High-performance ticket reservation system for ultra-large-scale events

Solution/Architecture design case study by Tamás Horváth (txttw)
Last updated: 2025-09-15

Core problem

Managing an **instantaneous**, **massive surge** of **traffic** the moment registration opens, while **ensuring fairness**, **consistency**, **performance** (acceptable response times) and **user-friendly** experience.

In this document I am going to take into consideration UX, architectural, data model, backend and frontend related design decisions.

Note: *This is not a full system/architectural design. I will only discuss critical components related to the most challenging nature of such system: mixture of high performance, great UX, fairness under extreme load.*

Business Philosophies

Fairness: Provide equal opportunity for all users to secure a ticket. The system must prevent advantages for users based on device performance, location, network latency, or brute-force scripting.

Transparency: Users must receive clear, timely feedback on their status (e.g., "You are in a queue," "Your estimated wait time", "Your have 10 mins to finish reservation", "Sorry, tickets are sold out", etc.)

Integrity: The system must prevent overselling (selling more tickets than available seats), duplicate bookings, and race conditions.

Performance: The system must respond in a predictable, consistent way even during high load.

Ease of Use (UX): The user interface must be simple, intuitive, and resilient. It should manage user expectations during high-stress periods (e.g. waiting in queue, limited time to select seats and finish payment) and guide them to follow best practices (e.g. do not refresh the page – it refreshes the data you need, auto assign for higher chance, unfair automation will be penalised, etc.)

Requirements

In this design I will **omit** requirements that are not related to the large-scale and high-load nature of the system (or solved by other system design decisions), such as Registration, Authentication, Event catalogue, Ticket delivery (PDF or QR), Notification (email, sms), Payment processing, etc

I use seat reservation for easier understanding but the system handles reservable units (e.g. seat, private section, VIP table, etc.) After reservation it can map actual tickets to those units (e.g. multiple tickets for a private section).

Functional Requirements

Multi scale visualisation: To handle up to ~100K seats from a UI/UX perspective we need an interactive multi scale section/seat map for easy and efficient browsing.

High-concurrency reservation process:

- **Queue management:** Place users into a fair, managed queue upon event access.
- **Seat selection:** Allow users to select one or more (limited number to avoid speculative purchase) available seats from the map or auto allocate for a higher chance during high demand.
- **Seat holding:** Temporarily hold (lock) selected seats for a limited time (e.g., 5-10 minutes).
- **Checkout:** Keep holds while the user is redirected (5-7 mins) to a payment processor.
- **Reservation finalisation and feedback:** Finalise the reservation and generate a unique reservation ID. Show the ID to the user as immediate confirmation. (email notification can take more time)
- **Delivery and notification**: *Omitted from this discussion because it is tied to the seat count (e.g. total 100k) not to RPS (e.g. 1M/s), can be queued and delivered in an acceptable rate (Communicate clearly it is not immediate)*

Non-Functional Requirements

Scalability & Load

- Handle ≥ 1,000,000 RPS for browsing and seat map data.
- Optimise DB writes to not be the reservation bottleneck.
- Support an active user base of millions of potential buyers (queue size).

Performance & Latency

- Queue page response: typical < 50 ms for a user to enter the queue (abuse detection might increase to 150-500ms for a subset of requests)
- Seat map data (cached): < 50 ms response time
- Seat hold action: < 2 seconds from user click to confirmation
- System must remain responsive under peak load

Availability & Reliability:

- 99.99% (four nines) availability for the core queue and reservation API services during the sale period
- Zero data loss for finalised transactions (reservations and payments).
- Consistency & Integrity: Eventual Consistency is acceptable for seat map visibility during the rush.
- Strong consistency is required for seat reservation to prevent double-booking

Fairness

- The queue system must be implemented to provide a statistically fair outcome
- Mechanism to detect and mitigate bot advantages are mandatory

Security

- Protect user data (PII) and payment information.
- Implement robust DDoS mitigation
- Implement bot/automation/cheating detection (e.g., CAPTCHA, behavioural analysis) before accepting reservation

To satisfy the availability, performance and durability requirements I design the system with AWS services.

Where is the bottleneck?

DB Writes: Based on current AWS default service limits, a DynamoDB table with appropriate partitioning can scale to handle 40k WCUs (for global tables it can decrease as replication also needs write capacity). As later discussed, we heavily rely on caching therefore we do not need multi-region replication (global tables) for reads. For writes we can accept a 50-300 ms latency + 20-50 ms conditional write (specified < 2s for reservation actions under requirements). Single region table also ensures strong consistency that strictly required to avoid duplicate bookings. To hold a seat we use a conditional write that takes 2-4 WCUs so we can handle ~10k reservation requests per sec in a local table. If we try to predict the peak write OPS within a reservation window it is likely way less than 1/s so purely from a write perspective our max write concurrency >10k.

User experience (UX): The concurrency we just discussed is in the 10k magnitudes. But if we allow 10k concurrent reservation sessions even for a 100k stadium we probably have a terrible UX. Seat popularity is not uniform and the competition for hot seats would be way too high for a predictable and fair user experience.. At the end, the majority of our customers would be unsatisfied.

