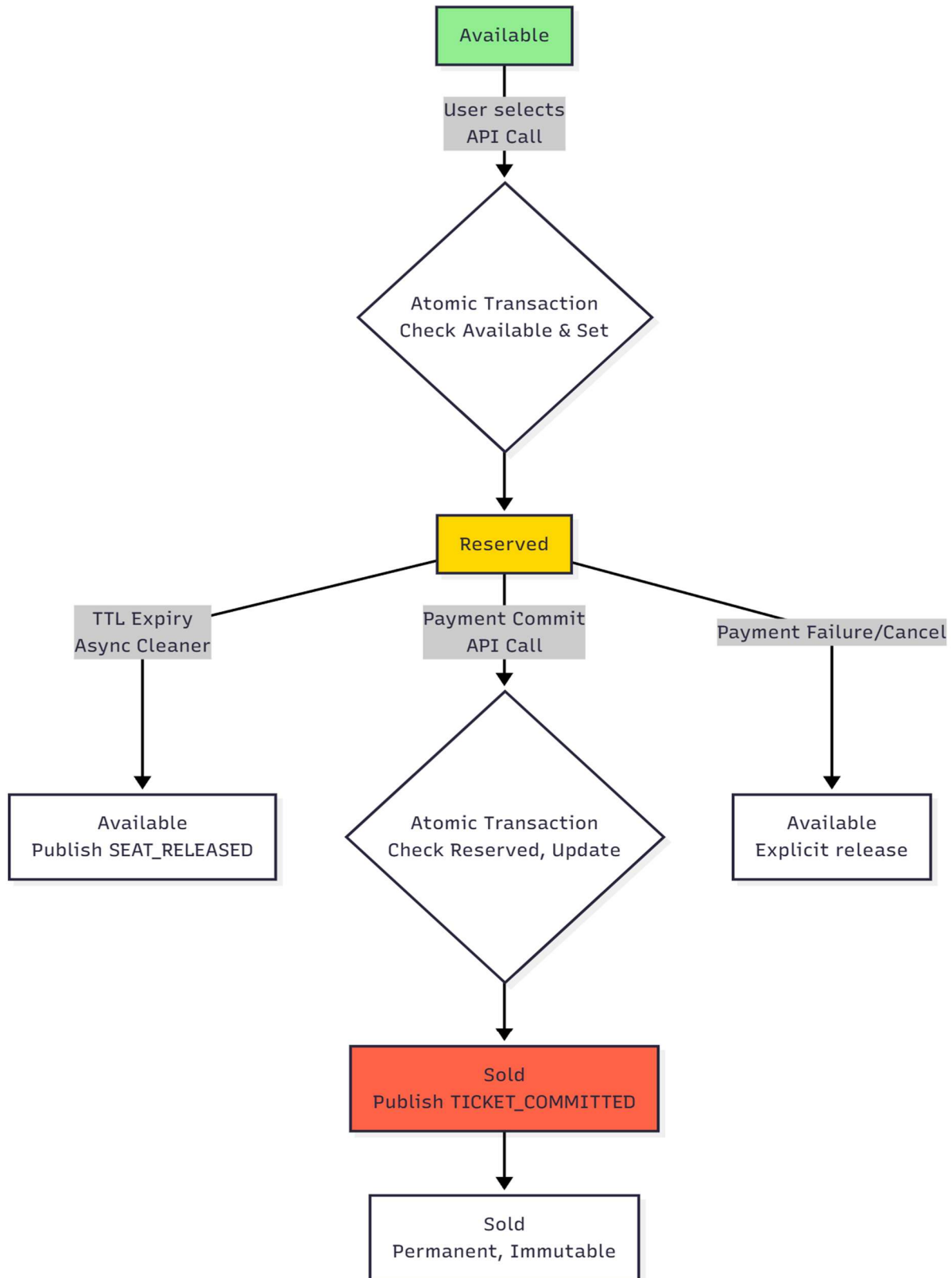
The central challenge is managing the state of a Seat during high contention. Strong consistency is enforced via database transactions in the Seat Service.

