Explain your thought process, challenges encountered, and trade-offs made during implementation.

Handling Live Data Fetching and Season Statistics

The Challenge:

We need to provide **season-level statistics** for both players and teams—even when they are **actively playing a game**. This means the system must serve **up-to-date aggregated stats**, blending both **completed game data** and **live game activity**.

Initial Idea: Merge Completed Data (from DB) + Live Data (from Redis)

My initial approach was to:

- Store **completed game stats** in **PostgreSQL (RDS)**.
- Store **ongoing/live game stats** in **Redis (ElastiCache)**.
- On each query, **merge the two sources** to produce season-level stats.

❌ *Issues with this approach:*

- **Code complexity:** Merging logic for each request adds overhead and increases error potential.
- **Latency:** Redis is fast, but fetching and merging with SQL adds delay.
- **Wasted potential:** We underutilize Redis by treating it as a transient layer only.

☑ Final Design: Keep Season Stats in Redis all times

We revised the design to simplify and speed up both read and write operations:

- Redis now acts as the **primary store for in-season statistics**.
- Every game update—live or final—**directly updates Redis** (per player and per team).
- PostgreSQL stores raw, completed game logs for durability and historical analysis.
- When a game ends, **no special merging or migration is needed**—Redis already holds the up-to-date season totals.

Benefits:

- **High performance reads and writes** for fans and APIs.
- **Simple data model** – no need to merge Redis + DB every time.
- **Durability retained** via background persistence to PostgreSQL.
- **No staleness** – current season stats always reflect the latest updates.

Drawbacks:

- Requires careful and sometimes painful setup of serializers in Java (e.g., Jackson).
- Higher implementation effort for atomic updates and consistency logic.
- Serialized data in Redis is not easily human-readable.
- Steeper learning curve compared to using traditional databases.

Why PostgreSQL?

I chose a SQL database because statistical computation is easier and more convenient in SQL due to its structured nature. Choosing SQL over NoSQL is justified because statistical operations (like averaging, grouping by player/team) are more naturally and efficiently expressed in SQL.

Among SQL options, I selected PostgreSQL instead of MySQL because PostgreSQL has richer language features for analytical queries and offers better performance for complex aggregations and calculations.

How to hold HA and fault tolerance on DB?

Additionally, I opted for a master-replica architecture for RDS:

- One primary PostgreSQL instance handles all writes and critical reads.
- One read replica improves read throughput.
- In case of increased load, more read replicas can be added.
- If traffic or write-intensive workload increases significantly, migrating to Amazon Aurora PostgreSQL is a viable upgrade path due to its higher scalability and fault-tolerance.

API Considerations?

To support this, I designed the following API:

`PUT /stat/live/game`

It receives a JSON object that contains statistics for one of the players. Example format:

- Game ID: 1008
- Team ID: 19
- Player ID: 19
- Points: 10
- Rebounds: 8
- Assists: 54
- Steals: 2
- Blocks: 4
- Fouls: 2
- Turnovers: 3
- Minutes Played: 34

Initially, I considered creating **separate APIs** for updating player and team statistics. However, I later decided that **team statistics can be derived** by aggregating the data from individual players. This approach simplifies the ingestion logic and avoids data duplication.

Query Endpoints

We expose two main endpoints for retrieving statistics:

- `GET /stat/player/{playerId}` – Returns aggregated statistics for a specific player.
- `GET /stat/team/{teamId}` – Returns team-level statistics, computed on-the-fly from player stats.

These endpoints read data from **Redis**, which serves as our **hot cache** for fast access. The **PostgreSQL database** acts as a **cold store**, ensuring durability and allowing loading data back into Redis cache when necessary.