DB Reads can be heavily cached with strategic TTL and invalidation (design shown later). With controlled concurrency, seat map change rate is lower, therefore users can tolerate not completely real-time availability updates. As the reservation view is partitioned around sections we can set shorter TTL for hot sections and longer for not so popular ones. Cached reads can handle ≥ 1M RPS with ≥p95 low latency and low response time (<50ms). Well designed DynamoDB tables can manage sub second TTL / cache invalidations for hot sections for an almost real time experience.

From a UX perspective we have identified that the number of concurrent users is the bottleneck. We need to determine an acceptable concurrency for our reservation use case that provides a consistent, predictable reservation flow with urgency but not chaos and panic. I am not going to propose a method to choose this number because it is not a one size fits all. It should consider historical data, business requirements, may depend on the stadium/venue, event type, user behaviour etc.

Once we have that number we can calculate the acceptance rate by Little's Law

Little's Law $L = \lambda W$, states the average number of items in a stable system (L) is equal to the average arrival rate (λ) multiplied by the average time an item spends in the system (W)

In our case:
L - average number of concurrent reservation sessions
W - the average reservation session time.

If the arrival rate λ is enforced, the system maintains an average L concurrent sessions with an average session time W. For example, to have an average 1000 concurrent sessions in our system with 5 min average session time we can accept ~3 sessions per sec.

Our system will utilise a more dynamic approach to accept sessions but this is a great example to show the slow rate we can accept users not because of system limitations but because of UX (10 min session and max concurrency).

Just letting users to try and rejecting them would result in a terrible and unfair (who can send more requests to increase the acceptance chance) experience. At this scale we need to design a statistically fair waiting room for far better UX.

UX design

I am not attempting a UX design just outlining key UX decisions that shape the system architecture

Event landing page: Does not change frequently, cacheable (CloudFront / CDN) with long TTL and invalidated on updates using Change Data Capture (CDC) events. (e.g. DynamoDB Stream, or similar concept for RDBMS)

Reservation view can be part of the event landing page or a separate page. If we decide to make it part of a SSR (Server Side Render) Landing page we should use an RSC (React Server Component) for the landing page (rarely changing content) and the reservation view (Client Component) should be inside a Suspense. This way the skeleton can be served from a CDN with a long TTL (rarely changing) and the client component can fetch availability and seat-maps from an API cached with short TTL.

The reservation view with prices (without reserve action) should be available long before the sale period to let users plan their seating and accommodate to the UI. From historical data, the UI can show potential hot sections/seats and warn users to make an alternative plan in case their preferred seat already sold out.

Reservation multi scale view

Goal: The detailed seat map is only displayed for one sector/block/etc. This reduces the overwhelming amount of displayed seats compared to showing the entire stadium map (potentially 100k seats). This behaviour also reduces the read load on the system.

Section level view

- Large scale overview of the stadium/venue (e.g. an embedded svg with attached event handlers for hover and click)
- Default view
- Each section is identified clearly by a name/code
- During pre-sale it can show hot sections, prices, etc.

- During ticket sale either color coded by availability or an approximated availability (e.g., "Good availability", "Few left", "Sold out") on hover.
- Hovering over a section shows more information: Price, etc.
- **Action**: Clicking a section takes the user to intermediary view (if exists) or directly to the section's seat map

Intermediary view (optional)

- For large sections it is possible to contain multiple blocks, levels, etc.
- This view behaves the same way as the section level view
- **Action**: Clicking a block/level/etc. takes the user to the detailed seat map of the selected block

Detailed seat map

- Clear, zoom-able map showing individual seats (e.g. an embedded svg with attached event handlers for hover and click)
- Seat status (available, taken, selected/hold/lock/etc.) is color coded or displayed with a different icon.
- Taken seats are disabled. Available seats are selectable.
- On hover, seat price and other info can be displayed
- Auto reservation actions: System can offer auto reservation options (e.g. "Best available", "N consecutive", etc.). After successful lock the UI highlights the auto reserved (hold/locked) seats on the map.
- **Action**: Clicking on the seat will send a hold/lock request to the server. Alternatively users can select multiple seats and send a batch request. System either holds all the requested seats or nothing. The UI highlights the locked seats. Users have a limited time frame (reservation session) to select seats.