2.1.1 Seat State Transition Flowchart

This table represents the mandatory flow and state changes for a seat, ensuring strong consistency checks at each transactional step.

Figure 2.1: Critical Seat State Transitions Flowchart

Current State	Triggering Event	New State	Action / Constraint
Available	User selects (API Call)	Reserved	Atomic Transaction (Check Available & Set)
Reserved	TTL Expiry (Async Cleaner)	Available	Seat released for general sale. Publishes SEA
Reserved	Payment Commit (API Call)	Sold	Atomic Transaction (Check Reserved, Update)
Reserved	Payment Failure/Cancel	Available	Explicit release by Payment Service.
Sold	N/A	Sold	Permanent state. Immutable record.



2.1.2 Entity-Relationship (ER) Schema Outline

Table 2.1: Core Entity Definitions and Data Stores

Entity	Key Attributes	Primary Store
Event	event_id (PK), name, date, venue_id, total_seats	Event DB (PostgreSQL)
Seat	seat_id (PK), event_id (FK), section, status, reserved_until	Seat DB (CockroachDB)
Reservation	reservation_id (PK), user_id, seat_ids [], expiry_time	Seat DB (CockroachDB/Redis)

Table 2.1: Core Entity Definitions and Data Stores

Entity	Key Attributes	Primary Store
Order	order_id (PK), user_id, event_id, final_price, payment_tx_id	Order DB (NoSQL/PostgreSQL)
IdempotencyKey	key (PK), service (Payment/Seat), status (Pending/Complete)	Redis / Seat DB
Payment_Audit	tx_log_id (PK), idempotency_key, gateway_response, status	Payment DB (PostgreSQL)

Chapter 3

Scalability, Caching, and Transaction Flow

Capacity planning targets a 5-minute flash sale selling 20,000 seats, implying 200 commits/second baseline, but peaking at 50,000 reservation attempts RPS. Reads are offloaded to Redis with >95% hit ratio, reducing DB load; writes require ~400 TPS throughput via CockroachDB sharding.

Seat Service APIs are contention hotspots: /reserve locks seats atomically with idempotency; /commit finalizes to Sold and publishes events; /release handles failures; /seats GET serves cached maps. The checkout flow sequences cache reads for selection, atomic reserve, external payment, and atomic commit, ensuring low-latency reads and oversell-proof writes.

Payment integration uses client-generated idempotency keys, pre-checks in Redis/audit logs to skip re-execution, durable logging before gateway calls, and commit triggers only on success, maintaining financial consistency even under retries or failures.

3.1 Capacity Planning and Estimates

To achieve 100,000 RPS capacity during peak, we must offload read traffic and minimize synchronous writes to the database.

Table 3.1: Capacity Estimation for a 5-Minute Flash Sale

Metric	Value/Rate	Implication for System Design
Total Seats Sold	20,000	Requires 20,000 successful COMMIT transactions.
Successful Commits Rate	200 commits/sec	Puts a baseline load on the Seat DB's write capacity.
Reservation Attempts (Peak)	≈ 50,000 RPS	Handled by the API Gateway Rate Limiter and the Seat Cache (Redis).
Cache Hit Ratio (Read)	> 95%	Most seat availability checks must be served from cache, not the DB.
Required DB Write Throughput	≈ 400 tps	$2 \times \text{Commits/sec} + \text{Cleaner/Sec} + \text{Reserves/sec}$. Requires sharding or a distributed SQL solution (CockroachDB).

3.2 API Contracts (Seat Service)

These are the most critical, high-contention endpoints that must guarantee strong consistency.

Table 3.2: Critical Seat Service API Contracts

Endpoint	Method	Functionality & Requirements	Idempotency?
/v1/seats/reserve	POST	Requests temporary lock on seat_ids. Returns reservation_id and expiry_time. Must use a distributed transaction.	Mandatory
/v1/reservations/commit	POST	Finalizes the lock after payment success. Atomic: updates Reserved → Sold. Publishes a message to Kafka.	Mandatory
/v1/reservations/release	POST	Explicitly releases the lock (e.g., payment failure, user abandonment).	Optional
/v1/events/seats	GET	Retrieves current seat map and status. Served primarily from Redis Cache.	N/A

3.3 Checkout Read/Write Path Flowchart

This table serves as the primary transaction flow visualization, emphasizing the strong consistency checkpoints in sequence.