Ensuring High Availability for Redis

I considered different options for how to set up Redis across two availability zones:

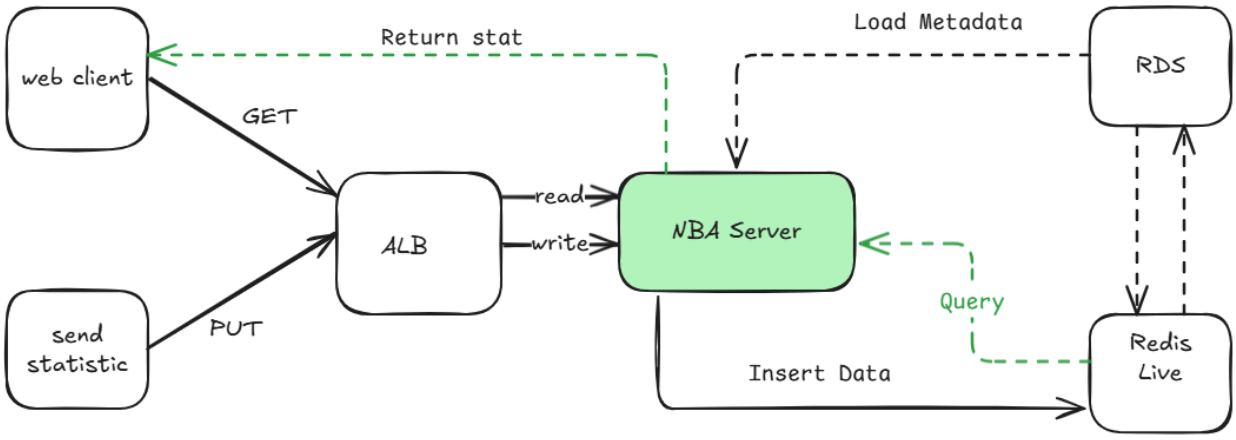
1. Two Redis instances (Zone A and B), both write-enabled – but this adds complexity and risks with syncing.
2. One Redis in Zone A as primary (read and write), one in Zone B as read-only replica.
    a. Pro: Read load can be shared
    b. Con: Writes from Zone B would cross zones to write to Zone A.
3. One Redis in Zone A (read/write), with a replica in Zone B only for failover.
    a. Pro: Simpler and safer with AWS Multi-AZ support
    b. Chosen option: I selected this model as it simplifies write routing and relies on built-in failover without extra logic.

This setup provides strong availability with minimum operational complexity.

System Architecture Diagram: Data Flow for Write and Read Operations

Below is a high-level system diagram (see attached image) that illustrates the flow of data:

- `PUT /stat/live/game` requests go to the Java service, which writes to Redis.
- `GET` requests for team or player stats first check Redis for live updates and then retrieve historical data from PostgreSQL.
- At the end of a game, Redis data is flushed to PostgreSQL for persistence



Enriching Redis Results with Human-Readable Data

To provide user-friendly responses (e.g., real player or team names instead of just IDs), we explored several solutions for enriching Redis results:

Option 1: Querying the Database at Runtime

After retrieving statistical data from Redis, the system sends an additional query to the database to fetch the corresponding player or team name based on `playerId` or `teamId`.

- ☑ **Pros:** Simple to implement; always gets the latest names.
- ❌ **Cons:** Involves additional database queries on every request, which **introduces latency** and defeats the purpose of fast Redis access.

Option 2: Load Player and Team Metadata into Redis

During system startup (or via a scheduled background job), we preload the **roster data**—including player and team names—into Redis. When a query is made, both the statistical data and the human-readable metadata are retrieved from Redis and merged before returning the response.

- ☑ **Pros:**
  - Keeps response latency low (no DB hits).
  - All data served from Redis = fast and scalable.
  - Works well if names change infrequently.
- ❌ **Cons:**
  - Adds complexity to **data loading and cache management**.

- Requires extra logic to **merge statistical and metadata entries**.
- Risk of **data staleness** if names change and Redis isn't refreshed.