Reservation flow (user journey)

1. **Landing page (pre-sale)**: Event info, countdown timer, pre-sale reservation view with prices, and a prominent "Prepare for sale" message that links to a reservation best practices page (e.g., "Register/Log in early", "Check seat map before reservation", "Prepare your payment info", etc.)
2. **Landing page (during-sale)**: Event info, "How to use / rules info", Multi scale reservation view where users can check availability, price, etc. and a CTA (e.g. "Begin reservation") that leads them to the waiting room.
3. **Waiting room**: When users click "Begin reservation" they enter the waiting room. Nobody is rejected from the waiting room. A waiting room UI is shown. The waiting room token is short lived (e.g. 10 min but can be extended in long queues) and stored in the browser/app. Users can restart the browser/app but can not change devices or come back later without requesting a new token. The waiting room service provides info about the approximate waiting time and controls admission polling with a backoff time (for the client when to send

the next entry attempt to reservation). The UI will respect the backoff time and sends the next attempt accordingly. Refreshing the page won't send entry attempts (discourage this action). *Because of this behaviour, the service can differentiate between legit UI based app users from bots/scripting users sending mass requests looking for an advantage.*

4. **Reservation session**: When the user granted access, the reservation session begins. User sees the multi-scale reservation view now with active reserve functionality and a countdown timer indicating the remaining time (e.g. 10 min). During the reservation session they can hold/release multiple seats with a limit for fairness. The session is stored on the server so closing the browser/app or changing the device does not have any negative effect. However the timer does not stop while the user is offline. User has to start the checkout process before the time elapses otherwise their hold seats are automatically released.
5. **Checkout**: If the user begins the checkout phase in time, seats are locked for another time frame (e. g, 5-8 min). During this time, payment has to be confirmed by the payment processor for the reservation to be finalised. The UI shows a payment transaction ID and a reservation ID for immediate confirmation and informs the user about a potential delay of email or in app notification.
   If the payment is unsuccessful or abandoned, seat holds are released either directly or automatically by expiring the lock.

Waiting room

Waiting room is essentially a queue.

Requirements

- Handle ultra-high initial load (1M RPS), millions of users requesting to enter the queue.
- Decrease the load by letting the client (frontend) govern the admission process in a controlled fashion. Make refreshes useless and cheap to serve.
- Even with controlled admission attempts the queue and admittance service should be designed to be able to handle 1M RPS
- Handle very long queue size (>10M tokens)
- Admittance shall satisfy statistical outcome fairness
- Prevent cheating and catch bad actors (best effort)

When the supply and demand is close to each other FIFO can achieve the required fairness.

However, if the demand is magnitudes higher than supply, FIFO amplifies noise (geographic location, latency, operational inconsistency, time synchronisation, etc.) and favours cheating (bots, scripting). Consider 1M RPS, even 100 ms latency would mean 100k difference in queue position. Pure FIFO is deeply unfair and exploitable with these conditions.

The other end of the spectrum is pure randomness, like pre-registration (e.g. tied to verified phone number) and a random lottery draw. This approach is way more fair than pure FIFO but does not back fans who are willing to spend hours to wait in line to get a ticket.

Therefore, I recommend the combination of a lottery ticket and probabilistic FIFO (combining the advantages of the two worlds) with a curve that is tuned (business decision) to balance FIFO and luck. The curve also has a hyper-parameter that affects the acceptance rate so we can control the concurrency limit for good UX.

To take into account the performance requirements (>10 M queue size, >1 M RPS), implementing a regular server side queue is not feasible. Instead we are going to give a waiting room token to the user. Data only persisted in the token that are stored in the browser/app (client side queue). We use cryptographic signatures to ensure token validity and prevent forging.

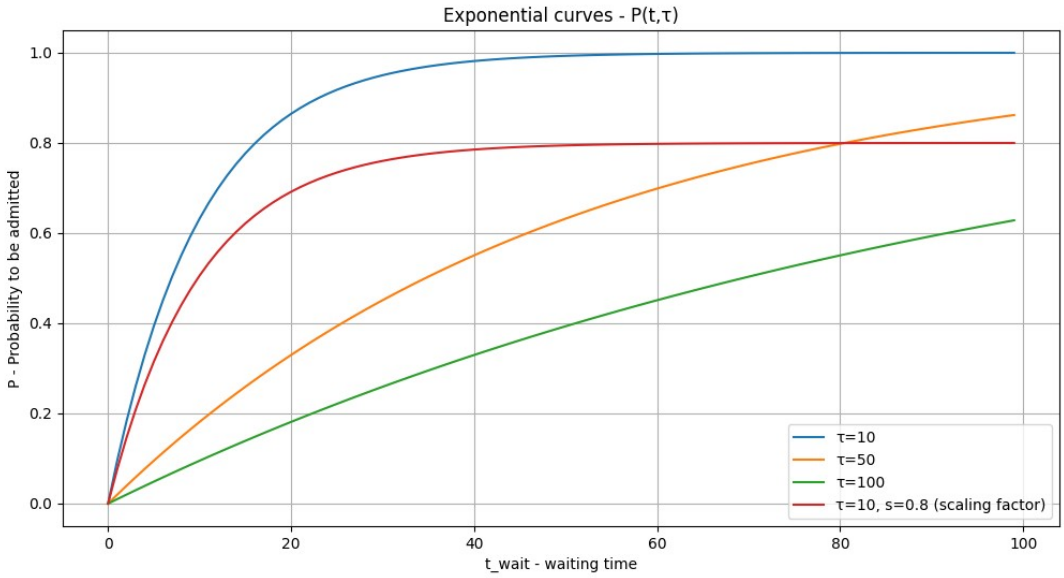
How it works?

