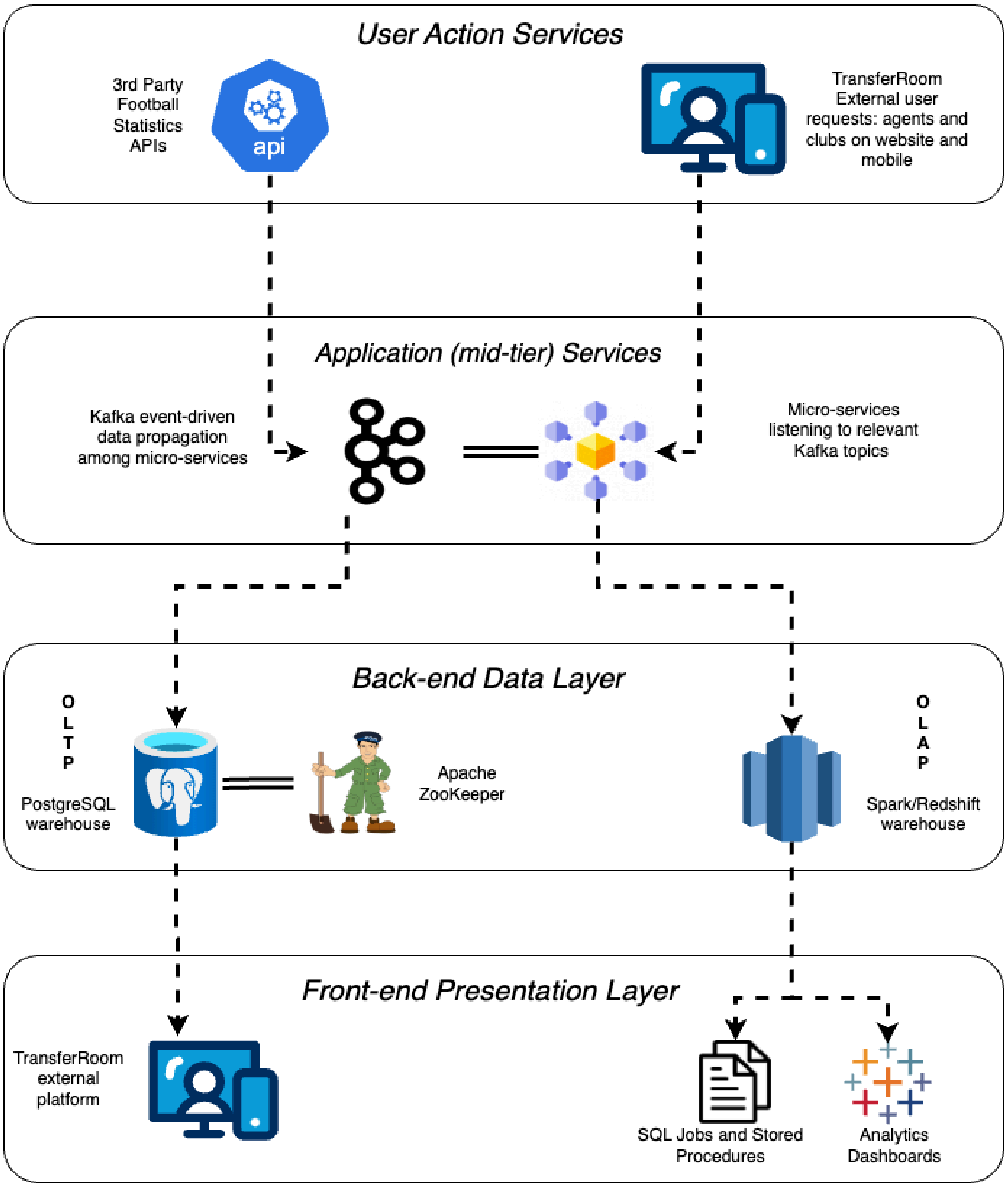
ShellCorp (TransferRoom) data architecture

Key idea

To break down a monolithic application. The data architecture below combines stateless services, event-driven communication via Kafka, and a separation of OLTP and OLAP concerns. This makes the system more resilient and flexible, as it enables horizontal scaling, ensures real-time data propagation, and guarantees that each service can be independently scaled and maintained.


User Action Services
3rd Party Football Statistics APIs
api
TransferRoom External user requests: agents and clubs on website and mobile
Application (mid-tier) Services
Kafka event-driven data propagation among micro-services
Micro-services listening to relevant Kafka topics
Back-end Data Layer
OLTP
PostgreSQL warehouse
Apache ZooKeeper
OLAP
Spark/Redshift warehouse
Front-end Presentation Layer
TransferRoom external platform
SQL Jobs and Stored Procedures
Analytics Dashboards

Why move to a microservices architecture? A transition to this architecture will allow more resilience in times of high traffic. This is highly relevant for TransferRoom, where parts of the platform (e.g., communication flows between agents and clubs) will experience higher traffic during transfer periods twice a year. An isolation of discrete functionalities in the form of services will promote clear domain boundaries, and reduce the risk of side effects between unrelated parts of the application. The ability to scale services independently also means that resources will be allocated more efficiently based on the load.