Option 3 (Chosen): Local In-Memory Caching on Each NBA Server Instance (Docker/Pod)

Each **NBA server instance** (Docker container or Kubernetes pod) holds a **local in-memory cache** of player and team metadata (e.g., names). The cache is refreshed **once per night**, since the data changes very rarely.

Given the modest size of this dataset—approximately **400 players** and **40 teams**—the memory footprint is negligible, and the approach remains lightweight and highly efficient.

- ☑ **Pros:**
  - **Ultra-fast enrichment** with no external lookups.
  - **Simple logic** — immediate access to names for response formatting.

Alternative Consideration: Adding Kafka for Decoupling and Durability

At one point, I considered introducing **Kafka (or a similar message queue)** to decouple the system. In this model:

- The NBA server endpoints would act as **stateless converters**, receiving incoming HTTP requests with player statistics and publishing them to a **Kafka topic**.
- **Consumer services** would then process those messages asynchronously, updating Redis and PostgreSQL in the background.

This architecture would have offered several benefits:

- ☑ **Durability:** Kafka retains messages even if the server crashes — no data is lost.
- ☑ **Fault tolerance & scalability:** Decouples write path from processing, enabling horizontal scaling of consumer workers.
- ☑ **Backpressure handling:** Kafka can buffer bursts without overwhelming Redis or the DB.

However, after evaluating the **expected load** described in the assignment — around **tens to a few hundreds of requests per second** — I concluded that introducing Kafka would be

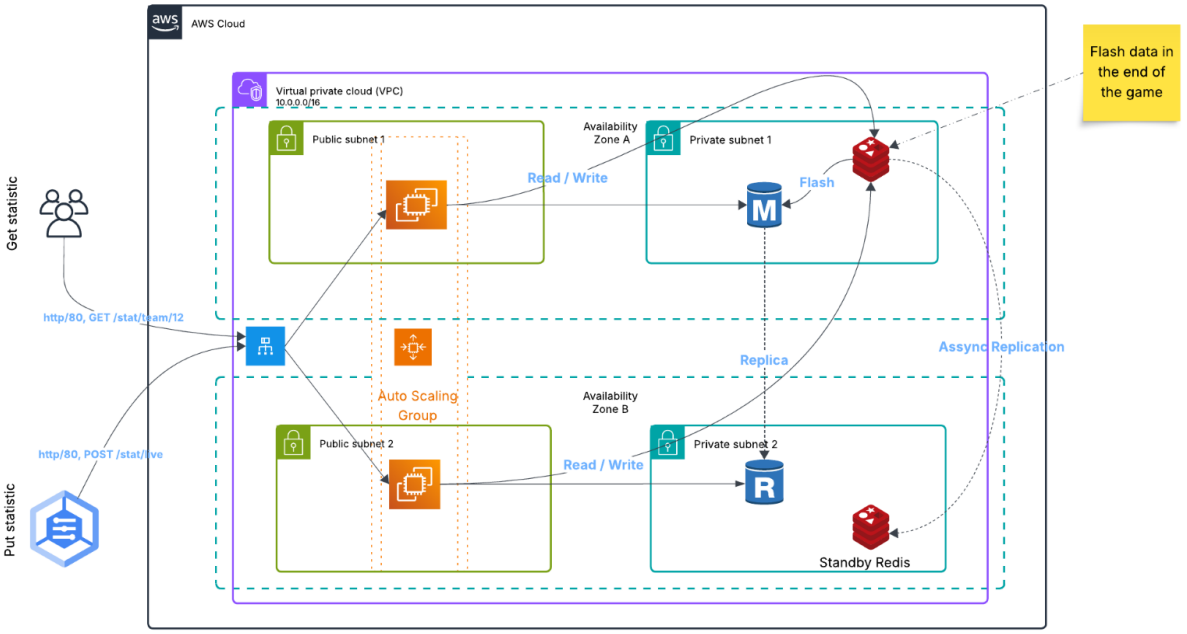
overkilling for this scale.