At token creation, random data added to the token (represents lottery ticket). At admission a cryptographic process with a secret (e.g. HMAC_SHA256) uniformly converts the data from the token to a number between 0 and 1. As long as the secret is protected and the mapping is close to uniform the client is unable to guess how good their ticket is.

To create a probabilistic FIFO we need a curve that monotonically increases with respect to time $P(t, \tau)$ – where t is time, τ is the hyper-parameter to control admission rate. So the longer the user is waiting has a higher probability to get in.

At admission, we have a number (v) between 0,1 from the lottery ticket and we also have a probability $P(t, \tau)$ from the curve. If $v \geq P(t, \tau)$ user gets admitted otherwise stays in the queue (waiting room).

Here are a few example curves with different hyper-parameters (τ):



Lottery ticket generation (python example)

Unrelated code omitted for simplicity, token expiry, backoff time, etc.

```
import secrets
import base64
from datetime import datetime
```

```
lottery_ticket = secrets.token_bytes(16)
# payload is the token payload
payload = {}
payload[‘lottery’] = base64.b64encode(lottery_ticket).decode('UTF-8')
# payload[‘iat’] – issued at
payload[‘iat’] = datetime.now().isoformat()
```

Lottery number derivation from ticket (python example)

```
import hmac
import hashlib
import secrets
import struct
# secret_key stored in KV and rotated frequently
# payload is the token payload
signature = hmac.new(secret_key, payload[‘lottery’].encode('UTF-8'), hashlib.sha256)
# Truncated but small number of collisions are accepted
lottery_number = int.from_bytes(signature.digest()[:8]) / 2**64
```

Admission decision (python example)

Using both the probability curve and the lottery number. Unrelated code omitted for simplicity.

```
from datetime import datetime
import math
# τ - control parameter to manage max concurrency, comes from KV
now = datetime.now()
# payload['iat'] - issued
wait_time = now – datetime.fromisoformat(payload['iat'])
decision = False
if wait_time > 0:
   # P – probability depending on wait time and control param τ
   P = 1 - math.exp(-wait_time / τ)
   # lottery_number calculated above
   decision = lottery_number >= P
```

To handle the very high load requirements we need to carefully design the two services.

- Waiting room service (token creation endpoint)
- Admission service (admission endpoint)

At this scale, multi-region coverage and auto-scaling is a must. I recommend edge computing. AWS CloudFront Functions is the best candidate for this task.

Edge availability makes it excellent for these services as they directly face global users and the load intrinsically divided between edge locations. Provides low latency and fast response time.

CloudFront Functions are serverless therefore there is no need to manage infra, auto-scales automatically.

It has 1 ms CPU limit: Auth JWT verification, waiting room token signing, lottery number derivation utilises HMAC_SHA256. Challenge verification also uses SHA256. 1 ms is acceptable, even an older CPU can generate >500 hashes/ms, AWS uses native, highly optimised crypto libraries.

Functions can not access network and interact with other AWS services only with an attached read-only KV store. As already discussed, these services do not persist any data in DB/Redis/etc. only create/update the waiting room token and sign it. Keys for signing/verifying, hyper-parameter (τ) for probabilistic FIFO can be stored in the read-only KV store.

KV store update (outside of CloudFront Functions) is eventually consistent and propagation to edge locations can take several seconds. For signing key rotation this is acceptable. We also use KV for τ to adjust admittance rate dynamically. As the average reservation session is 5-10 mins, updating the control parameter in every ≥10 seconds is acceptable.

From a cost point of view, CloudFront Functions are designed to face high traffic directly, they are way cheaper than Lambda@Edge. When I write this document it has a fixed rate $0.10 / million invocations. Considering that the waiting room client (frontend) uses controlled polling (backoff time) and does not send requests on refresh it is an acceptable cost.

Waiting room token

Header fields:

- typ: Token type
- alg: (Algorithm) HMAC_SHA256
- kid: (keyId) It can be used for key rotation. e.g. older keys are kept for a while to sign/verify older versions

Payload fields:

- id: (Token ID) A unique identifier (e.g. UUID v7) for the token that is registered on admission. It prevents token reuse.
- iat: (Issued at) The time the token was issued
- exp: (Expires at) The time after that the token is no longer valid
- sub: (Subject) The principal that is the subject of the token (e.g. userId)
- lot: (Lottery ticket) 16 bytes uniform random data
- boff: (Backoff time) For controlled polling. The client must respect this time before retries the admission
- ch: (Challenge) Uniform random data. Protects the token creation with a PoW

The signature is created by combining the encoded header, the encoded payload, a secret with the algorithm specified in the header. As long as the secret key is protected a bad actor can not change the content of the token without detection (signature mismatch). Frequent key rotation is recommended.