Figure 3.1: Sequential Checkout Flow Diagram: Reservation and Atomic Commit

Phase 1: Read/Selection (Cache)	Phase 2: Reserve (Atomic Lock)	Phase 3: Payment (External)	Phase 4: Commit (Atomic Write)
1. Client loads map (Cache read).	1. POST /reserve with Idempotency Key.	1. POST /v1/process with Idempotency Key.	1. POST /commit called by Payment Service.
2. Cache Miss → DB read.	2. Seat Service Idempotency Check.	2. Payment Service logs REQUESTED status to Audit DB.	2. Seat Service Idempotency Check.
3. Cache is populated.	3. DB Transaction Start (Available → Reserved).	3. Payment Service calls External Gateway.	3. DB Transaction Start (Reserved → Sold).
4. Client view is Eventual Consistent.	4. DB Transaction Commit (Strong Consistency).	4. Gateway returns TX_ID (Success).	4. Commit successful, seat is permanent Sold.
Goal: Low-Latency Read	Goal: Seat Lock / Oversell Prevention	Goal: Secure Financial Exchange	Goal: Strong Consistency Final State

3.4 Payment Service Integration and Idempotency

The Payment Service acts as an orchestrator, handling synchronous communication with the external gateway and ensuring financial consistency.

1. **Idempotency Key Generation:** The client generates a unique Idempotency Key for the entire checkout flow and passes it to the Payment Service.
2. **Pre-Execution Check:** Upon receiving a POST /v1/process request, the Payment Service first checks its audit log/Redis. If the key is found and marked COMPLETE, it skips execution and returns the original result.
3. **Audit Logging:** Every payment attempt is logged in a separate, durable Payment_Audit table (e.g., in PostgreSQL) before calling the external gateway. This log includes the Idempotency Key, reservation_id, and the initial status (REQUESTED).
4. **Commit Trigger:** Only upon receiving a final success status from the external gateway does the Payment Service call the /reservations/commit endpoint, passing the payment_tx_id.

Chapter 4

Data Sharding and Scaling

The Seat DB scales horizontally with CockroachDB to handle write contention, supporting multi-region replication for low-latency transactions. Sharding by `event_id` routes all operations for an event to one shard, minimizing cross-shard latency and distributed commits.

For hot shards in flash sales, pre-splitting ranges by seat sections distributes load across nodes preemptively. Read offloading to Redis ensures 95% of availability checks bypass the DB, alleviating pressure on contended shards during peaks.

4.1 Seat Database Scaling Strategy

Given the Seat Service is the highest-contention write service, horizontal scaling of its database is mandatory. We selected CockroachDB due to its distributed SQL capabilities, multi-region replication, and ability to handle strongly consistent, low-latency transactions.

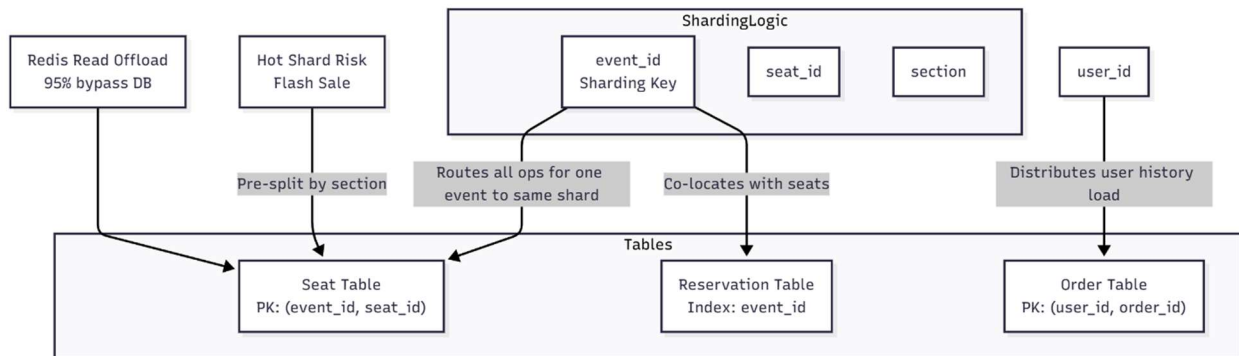
4.1.1 Sharding Key Selection

Choosing the correct sharding key is vital to distributing load and minimizing cross-shard transactions. Since contention is concentrated per event, sharding must be done by the `event_id`.