Front-end services

Client-facing services request data from mid-tier or application services, and fetch data from various domains to handle presentation logic. These presentation services only access mid-tier APIs, they're decoupled from backend storage and unaffected by concurrency complexities in Riak or Kafka.
Example: The `LiveRatingService` frontend can subscribe to `LiveRatingUpdates` to show live updates to users as they are published.

Application (mid-tier) services

The core business logic layer that handles interaction with the microservices. It coordinates services to handle products, including both agent products (live player ratings in profiles and pitches to clubs); and club products (player messaging and up-to-date performance rankings). Each mid-tier service listens to Kafka topics for updates relevant to their functionality. For each new product, a new service can be created with its own API and business logic.
Example: `PlayerPerformanceService` can subscribe to the `PlayerUpdates` topic to receive real-time stats. Each service handling high-throughput, eventually-consistent data can rely on Riak's CRDTs to resolve concurrent updates automatically.

Kafka service (Event Stream)

Kafka will serve as the backbone for the event-driven data propagation between services. By using Kafka as an event bus, each service can broadcast or consume events as needed, ensuring asynchronous communication, where various services (OLTP, OLAP) produce and consume events, such as user actions or updates, without direct coupling between systems.

Why Kafka? By partitioning topics based on unique identifiers (e.g., player IDs or match IDs), Kafka ensures that related events are processed in order, and enables horizontal scalability by distributing the event load across multiple consumers. Additionally, message retention capabilities allow for replaying past events, enabling features like system recovery or consistency checks, which would be difficult to achieve with direct database-to-database communication.

Example:

1. When a mid-tier service (e.g., `PlayerPerformanceService`) processes an update, such as a player rating changed after a match, it publishes an event to a Kafka topic (e.g., `LiveRatingUpdates`). `PlayerPerformanceService` also manages rating data in Riak. Riak's

CRDTs ensure these scores are consistent, while Kafka propagates them to other services in real-time.
2. Other services (e.g., `AnalyticsService`, `TeamPerformanceService`) that require this data for analytics or reporting subscribe to the `LiveRatingUpdates` topic to stay updated without direct calls to the data layer.

OLTP Service (PostgreSQL Database)

This service is responsible for handling all OLTP transactions. It manages real-time data consistency (e.g., user profiles, transactions, account balances) and interfaces with Kafka to send updates. Data is synchronized with Kafka to update other systems without slowing down OLTP performance. PostgreSQL maintains ACID properties to ensure reliable and consistent transaction processing. High-availability databases can also be optionally built on Riak/Cassandra for services like `LiveScores` or `PlayerPerformance`, where eventual consistency and low-latency writes are prioritized.

ZooKeeper

Coordinates the health, availability, and leadership management for distributed PostgreSQL nodes to ensure high availability. ZooKeeper keeps track of cluster metadata - each node registers itself in ZooKeeper, and ZooKeeper maintains an authoritative mapping of partitions to nodes.

OLAP Service (Spark/Redshift)

This service handles all analytics and batch processing of data. It is responsible for data aggregation, historical analysis, and reporting. It consumes events from Kafka to update the OLAP layer in real-time. Kafka streams player performance data and score updates to the Massive-Parallel-Processing system, enabling immediate aggregation for real-time analytics without impacting OLTP performance. Kafka Streams or Spark Streaming can also perform in-stream aggregations for metrics like average player rating, which are then persisted to the OLAP database for historical analysis. Finally, Kafka's event storage allows replays of past events, facilitating system recovery and consistency checks when discrepancies are detected.

Why break a single SQLServer into separate OLTP and OLAP services? OLTP systems are built for low-latency, high-concurrency transactional workloads, ensuring consistency across user interactions. OLAP systems, on the other hand, are designed for heavy batch processing and complex analytical queries that require aggregating large volumes of data across various dimensions. Separating OLTP and OLAP processes enables each system to focus on its specific task, optimizing for different trade-offs.

Data Ingestion Step-by-Step Process

In this SOA architecture, data ingestion flows are centered on Kafka for decoupled, real-time data streaming between the OLTP and OLAP layers:

1. User Actions Service: Captures user actions (e.g., score updates) and sends events to the Application (mid-tier) Service
    - Included in sample implementation. Note that I have created a mock-api Python file that generates random player ratings when requested. This can be replaced by APIs that are connected to real-time data.
2. Application Service: Validates and publishes these events to Kafka as messages.
    - Included in sample implementation. Note that listening, publishing, and simple validation functionalities are mocked in the implementation. The first next step would be to write unit tests for these services.
3. Kafka Service: Acts as a central event stream, enabling real-time distribution to downstream consumers.
    - Included in the sample implementation
4. OLTP Service (PostgreSQL): Consumes Kafka messages to update the PostgreSQL database with transactional data, such as user profiles and ratings, ensuring immediate consistency.
    - Included in the sample implementation.
5. OLAP Service (Spark/Redshift): Consumes Kafka messages to process and store data for analytics, supporting real-time and batch analysis for performance metrics and historical reports.
    - Not included in the sample implementation. This relational-database service layer will listen to Kafka topics like the OLTP service layer.
6. ZooKeeper: Coordinates high availability for PostgreSQL in the OLTP layer.
    - Not included in the sample implementation.
7. Internal Service Layer: Queries the OLAP layer to display real-time and aggregated analytics on dashboards.
    - Not included in the sample implementation.