Admission process

Prerequisites: User needs to be signed in and the client sends a JWT either as an Authorization header or a HTTP cookie with all requests

1. User clicks the “Begin reservation” button

2. Client (frontend) checks (local storage) if it already has a valid (payload.exp < now) waiting token. If not, requests a new one by sending a request to the **token creation endpoint** and stores the returned token in local storage (or set as a HTTP only cookie by the server, but some parts of the payload should be returned because the client needs it). Token is tied to userId.
3. Client displays a waiting room UI to the user with an approximate wait time if the server response had one
4. If client finds a challenge (e.g. PoW) in the payload solves and stores the solution in local storage. Challenge protects token creation so won't change for this token
5. Client sets a timeout to the payload.boff value. This is a minimum time it has to wait before attempts an admission.
6. After timeout elapsed, client attempts an admission by sending a request to the **admission endpoint** including the token and the solution for the challenge.
7. Server validates the token and the solution and decides if admitted or rejected. If rejected the server updates the payload.boff time, possibly payload.exp, regenerates the signature and returns the updated token. It can return an updated "approximated wait time" to inform the user
8. If rejected, client stores the new token, adjusts the UI with the new "approximated wait time", sets a timeout with the new backoff time and repeat until admitted or the token expires.
9. When the client admitted, the admission service routes the request to the **reservation session creation endpoint**
10. The reservation service verifies the token and revalidates the admission criteria. This service is stateful, it can check more criteria to accept or reject the token (e.g. user behaviour to prevent cheating, current concurrency, etc.). If accepted a reservation session is created for the user. The token is tied to the session so it can not be reused.
11. Client receives a success response from the reservation service so it can show the reservation UI and send reservation related requests. It also displays the reservation timer.

Token security, cheat prevention and statistical fairness

Token creation

- Token creation is not rejected, everyone can enter the queue (fairness)
- Token tied to userId (read from Auth. JWT), makes stealing useless
- Short lived, can not store and come back later. Admission service can extend lifetime when needed. Favours fans who are willing to wait in line.
- Token request requires a challenge (PoW – e.g. SHA256) to root out simpler bots and automations
- Token requests probabilistically redirected to a stateful token creation endpoint that stores userId, timestamp, location, IP (in temp storage). Because the waiting room is designed to

create new tokens very rarely (exp extension for long queues) for the same user, it is easier to catch bad actors even if requests are sampled. First, demand more serious challenges (Captcha + difficult PoW) and inform the user they are flagged. Offer guidance how to properly use the reservation system and what to avoid (e.g. using multiple devices, automation). If bad behaviour continues, lock them out for a time period and warn them (e.g. via email) they violated Terms and Conditions.

Admission attempts

- Invalid Auth JWT rejected and redirected to stateful endpoint to count invalid attempts and flag the IP, userId
- Invalid token signature redirected to a stateful endpoint to flag the userId (from JWT). Detects token manipulation.
- Not respected backoff times, missing challenge solutions redirected to a stateful endpoint to flag the userId. Indicates scripting as the UI respects backoff time and challenge.
- Even valid admission attempts probabilistically redirected to a stateful admission endpoint that stores userId, timestamp, location, IP (in temp storage) to catch potential misuse and flag userId on suspicious behaviour.
- Stateless admission on the edge is just the first line. It handles huge load fast with a probabilistic decision, but the final decision is on the reservation service.
- Final admission is decided in the Reservation service. It is stateful and validates multiple criteria (current concurrency, token reuse attempt, session limit, previously flagged user behaviour, etc.) before accepts a new session.
- Accepted tokens are registered/persisted, tied to userId. Reuse is not possible.
- Reservation sessions are tied to userId with timestamps. Limits can be enforced (e.g. 1 session / event / user / 24h)

Reservation session

When a user is admitted from the waiting queue, the system creates a reservation session tied to the userId (also to the waiting queue token as stated above). Session is stateful so once admitted the user can change devices.

Session allows the user to send reservation attempts (hold/lock) while the session is active.

Reservation entity (data model) is a good place to cache the number of hold/locked seats so we can easily enforce limits (without using joins and aggregation). For high load, high performance queries, denormalised data storage can improve performance and lower DB cost. We can use TransactWriteItems (DynamoDB) on different tables in the same region providing strong consistency. Alternatively, optimistic concurrency with versioning in multi region setup (only eventual consistency).

Session states

- reservation: accepts seat hold/release requests
- checkout: checkout/payment in progress, does not accept seat hold/release requests

- finalised: payment was successful
- cancelled: user cancelled the session
- expired: session expired. Session duration (e.g. session.state == ‘reservation’ AND now – session.createdAt < reservation_time_window) always checked before an action (reservation, checkout) therefore it is not strictly necessary but for consistency and analytics (e.g. on CDC events if state changes synced to analytics service) it can be a good practice.

Reservation process

Prerequisite: User is signed in and sends an Auth JWT with each request (Cookie or Header)

1. Client sends a reservation request with a seatId (or multiple)
2. Server reads the user model (determined from JWT) from DB. Cache (e.g. ElastiCache/Redis) works well here as session concurrency is limited and users send multiple requests within the reservation window.
3. Reads the active (not expired) session associated to the user.
4. Checks hold/lock limit. Current hold count either stored in the session or queried from the DB
5. Attempts to lock (ConditionalWrite) the requested seats (multiple can be requested, as it uses TransactWriteItems – limits are acceptable for this application)
6. If the lock is successful (TransactWriteItems ensures all or nothing) returns the locked seatIds to the client
   Alternatively, the client could request auto seat selection (e.g. “Best available”, “N consecutive”)

Security, data integrity and fairness

- Session creation is limited: event / user / time frame
- Session has a max reservation limit (e.g. 10 seats)
- Session is short lived (e.g. 10 min reservation + 5 min checkout)
- Server always checks session validity, associated user, limits before accepting a reservation/checkout/etc. request. Stealing a session does not give too much advantage.
- Seats always hold atomically (Conditional write, TransactWriteItems) and single region DynamoDB Table ensures strong consistency.