Table 4.1: Sharding Key Strategy for CockroachDB

Sharding in the Seat DB uses `event_id` as the primary key to route all operations (reads/writes) for a single event to one shard, minimizing cross-shard transactions and latency. The Seat table has a composite primary index on (`event_id`, `seat_id`), ensuring co-location of event-specific seats. Reservations are indexed by `event_id` to keep data with seats and avoid distributed joins. For flash sales, pre-splitting ranges by seat sections distributes hot event load across nodes. Orders shard by `user_id` to balance user history queries independently.

Table	Sharding Key / Index	Rationale
Seat	Primary Index on (<code>event_id</code> , <code>seat_id</code>)	Allows all read/write operations for a single event to be routed to the same shard (minimizing distributed latency).
Reservation	Index on <code>event_id</code>	Reservations are always tied to a single event; keeps reservation data co-located with the seat data.
Order	Primary Index on (<code>user_id</code> , <code>order_id</code>)	Orders are primarily queried by the user. Sharding by user distributes lookup load for history.



4.1.2 Hot Shard Management (Flash Sale)

The risk of a "hot shard" (where all traffic focuses on a single event's data) is mitigated by:

1. **Pre-splitting Ranges**: For highly anticipated events, the Seat data can be manually pre-split into multiple ranges within CockroachDB based on seat sections, distributing the load across multiple physical nodes before the sale starts.
2. **Read Offloading**: The use of Redis for the seating map view ensures that 95% of read traffic bypasses the database entirely, drastically reducing the load on the hot shard.

Chapter 5

Asynchronous Processing and Queuing

Kafka serves as the event bus for decoupling high-throughput seat operations from slower downstream processes like order creation and notifications. Producers publish post-transaction events (e.g., SEAT_RESERVED after reserve), consumed by invalidators, monitors, or fulfillment services.

