This are the consideration I took building the NBA Stat app (HoopsTracker)

## Functional requirements:

* Each player is associated with a basketball team.
* The statistics recorded per game:
  + Points,
  + Rebounds,
  + Assists,
  + Steals,
  + Blocks,
  + Fouls (integer, max value: 6),
  + Turnovers
  + Minutes Played (float, between 0 and 48.0)

• Input/Output:

* the system will consume this data from an external source (non-human

system), meaning the input will be machine-readable.

* Calculating Aggregate Statistics

Once data is logged, the system should calculate aggregate statistics and serve them

* API

- Season Average per player.

- Season Average per team (average stats for all players in a team).

* The system should provide these stats in a human-readable format The stats should always reflect the most up-to-date data once the player statistics are written.
* We should support 50 000 concurrent API calls

## No Functional requirements

* Data will come in high burst for a short time
* Data aggregation should be quick
* The system should be maintainable and support frequent updates and changes across all levels.
* Scalability, High Availability and Fault Tolerant

## First Action Items:

* Check Metrics
  + Network demand
  + Data size
* Chose Incoming protocol
* Choose DB
* Select Services architecture

## Project assumptions:

* One message per event (not an array)

## Data calculation

### Assumptions:

* **League**: 1 league (NBA), fixed across sessions.
* **Teams**: 30 teams (aligned with 1230 games,), ~600 players (30 teams × ~20 players/team, conservative estimate vs. ~200 active players in prior calculations,).
* **Session**: 1 session (2024/2025), ~1230 games (confirmed,).
* **Games**: 1230 games, ~24.6 games/day over ~50 game days (1230 ÷ 20 concurrent games × ~2.5 days/week,).
* **Player Game Stats**: ~250 events/player/game (e.g., points, assists, fouls, rebounds, May 13, 2025), ~600 players × 82 games × 250 events ≈ **12,300,000 events/session** (adjusted from ~50,000,000 with ~200 players, reflecting 30 teams).
* **Field Sizes** (bytes, JSON serialized, rounded to 10):
  + UUID: 36 bytes ≈ **40 bytes**.
  + VARCHAR(255): Average 50 bytes ≈ **50 bytes**.
  + VARCHAR(10): 10 bytes ≈ **10 bytes**.
  + TIMESTAMP/TIMESTAMPTZ: 20 bytes ≈ **20 bytes**.
  + DATE: 10 bytes ≈ **10 bytes**.
  + TIME: 8 bytes ≈ **10 bytes**.
  + BOOLEAN: 1 byte ≈ **10 bytes**.
  + ENUM: 10 bytes ≈ **10 bytes**.
  + TEXT: 10 bytes ≈ **10 bytes**.
  + INTEGER/FLOAT: 10 bytes ≈ **10 bytes**.
* **Indexes**: ~50% of row size
* Estimated message size for Incoming request (9 fields): 150 bytes
  + You can see the calculation in the Appendix

### Realistic Player Stat Averages (per game):

|  |  |
| --- | --- |
| **Event Type** | **Estimated Events** |
| Points | 20 |
| Rebounds | 6 |
| Assists | 5 |
| Steals | 2 |
| Blocks | 1 |
| Fouls | 3 |
| Turnovers | 3 |
| **Minutes Played** (1/min) | 48 |
| **Reserved Future** | **10** |
| **Total** | **98 events/player/game** |

### 10 Players on the Court All the Time:

10 players × ~100 events/player/game = 1000 events/game

### Events/sec per game

1,000 events ÷ 2,880 seconds = ~0.347 events/sec/game

### Events/sec per 20 games(load)

0.347 × 20 = 6.94 events/sec (total)

### Expected Incoming Data Volume (Live Streaming)

|  |  |
| --- | --- |
| **Metric** | **Value** |
| Total events per second | ~6.94 events/sec |
| Event size | 150 bytes |
| **Incoming data per second** | **~1,041 bytes/sec (~1.02 KB/sec)** |
| Incoming data per minute | ~62 KB |
| Incoming data per hour | ~3.7 MB |

### Expected outgoing network calculation:

#### Assumptions

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| **Fields per message** | 10 |
| **Concurrent messages/sec** | **50,000** |
| **Serialization format** | JSON |
| **Field types** | UUIDs, timestamps, enums, integers, etc. |
| **Estimated size per message** | Let’s assume ~**150 bytes** per message (based on 10 fields) |