Data model and reservation process

I am going to only design the potentially performance bottleneck related parts of the data model and omit most entities and attributes.

RDBMS (PostgreSQL) or NoSQL (DynamoDB)?

Large systems are usually service based (microservice architecture) so they can choose the optimal DB for the service. Services can utilise versioned records, Change Data Capture (CDC) events and durable queues to synchronise data across services reliably.

RDBMS is very flexible from a query and data access pattern point of view and the data model can be highly normalised because joins are performed on indexed columns and relatively fast.

For most of the services and related entities e.g. Users, Events, Payments, etc. the system can use RDBMS for easier design and flexibility.

But for hot data like the seat map, where high level of flexibility is not needed (queries and data access patterns are not too complex and do not change) we can utilise a very fast and cost effective NoSQL database like DynamoDB.

Data access patterns

To understand Read patterns we can recap the views on the UI

- sections (high level)
- blocks/levels/etc. inside section (intermediary)
- seat map (seats inside section/block)

Section and block level is very similar, we have a section ID and an approximated availability. As we do not need exact availability, eventual consistency, caching with stale-while-revalidate is acceptable.

For the seat map we still do not need real-time availability (1-15s data refresh is tolerable for the user, 1s for hot sections and 15s for not popular ones). But we need strong consistency on writes to avoid duplicate booking.

Single region or global tables

As concurrency is controlled and we can tolerate some latency (~1s response time for hold/release operation) for writes, a single region DynamoDB table is the best choice. We get

- fast reads (adequate partitioning and GSI),
- autoscaling (default limit: ~20k conditional writes/sec, well partitioned data),
- strong consistency,
- TransactWriteItems up to 100 items (multi-table inside same region),
- way cheaper than multi-region replication.
- Availability SLA 99.99%

Read latency will be handled with edge and regional caching (see later).

*We can use **global tables** for higher availability (SLA 99.999%) for high-stake events but partition seats and writes to a designated region to avoid write concurrency. Eventual consistency is acceptable for reads.*

Seats Table

- **Hold/reserve (write) pattern**: Conditional update on a known (user or auto-assign algorithm selected) seat.

- **Read seat-map pattern**: Query all seats for a specific section (available or not), block or row.

DynamoDB scales by increasing the number of physical partitions that are available to process queries, and by efficiently distributing data across those partitions. Therefore, to best utilise scaling we need to partition data to avoid hot partitions.

To satisfy this condition we can not partition by event as a large stadium event expected to get large write volume. The next natural partitioning is by section. If a section can tolerate < 500 hold/reservation attempts per sec it is acceptable. But, in this ultra large scale (e.g. 100k seats) design I assume very hot sections, therefore I use seat based partitioning for max scalability and a GSI for efficient section/block level seat maps.

Key-schema

Primary key		GSI1	
PK	SK	GSI1PK	GSI1SK
<eventId>#<seatId>	META	<eventId>#<sectionId>	<blockId>#<rowId>#<seatNumber>

With this key-schema we can efficiently query

- Seat availability (PK: <eventId>#<seatId>)
- Lock a seat (PK: <eventId>#<seatId>)
- Finalise a seat after checkout (PK: <eventId>#<seatId>)
- Release a seat from lock (PK: <eventId>#<seatId>)
- All seat availability in a Section (GSI1PK: <eventId>#<sectionId>)
- More granular availability in a Section
  - Block availability in a Section (GSI1PK: <eventId>#<sectionId>, begins_with(GSI1SK, <blockId>))
  - Row availability in a Block (GSI1PK: <eventId>#<sectionId>, begins_with(GSI1SK, <blockId>#<rowId>))

Attributes
eventId: <eventId>
sectionId: <sectionId>
blockId: <blockId>
rowId: <rowId>
seatNumber: <seatNumber>
seatId: <seatId> (unique id)
expiresAt: <timestamp>
updatedAt: <timestamp>
availability: available/locked/reserved

CreateTableCommand input definition

```
{
  TableName: 'Seats',
  BillingMode: 'PAY_PER_REQUEST',
```

```
  AttributeDefinitions: [
    { AttributeName: 'PK', AttributeType: 'S' },
    { AttributeName: 'SK', AttributeType: 'S' },
    { AttributeName: 'GSI1PK', AttributeType: 'S' },
    { AttributeName: 'GSI1SK', AttributeType: 'S' }
  ],
  KeySchema: [
    { AttributeName: 'PK', KeyType: 'HASH' },
    { AttributeName: 'SK', KeyType: 'RANGE' }
  ],
  GlobalSecondaryIndexes: [
    {
      IndexName: 'GSI1',
      KeySchema: [
        { AttributeName: 'GSI1PK', KeyType: 'HASH' },
        { AttributeName: 'GSI1SK', KeyType: 'RANGE' }
      ],
      Projection: {
        ProjectionType: 'INCLUDE',
        NonKeyAttributes: ['availability', 'expiresAt']
      }
    }
  ]
}
```