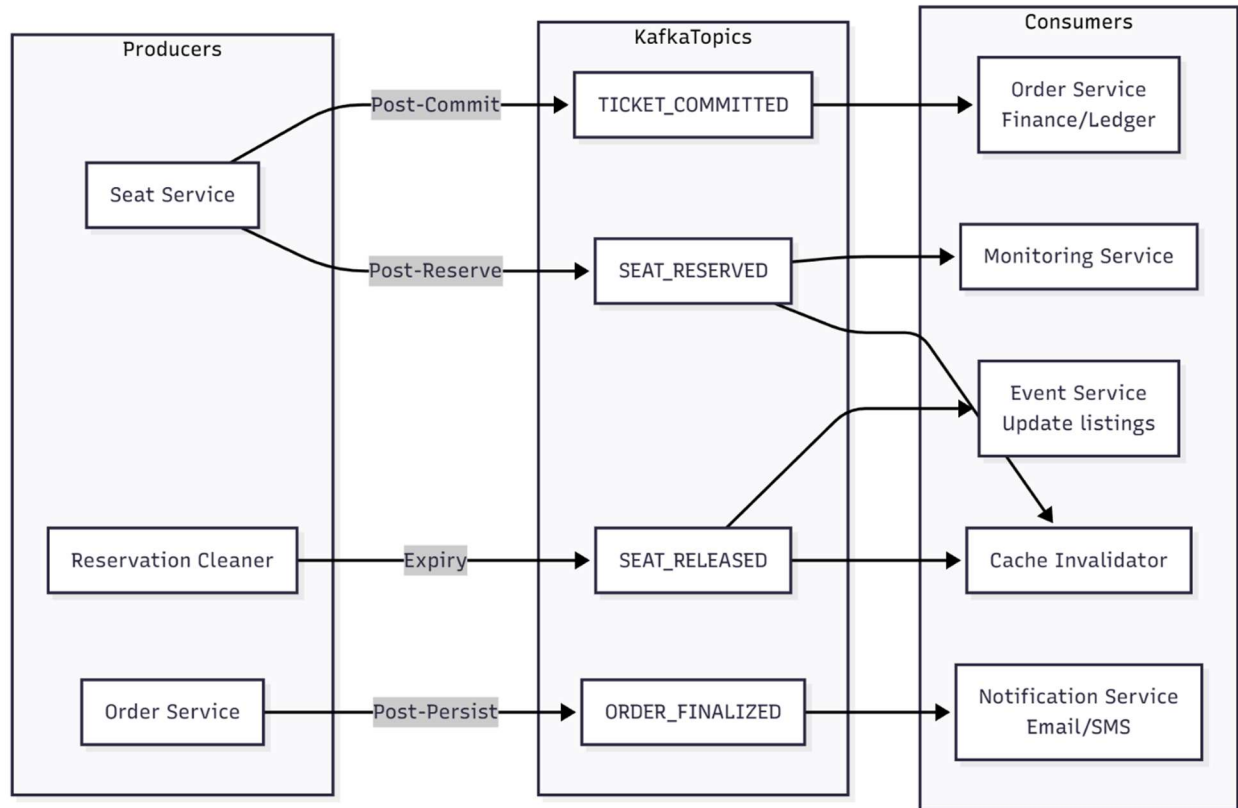
Reservation cleanup runs as a periodic worker querying expired seats, atomically releasing them, and publishing SEAT_RELEASED events to update caches and listings, preventing indefinite locks without blocking the critical path.

5.1 Queuing Strategy (Kafka)

Kafka is used as the backbone for inter-service communication and load leveling, ensuring the high-throughput Seat Service does not wait for slower services (Order fulfillment, notifications).

Table 5.1: Asynchronous Event Handling via Kafka Topics and Flow

Event Topic	Producer Service	Consumer Services
TICKET_COMMITTED	Seat Service (Post-DB Commit)	Order Service, Finance/Ledger Service
SEAT_RESERVED	Seat Service (Post-DB Reserve)	Cache Invalidator, Monitoring Service
SEAT_RELEASED	Reservation Cleaner / Seat Service	Cache Invalidator, Event Service (Update listings count)
ORDER_FINALIZED	Order Service (Post-Persistence)	Notification Service (Email/SMS)



5.2 Reservation Cleanup Strategy

A dedicated, scheduled worker is required to clean up abandoned reservations.

- **WorkerJob:** The Reservation Cleaner is a background process that runs every minute.
- **Query:** Selects all Seat records where status = Reserved AND reserved_until < NOW().
- **Action:** For each expired seat, it executes an atomic transaction setting status = Available and publishing a SEAT_RELEASED event to Kafka.

Chapter 6

Resilience and Failure Management

The resilience of the Scalable Event Ticketing & Seat Allocation System is paramount, given the financial and reputational risks associated with overselling seats or losing transactions during high-concurrency flash sales. The design incorporates multiple layers of defense: transactional guarantees for critical paths, idempotent operations to handle retries safely, automated failovers, and proactive throttling. By isolating failures and providing recovery mechanisms, the system maintains strong consistency for seat states while allowing graceful degradation for non-critical components.

Oversell prevention is achieved through rigorous lock management in the Seat Service. Both the reservation (`/reserve`) and commitment (`/commit`) operations are executed as single, atomic database transactions in CockroachDB. These transactions use a serializable isolation level, which detects and aborts any conflicting concurrent attempts to modify the same seat row. For instance, if two users attempt to reserve the same seat simultaneously, one transaction will succeed in updating the status from Available to Reserved, while the other will fail with a serialization error and retry (client-side). Explicit row-level locking can be added via `SELECT FOR UPDATE` in queries to further serialize access. This ensures that under no circumstances can a seat transition to Sold without a prior valid Reserved state tied to a successful payment.