#### Data Volume per Second

50,000 messages/sec × 150 bytes = 7,500,000 bytes/sec = **~7.15 MB/sec**

|  |  |  |
| --- | --- | --- |
| **Time Frame** | **Formula** | **Data Volume** |
| **Per second** | 50,000 × 150 | **~7.15 MB** |
| **Per minute** | 7.15 MB × 60 | **~429 MB** |
| **Per hour** | 429 MB × 60 | **~25.7 GB** |
| **Per day** | 25.7 GB × 24 | **~617 GB/day** |

### DB size calculation:

We have 5 basic tables here is a raff estimate:

#### Storage Breakdown (Per Season)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Table** | **Rows** | **Est. Row Size (Bytes)** | **Data Size (MB)** | **Indexes (Count × 50%)** | **Index Size (MB)** | **Total Size (MB)** |
| **League** | 1 | 210 | 0.00021 | 1 × 105 B | ~0.0001 | **~0.0003** |
| **Team** | 30 | 280 | 0.0084 | 1 × 140 B | ~0.0042 | **~0.0126** |
| **Session** | 1 | 120 | 0.00012 | 1 × 60 B | ~0.00006 | **~0.00018** |
| **Game** | 1230 | 370 | 0.46 | 1 × 185 B | ~0.23 | **~0.69** |
| **Player** | 600 | 180 | 0.108 | 1 × 90 B | ~0.054 | **~0.162** |
| **Player\_Stat\_Event** | 1,230,000 | 100 | 123.0 | 4 × 50 B | ~246.0 | **~369.0** |
|  |  |  |  |  |  |  |
| **🟩 Total Storage** | — | — | **~123.58** | — | **~246.29** | **~369.87 MB** |

## DB Selection

Database Comparison

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature** | **Cassandra** | **ClickHouse** | **TimescaleDB** | **PostgreSQL** | **MongoDB** |
| **Data Model** | Wide-column (NoSQL) | Columnar (OLAP) | Relational + Time-series | Relational (OLTP) | Document (JSON/NoSQL) |
| **SQL Support** | ⚠️ Partial (CQL only) | ✅ SQL-like (read-optimized) | ✅ Full SQL (PostgreSQL-based) | ✅ Full SQL | ⚠️ Limited (aggregation only) |
| **Joins / Relationships** | ❌ None | ⚠️ Limited | ✅ Yes | ✅ Yes | ⚠️ Limited (no true joins) |
| **Write Throughput** | ✅✅ Very High | ✅ High | ✅ Moderate to High | ⚠️ Moderate | ✅ High |
| **Read Throughput** | ✅ High (partitioned) | ✅✅ Very High | ✅ Moderate to High | ⚠️ Moderate | ⚠️ Moderate |
| **50K QPS Support** | ✅ Yes (denormalized only) | ✅✅ Yes (designed for this) | ⚠️ Borderline (w/ replicas) | ❌ No (not at that scale) | ❌ No |
| **ACID Support** | ⚠️ Limited | ❌ No | ✅ Full (via PostgreSQL) | ✅ Full | ⚠️ Limited (tunable) |
| **Horizontal Scale** | ✅✅ Native | ✅✅ Native | ⚠️ Requires manual sharding | ❌ Not native | ✅ Yes (with sharding) |
| **Aggregate Support** | ⚠️ Poor | ✅✅ Excellent | ✅ Good | ✅ Good | ⚠️ Moderate |
| **Scalability** | ✅✅ Linear | ✅✅ High (read-mostly) | ✅ Good (replicas + tuning) | ⚠️ Limited | ✅ High (with config) |
| **Adoption & Ecosystem** | ⚠️ Niche | ⚠️ Specialized | ✅ Growing (PostgreSQL-based) | ✅ Mature | ✅ Very High |