Example item definition

```
{
  "PK": "world-cup-final#A38#B10#R6#S34",
  "SK": "META",
  "eventId": "world-cup-final",
  "sectionId": "A38",
  "blockId": "B10",
  "rowId": "R6",
  "seatNumber": "S34",
  "seatId": "A38#B10#R6#S34",
  "expiresAt": "2025-08-28T14:30:14Z",
  "updatedAt": "2025-08-28T14:15:14Z",
  "availability": "locked",
  "GSI1PK": "A38",
  "GSI1SK": "B10#R6#S34"
}
```

Example to query availability in a Section

```
{
  TableName: 'Seats',
  KeyConditionExpression: 'GSI1PK = :es',
```

```
  IndexName: 'GSI1',
  ExpressionAttributeValues: {
    ':es': {
      S: `${eventId}#${sectionId}`
    },
  },
}
```

Example to query availability for a Row in a Block in a Section

```
{
  TableName: 'Seats',
  KeyConditionExpression: 'GSI1PK = :es AND begins_with(GSI1SK, :sk)',
  IndexName: 'GSI1',
  ExpressionAttributeValues: {
    ':es': {
      S: `${eventId}#${sectionId}`
    },
    ':sk': {
      S: `${blockId}#${rowId}`
    }
  },
}
```

Reservations Table

We will store 3 types of records in this table (common practice for NoSQL DB)

- **Event**: connected to a User by PK
- **Session**: connected to a User-Event by GSI
- **Reservation**: connected to a Session by PK and to a User-Event by GSI

Key-schema

Primary key		GSI1	
PK	SK	GSI1PK	GSI1SK
<userId>	<eventId>	<userId>#<eventId>	<eventId>
<sessionId>	S#<sessionId>	<userId>#<eventId>	<sessionId>
	R#<reservationId>	<userId>#<eventId>	<sessionId>#<state>

With this key-schema we can efficiently query

- All events the User has made reservations (PK: <userId>)
- Session for a User-Event (GSI1PK: <userId>#<eventId>, GSI1SK: <sessionId>)
- All Reservations for a session (PK: <sessionId>, begins_with(SK, R))
- Session for a User-Event and all its reservations in one query (GSI1PK: <userId>#<eventId>, begins_with(GSI1SK, <sessionId>))

- All finalised/locked (state == R/L) Reservations for a Session (GSI1PK: <userId>#<eventId>, GSI1SK: <sessionId>#R)

CreateTableCommand input definition

```
{
  TableName: 'Reservations',
  BillingMode: 'PAY_PER_REQUEST',
  AttributeDefinitions: [
    { AttributeName: 'PK', AttributeType: 'S' },
    { AttributeName: 'SK', AttributeType: 'S' },
    { AttributeName: 'GSI1PK', AttributeType: 'S' },
    { AttributeName: 'GSI1SK', AttributeType: 'S' }
  ],
  KeySchema: [
    { AttributeName: 'PK', KeyType: 'HASH' },
    { AttributeName: 'SK', KeyType: 'RANGE' }
  ],
  GlobalSecondaryIndexes: [
    {
      IndexName: 'GSI1',
      KeySchema: [
        { AttributeName: 'GSI1PK', KeyType: 'HASH' },
        { AttributeName: 'GSI1SK', KeyType: 'RANGE' }
      ],
      Projection: {
        ProjectionType: 'ALL',
      }
    }
  ]
}
```

Example to query a Session and all its Reservations in one query

```
{
  TableName: 'Reservations',
  KeyConditionExpression: `GSI1PK = :userEvent AND begins_with(GSI1SK, :sid)`,
  IndexName: 'GSI1',
  ExpressionAttributeValues: {
    ':userEvent': {
      S: `${userId}#${eventId}`,
    },
    ':sid': {
      S: sessionId,
    }
```

```
  },
}
```

Seat Lock

Multiple atomic seat reservations (all or nothing) are supported as we use TransactWriteItems. We need to take into account the 100 actions / transaction limit. Even with multiple conditional writes the system can lock 16 seats / request. That is good enough, as the system wants to prevent seat hoarding anyway. However, if more needed, we can first conditionally lock the seats and then create the Reservation records in a different transaction. It is not atomic on the DB level but as seats are locked there is no concurrency on the application level. If system fails to create a reservation record the lock will expire automatically (auto-revert).