Deployment Diagram: AWS Cloud Infrastructure

In production, the system is deployed on AWS across two availability zones. Each zone contains a public and private subnet:

- **Public subnet** hosts an Application Load Balancer (ALB) and Java API servers, deployed via Docker.
- **Private subnet** hosts the backend services:
    - **Amazon RDS (PostgreSQL)** for durable game data storage, with one primary and one read replica.
    - **Amazon Elastic Cache (Redis)** for in-memory storage of live game statistics. The Redis setup uses a single primary in one zone and a replica in the other zone, configured for multi-AZ failover.

The ALB distributes HTTP traffic between Java servers in both AZs. The Java API layer communicates with Redis and RDS, performing live data ingestion and seasonal stats aggregation. Redis is used for low latency writes and reads, while PostgreSQL handles historical storage. This architecture ensures scalability, fault tolerance, and low-latency performance for both read and write workloads.



How it works

Live Stat Ingestion Flow – Functional Walkthrough

1. **Entry Point**
   The ingestion process begins with a PUT request to the **`IngestStatController`**, which receives a `LiveStatLine` object containing:
   a. `playerId`
   b. `gameId`
   c. Statistical snapshot (points, assists, rebounds, etc.)
   d. (Possibly other metadata in future — e.g., timestamps, source)
2. **Delegation to Service Layer**
   The controller forwards the request to `LiveStatService`.
3. **Delta Calculation Logic**
   To **update cumulative stats in Redis**, we must compute the **delta** between the new snapshot and the previous one stored for the same player/game.
   a. We **read the last snapshot** from a Redis bucket like `g:1000:p:20` (for game 1000, player 20).
   b. For each stat, we compute:
   `delta = newValue - previousValue`

4. **Atomic Aggregation Update**
   We apply the delta to the **season-level player stats** stored in a Redis key like: `s:2024/25:p:20`
   This is done using Redis atomic commands (`HINCRBY`, `HINCRBYFLOAT`) to update values like points, assists, etc.
5. **Snapshot Caching**
   Finally, the **new snapshot is stored** in Redis (in `g:1000:p:20`) for the use in future delta calculations.

- ## Why Delta Calculation is Necessary

When ingesting live player statistics, we receive **only the current snapshot** of a player's performance within a single game (e.g., 18 points, 4 assists so far in Game 6). The Redis key used for season stats, such as `s:2024/25:p:20`, is an **accumulation of stats across all games** played in the season. It is not tied to any specific game context.

At first (for example, during the player's first game of the season), it might seem sufficient to simply write the snapshot directly into the season bucket. However, as soon as: a second game begins, or multiple updates are received during a game, this approach breaks down. Simply overwriting the season bucket would **erase previously accumulated stats** and introduce incorrect totals. To avoid this, we: **Read the last known snapshot** for the player in the current game (from `g:{gameId}:p:{playerId}`). **Calculate the delta** (the change since last update). **Apply only the delta** to the season stats using Redis atomic increment commands (`HINCRBY`, etc.). **Update the game snapshot** so it can be used for the next delta calculation. This method ensures correctness, preserves performance, and handles repeated updates without duplication.

6. **Updating the Database via Dirty-Flagged Keys.** After updating a player's season aggregator in Redis (`s:{season}:p:{playerId}`), we signal that this data needs to be synchronized with the SQL database. **Mechanism:** we **mark the Redis season bucket as dirty**, e.g., by adding the key to a special Redis set `dirty:season:players.` Or setting a Redis key like `dirty:s:{season}:p:{playerId}` with TTL. A background **Update Process** (DB Sync Worker) periodically scans the dirty set or key space, reads the full player season stats from Redis, persists the updated stats to the SQL database. **Clears the dirty mark** to avoid redundant updates. This allows Redis to stay the **source of truth for hot data**, while PostgreSQL serves as **durable cold storage**. It avoids immediate writes to the DB on every update (high throughput benefit). It ensures the DB reflects Redis state eventually and consistently.

7.