### DB selected: TimescaleDB(With Redis):

#### **Selected TimescaleDB combined with Redis to reduce load and enhance top-scale performance.**

1. **Purpose-Built for Time-Series + Relational Data**  
    TimescaleDB extends PostgreSQL with time-series capabilities, making it ideal for tracking player stats, events, and game sessions that are all time-dependent — like live game stats streamed every second.
2. **Full SQL Support**  
    Unlike many NoSQL or specialized time-series databases, TimescaleDB supports **full SQL**, including joins, complex queries, and rich analytics. This makes it flexible and easy to integrate with existing applications and BI tools.
3. **ACID Compliance**  
    TimescaleDB inherits PostgreSQL’s ACID properties, ensuring data integrity and reliable transactions — critical when tracking live stats and player events where consistency matters.
4. **Good Performance and Scalability**  
    It handles large volumes of time-series data efficiently and can scale vertically with partitioning and hypertables. While horizontal scaling requires more setup than some NoSQL systems, it’s solid for medium-to-large workloads like yours (thousands of events per game, 50 concurrent games, etc.).
5. **Rich Ecosystem & Tooling**  
    Being built on PostgreSQL means access to a mature ecosystem, including powerful extensions, backup tools, replication options, and widespread community support.
6. **Compression and Data Retention**  
    TimescaleDB offers native compression and data retention policies, helping reduce storage costs for historical game data without sacrificing query performance.
7. **Query Flexibility**  
    Supports complex aggregation queries (e.g., player averages, game summaries) that you will need for analytics and reporting on player stats over seasons.
8. **Ease of use:**

Work with SQL queries, same as Postgress

## Traffic protocols:

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **WebSocket** | **HTTP Polling (Pull)** | **HTTP Push (SSE / HTTP/2 Push)** |
| Streaming Support | Full duplex, continuous streaming | No, client polls at intervals | Server pushes updates continuously |
| Latency | Very low (ms-level) | High latency due to polling intervals | Low latency (better than polling) |
| Reliability | High (persistent connection) | Lower (stateless, prone to missed data) | Moderate (persistent but less robust) |
| Scalability | Good, requires connection management | Easier to scale (stateless) | Moderate (long-lived connections) |
| Bandwidth Efficiency | High (only changes pushed) | Low (repeated full requests) | High (only changes pushed) |
| Server Load | Moderate to high | High (many repeated requests) | Moderate |
| Client Complexity | Medium (manage connection, subscriptions) | Low (simple requests, manual filtering) | Medium (listen to event stream) |
| Adoption & Support | Widely supported | Supported everywhere | Supported in modern browsers |
| **Rate Limiter Issues** | **Less prone** (single persistent connection) | **More prone** (many requests can trigger limits) | **Less prone** (persistent but limited by server) |

### Incoming Selection: Web Socket

|  |  |
| --- | --- |
| **Reason** | **Explanation** |
| **Low Latency & Real-Time** | WebSocket offers true bi-directional, low-latency streaming ideal for real-time sports stats. |
| **Efficient Bandwidth Usage** | Pushes only changes/events — no repeated full requests like polling, saving bandwidth. |
| **Scalable for Many Events** | Supports thousands of events per second per connection, minimizing overhead and latency. |
| **Reliable & Persistent** | Keeps connection open, reducing connection setup/teardown overhead and missed data risks. |
| **Supports Topic Filtering** | Can subscribe to specific players/games, so clients only get relevant updates, reducing load. |
| **Better Server Load Management** | One persistent connection per client reduces total connection churn compared to polling. |
| **Lower Rate Limiting Risk** | Fewer, long-lived connections avoid hitting HTTP rate limits common with polling. |
| **Widely Supported** | Supported by all modern browsers and backend frameworks, with mature libraries and tooling. |

### Outgoing traffic selection Http rest:

|  |  |
| --- | --- |
| **Reason** | **Explanation** |
| **Stateless & Scalable** | REST APIs are stateless, making it easier to scale horizontally with load balancers and auto-scaling. |
| **Simple & Ubiquitous** | REST is universally supported and easy to implement on both client and server sides. |
| **Better for Request/Response** | REST works well for discrete, on-demand data fetches (e.g., client requests latest stats or summaries). |
| **Easier to Cache & Rate Limit** | HTTP caches and rate-limiters can be easily applied, improving performance and protecting servers. |
| **Firewalls & Proxies Friendly** | REST works smoothly through firewalls and proxies, which sometimes block WebSocket traffic. |
| **Easier Debugging & Monitoring** | HTTP requests can be logged, traced, and debugged easily with standard web tools. |
| **Supports Load Distribution** | Can distribute load effectively across multiple servers with standard HTTP infrastructure. |
| **Statelessness Suits High Client Volume** | Because no persistent connection is maintained, REST can efficiently support **thousands or even millions of concurrent clients** making discrete requests without connection overhead. |
| **Supports Multiple Clients Easily** | REST API endpoints handle multiple clients naturally via stateless requests, allowing concurrent connections without complex connection management. |

## Cache: Redis

Why Use Caching in Your Scenario?