System needs to make the following conditional writes in one transaction:

Session

- Check if Session has not been expired
- Check if Session reservation count + reservations we are about to add <= session reservation limit
- Update session reservation count (actual + new items count)

Seat

- Check if the Seat we are about to reserve is available (I added a lazy release condition: locked but lock expired is considered available)
- Update seat status to locked
- Update the seat expiresAt (lock expires) attribute to the session's expiry time as this seat is now associated to that session

Reservation

- Put/Create a reservation record

Note: DynamoDB does not have a DateTime type instead a ISO 8601 date string is used. The system stores times in UTC for string comparability.

Example TransactWriteItemsCommand input

```
{
  TransactItems: [
    {
      Update: {
        TableName: 'Reservations',
        Key: {
          PK: session.PK,
          SK: session.SK,
        },
        ConditionExpression: 'resCount <= :limit AND expiresAt > :now',
        UpdateExpression: 'ADD resCount :q SET updatedAt = :now',
```

```
        ExpressionAttributeValues: {
          ':limit': { N: reservationLimit - requestedSeatsCount },
          ':now': { S: now },
          ':q': { N: requestedSeatsCount },
        }
      }
    },
    {
      Put: {
        TableName: 'Reservations',
        Item: {
          PK: session.PK,
          SK: { S: `R#${uuid}` },
          resState: { S: 'locked' },
          seatId: { S: seatId },
          createdAt: { S: now },
          GSI1PK: { S: `${userId}#${eventId}` },
          GSI1SK: { S: `${session.PK.S}#L` }, // L - locked
        },
      }
    },
    {
      Update: {
        TableName: 'Seats',
        Key: {
          PK: { S: `${eventId}#${seatId}` },
          SK: { S: 'META' }
        },
        ConditionExpression: '#avail = :a OR #avail = :l AND expiresAt < :now',
        UpdateExpression: 'SET #avail = :l, updatedAt = :now, expiresAt = :exp',
        'ExpressionAttributeNames': {
          '#avail': 'availability'
        },
        ExpressionAttributeValues: {
          ':a': { S: 'available' },
          ':l': { S: 'locked' },
          ':now': { S: now },
          ':exp': session.expiresAt
        }
      }
    }
  ],
}
```

Reservation finalisation

Before the user redirected to the payment processor we give them 5 minutes (+2m technical time e.g. payment processing) to finish the payment instead of relying on their remaining time from reservation. This gives a clear, transparent process and a piece of mind as they have time to correct mistyped payment info or trying multiple cards.

We can use a similar Transaction as above to extend the time on checkout and after successful payment update the state (locked → reserved) of the Seat and Reservation record.

Caching

To handle a large load of availability requests we already

- partitioned the UI to 3 layers (section level, intermediate, seat-map),
- created a GSI on the Seats table to efficiently query availability (seat-maps)

To avoid the high cost of DB reads under high load, we still need to cache. Caching is acceptable because users do not need real time access to availability. Even sub second TTL with “stale while revalidate” significantly decreases the DB reads in a high-load system.

Multi-layer cache design

Technology/service	Location	Role	Description
DynamoDB (persistent)	1 write region	Source of truth. Strong consistency	Transactional writes guarantee consistency
DynamoDB Streams (CDC) → SQS → Lambda	Serverless	Async invalidation	Invalidates Redis. SQS buffers high load spikes and batching protects the cache
Elasticache/Redis (in memory)	Multiple regions with local clusters	Fast App-level cache	Mid TTL + explicit invalidation increases consistency
CloudFront/CDN	Many edge locations	Edge cache	Absorbs global read traffic

1. **DynamoDB**: Writes atomic. Provides strong consistency all the time (single region).
2. **DynamoDB Streams → SQS → Lambda**: Data changes from the DB enqueued and batch processed by a Lambda.
   - Batching: Protects Redis from constant invalidation during high write-loads. Keeps Lambda cost low
   - Selective invalidation: only updated sections/blocks are invalidated
3. **Elasticache/Redis**: Local cluster provides less latency and fast (sub-ms) retrieval. Async invalidation can allow higher TTL without sacrificing consistency.

4. **CloudFront**: This layer can handle multi-million RPS. We can use short TTL (1-2s for hot sections, 2-5 for other) and do not worry about manual invalidation. Allow stale-while-revalidate to protect from the thundering herd problem.

Redis data model

Seat-maps per section/block/row*

Key: seatmap:<eventId>:<sectionId>
Key: seatmap:<eventId>:<sectionId>:<blockId>
Key: seatmap:<eventId>:<sectionId>:<blockId>:<rowId>
Type: JSON

Example:
```
{
  "availableCount": 30,
  "seats": {
    "R12#34": "available",
    "R12#35": "reserved",
    "R12#36": "locked"
  }
}
```

** Caching by rows can be helpful for the backend to find a row with N consecutive free seats*

Available count aggregates

These aggregates can be generated from the seat maps, there is no need to query them separately. They change together.

Section availabilities for the high level view:

Key: availability:<eventId>
Type: HASH

Example fields:
  A38 -> 124
  B23 -> 25
  C8 -> 68

Block availabilities for the intermediary view:

Key: availability:<eventId>:<sectionId>
Type: HASH

Example fields:
  B3 -> 40
  B7 -> 23