To manage locks effectively and prevent deadlocks during contention, a short TTL (e.g., 10 minutes) is enforced on reservations via the `reserved_until` timestamp. If a user abandons the checkout, the seat is not held indefinitely. This TTL is checked atomically during the commit phase—if expired, the commit fails, and the client must re-reserve. Idempotency keys play a crucial role here: generated uniquely per checkout flow (e.g., UUIDv4), they are stored in Redis with a matching TTL. On retry, the Seat Service queries the key; if `COMPLETE`, it returns the prior response without re-executing the transaction, avoiding duplicate locks or charges.

6.1 Oversell Prevention and Lock Management

Oversells are prevented by ensuring the `RESERVE` and `COMMIT` stages are single, atomic database transactions. Any concurrent attempt to reserve or commit the same seat is blocked by the transactional isolation level (e.g., `SERIALIZABLE` or `READ COMMITTED` with explicit locking).

6.2 Failure Scenarios and Mitigation

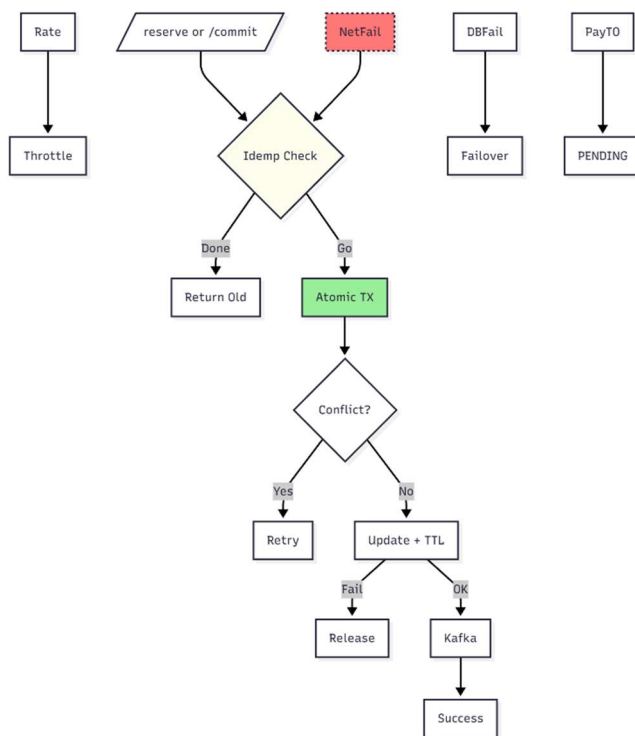
Table 6.1: Key Failure Scenarios and Retries Strategy

Failure Type	Scenario	Mitigation/Recovery
Idempotency	Client retries /commit due to network error.	The Seat Service checks the Idempotency Key before execution. If found and COMPLETE, the original success response is returned without re-executing. Prevents double-charge/double-commit.
Distributed DB	CockroachDB Node Failure	CockroachDB (or any distributed SQL) automatically fails over to a replica node in a different availability zone (AZ). Strong consistency is maintained across the failover.
Payment Timeout	Gateway takes > 10s to respond.	Payment Service returns PENDING status to client. Client is instructed to check back later. If the payment ultimately fails, the Seat Service releases the lock.
Queue Backlog	Kafka topic is overwhelmed during flash sale.	The critical path is protected (commit succeeded). Order creation and notifications are delayed but processed eventually once the Order Service catches up. Uses backpressure.
Data Corruption	Failed commit leads to charged user and reserved seat.	Manual support intervention uses Payment_Audit log to force Sold status or initiate refund.

6.3 Rate Limiting Implementation

Rate limiting is essential for protecting the Seat Service from denial-of-service (DoS) attacks or aggressive bot traffic.

- **Location:** API Gateway (Edge Service).
- **Method:** Sliding window algorithm using a fast Redis counter.
- **Limit:** 30 requests per minute per unique user_id/IP on read operations (/seats), and 5 requests per minute on write operations (/reserve).
- **Response:** When limits are exceeded, a HTTP 429 Too Many Requests status is returned.