* **High-frequency reads** (500 requests/sec, possibly 50K bursts).
* **Limited working set:** ~15 players/team × 30 teams = ~450 player records updated frequently.
* **Typical queries target the current season** — this keeps the dataset small and focused.
* **Small payloads** per item (a few fields per player stat).
* **Access patterns are predictable**: clients typically query recent stats or trending players.

## Scale & reliability and fault tolerate

### Options

1. **Single Monolith**  
    One service handles ingestion, processing, DB writes, and API reads.
2. **Split Write/Read Services**  
    Two services:
   1. Ingestion Service (receives and writes to DB)
   2. API Service (reads from DB and responds to REST requests)
3. **Three-Tier with Queue**
   1. Service A: Ingests incoming data → pushes to **Queue**
   2. Service B: Reads from Queue → writes to **DB**
   3. Service C: Reads from **DB** → serves REST API

### Architecture Comparison Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **🧱 Monolith** | **⚖️ Split Read/Write** | **🧩 3-Tier with Queue (Ingest + Process + API)** |
| **Complexity** | ⭐ Low | ⚠️ Medium | 🔺 High (more components, coordination needed) |
| **Reliability** | ❌ Single point of failure | ✅ Better (isolate crashes) | ✅✅ High fault tolerance (each part isolated) |
| **Scalability (Horiz.)** | ⚠️ Limited | ✅ Scalable by service | ✅✅ Scales per role (e.g., Kafka consumers, DB writes) |
| **Auto-Scaling** | ❌ Hard to isolate load | ✅ Scale read/write independently | ✅✅ Fine-grained auto-scaling per concern |
| **Fault Tolerance** | ❌ Full restart on failure | ⚠️ Partial crash possible | ✅✅ Retry queues, backpressure, no data loss |
| **Throughput** | ⚠️ Risk of bottlenecks | ✅ Moderate | ✅✅ Handles spikes with queue buffering |
| **Latency (End-to-End)** | ✅ Fastest | ✅ Similar to monolith | ⚠️ Slightly higher (queue adds delay) |
| **Monitoring/Debugging** | ✅ Simple logs | ✅ Moderate | ⚠️ Requires distributed tracing/log correlation |
| **Development Speed** | ✅✅ Fast | ✅ Moderate | ⚠️ Slower (more infra, services, CI/CD) |
| **Queue Backpressure** | ❌ None | ❌ None | ✅ Handles DB slowness gracefully |
| **Deployment Flexibility** | ❌ All-or-nothing | ✅ Partial deploys | ✅✅ Isolated deploys per service |
| **Cost** | ✅ Least infra cost | ⚠️ Moderate | 🔺 Higher infra (queue, load balancer, scaling) |

Given:

* Real-time incoming data (~1000 events/game)
* 450 active players per session
* 50K concurrent API requests
* Focus on **scalability**, **resilience**, and **performance**

### Architecture Selection: Three-Tier with Queue

* Handles spikes and failures well (queues absorb bursts)
* Let's you scale ingestion, DB write, and API services separately
* Improves resilience: one part can fail without breaking others
* Easy to throttle writes, retry failed operations, and recover gracefully