Chapter 7

Monitoring and Observability

Observability ensures uptime and bottleneck detection via Prometheus metrics like P99 commit latency (alert >1.5s), DB retry rates (>5%), Kafka lag (>5 min), error rates (>0.1%), and cleanup rates. Grafana dashboards visualize these for flash sale oversight.

Jaeger traces span the checkout path—from gateway to DB transactions, payment calls, and Kafka publishes—pinpointing latency contributors within the 2-second budget, especially external dependencies.

7.1 Key Observability Targets

Effective monitoring is crucial for maintaining the 99.95% uptime requirement and diagnosing bottlenecks during flash sales.

7.1.1 Metrics and Alerting (Prometheus/Grafana)

Monitoring the system's performance and health during normal operations and extreme flash sale loads is critical to meeting the 99.95% uptime NFR and rapidly identifying bottlenecks in the checkout path. Prometheus scrapes time-series metrics from instrumented services (using client libraries like prom-client for Node.js or PrometheusExporter for Java), storing them in a centralized TSDB. Grafana provides interactive dashboards with panels for real-time graphs, heatmaps, and tables, allowing operators to visualize trends such as latency spikes or contention buildup. Alerts are configured via Prometheus Alertmanager, routing notifications to Slack, PagerDuty, or email with severity levels (warning/critical) and runbooks for resolution.

Table 7.1: Critical Metrics for Performance Monitoring

Metric	Service/Layer	Alerting Threshold/Focus
P99 Latency (/commit)	Seat Service	Alert if > 1.5s (Warning), > 1.8s (Critical). Directly impacts NFR target.
DB Transaction Contention	Seat DB (CockroachDB)	Alert if transaction retry rate exceeds 5% (Indicates heavy load or lock contention).
Kafka Consumer Lag	All Consumers (Order/Notification)	Alert if lag exceeds 5 minutes for TICKET_COMMITTED (Indicates fulfillment backlog).
Error Rates (HTTP 5xx)	API Gateway / All Services	Alert if error rate exceeds 0.1% across the system.
Reservation Cleanup Rate	TTL Reservation Cleaner	Alert if the rate drops to zero (Indicates seats are not being released).

7.2 Distributed Tracing (Jaeger)

Distributed tracing allows for detailed analysis of the P99 checkout path.

- **Trace Span Focus:** Each critical step in the checkout flow (API Gateway, Seat Service DB Transaction, Payment Gateway call, Kafka message publication) is instrumented as a separate span.
- **Purpose:** To visualize where the 2s latency budget is spent, especially identifying slow spans in external calls (Payment Gateway) or database operations (CockroachDB transaction commit time).

Chapter 8

Future Work and Maintenance

Maintainability leverages modular deployments for independent scaling (e.g., ramp Seat Service pre-sale), config tools like Kubernetes ConfigMaps, and chaos testing to validate idempotency and failovers.

Future enhancements include dynamic pricing based on demand, secure ticket transfers via ledgers, and anti-bot integrations to further protect the gateway.

How to Paste: Copy each chapter block → Word: Go to chapter heading → Ctrl+V below tables/figures. Bold subheadings if needed. Update page numbers/TOC after.

8.1 Maintainability and Operations

- **Modular Services:** Each service (Event, Seat, Order, Payment) is deployed independently, allowing vertical or horizontal scaling of specific components (e.g., scaling the Seat Service aggressively before a flash sale).
- **Configuration Management:** Use standardized tools (e.g., Consul or Kubernetes ConfigMaps) for dynamic configuration updates without requiring service restarts.
- **Chaos Testing:** Introduce failures (e.g., random network delays, service crashes) to validate the Idempotency Key and failover logic under controlled conditions.

8.2 Future Enhancements

- **Dynamic Pricing Engine:** A separate service integrating market data and inventory levels to adjust seat prices in real-time.
- **Ticket Transfer/Resale:** Implementation of a smart contract or secure ledger to manage the immutable transfer of tickets between verified users.
- **Anti-Bot Integration:** Integration with specialized services to filter out non-human traffic before it hits the API Gateway.