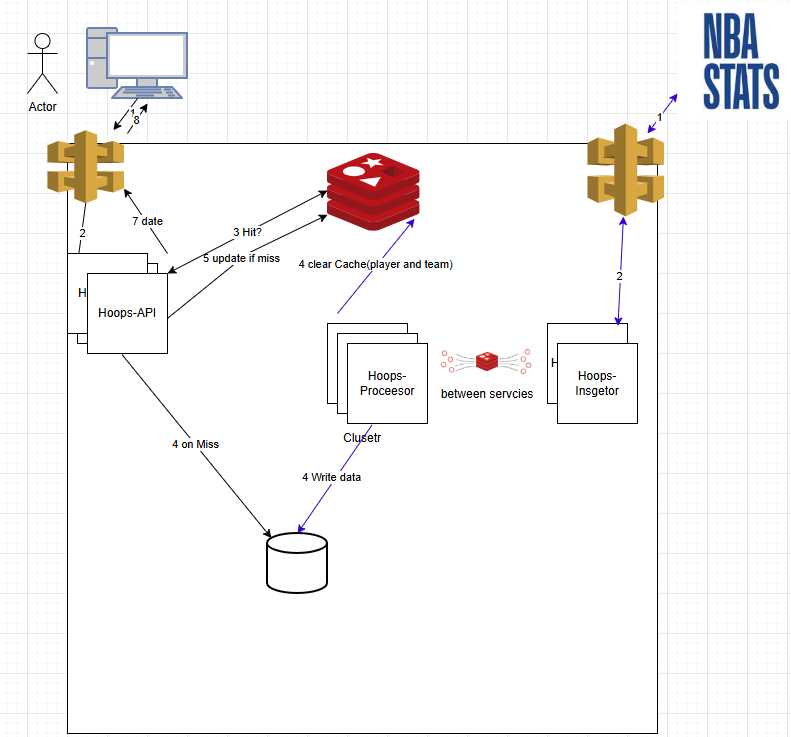
## Message Broker

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Redis Streams** | **RabbitMQ** | **Apache Kafka** |
| **Ideal Throughput** | 🔹 Low–Medium (<100K msg/sec) | 🔹 Medium (10K–100K msg/sec) | 🔸 High (100K+ msg/sec) |
| **Latency** | ✅ Low (~1ms) | ✅ Low (~1ms) | ✅ Low (~2–5ms typical) |
| **Persistence** | ⚠ Optional (data may be lost if Redis crashes) | ✅ Yes (messages can be persisted to disk) | ✅ Yes (strong durability, configurable retention) |
| **Ordering** | ✅ Per-stream | ✅ Per-queue | ✅ Per partition |
| **Fault Tolerance** | ⚠ Not built-in (needs Redis Sentinel or Cluster) | ✅ Yes (mirrored queues with HA) | ✅ Yes (built-in, replicated brokers/partitions) |
| **Scalability** | ⚠ Limited (shard manually, limited HA) | 🔹 Moderate (vertical scaling + HA clusters) | ✅ Excellent (horizontal & vertical) |
| **Ease of Setup** | ✅ Very easy (especially self-hosted or managed) | ⚠ Medium (more setup, needs tuning) | ❌ Complex (Zookeeper, broker tuning, etc.) |
| **Adoption** | ✅ Widespread in caching/stream use | ✅ Very common in traditional systems | ✅ Industry standard for high-throughput data pipelines |
| **Reliability** | ⚠ Medium (use AOF/persistence, but risk in crashes) | ✅ High (ACKs, delivery guarantees, HA possible) | ✅ Very High (exactly-once/at-least-once supported) |

### Message Broker selectionRedis Streams

* Super **lightweight** and **fast**.
* **Perfect** for ~10 events/sec scale.
* **Easy to deploy**, monitor, and scale if needed.
* **Built-in stream ID** **support** for ordering.
* **Native in-memory** (can persist to disk if needed).
* **Plays well** with a simple microservice model.
* **AOF (Append-Only File)** enabled for persistence.
* **Redis Sentinel** or **Redis Cluster** for failover/high availability.

## Design:



## Appendex

### Updated Field Size Breakdown

|  |  |  |
| --- | --- | --- |
| **Field** | **Value Example** | **Est. Size (bytes)** |
| eventId | UUID | 36–40 |
| playerId | short string | 5–10 |
| teamId | short string | 5–10 |
| gameId | short string | 5–10 |
| eventType | "point", etc. | 5–10 |
| value | number | 1–2 |
| timestamp | ISO string | ~20 |
| courtZone | e.g., "paint" | ~8–15 |
| eventSource | e.g., "sensor" | ~6–10 |
| JSON overhead | quotes, commas, etc | ~20 |
| **Estimated Total** |  | **~135–150 bytes** |