Software Architecture Document

Live Cricket Score for Inter/Intra University Tournaments

Group 10:
Syed Abrar — CS22BTECH11058
Mohammed Gufran Ali — CS22BTECH11040
Ashwin Kumar — CE21BTECH11008
Paavaneeswar Reddy — CS22BTECH11014
Adil Salfi — CS20BTECH11031

March 15, 2025

Contents

Software System Architecture Overview

1.1 System Overview

The Live Cricket Score system is designed to provide real-time score updates, match statistics, and related tournament information for inter/intra university cricket events. It caters to multiple stakeholders and offers a platform where administrators, match organisers, players, umpires, and users (fans) can interact with live match data.

1.2 System Context

The system integrates with external entities such as:

- **Users/Fans:** Access live scores, commentary, and match statistics.
- **Match Organisers:** Input match details and manage real-time score updates.
- **Players & Umpires:** Access and update their profiles, performance metrics, and match-related data.
- **Administrators:** Oversee system operations, manage databases, and ensure data integrity.

The architecture is built around a central system database and secured API endpoints that mediate communication between the front-end interfaces (web/mobile) and the back-end services.

1.3 Stakeholders

Key stakeholders include:

- **Administrators** – Responsible for overall system management and data governance.
- **Match Organisers** – Handle match scheduling, live score updates, and tournament management.
- **Players and Umpires** – Maintain their profiles and access match-related information.
- **Users/Fans** – View live scores, match statistics, and tournament details.
- **Technical Team** – Develop, deploy, and maintain the system.

1.4 Scope of the Document

This document details the software architecture for the Live Cricket Score system. It covers:

1. A comprehensive architecture overview.
2. Detailed design including component views and event-driven microservices.
3. An ATAM (Architecture Tradeoff Analysis Method) analysis evaluating scenarios, non-functional requirements, tradeoffs, and sensitivities.

The design choices were informed by our Software Requirements Analysis (SRA), Software Requirements Specification (SRS), and related design documents.

Software System Architecture Design

2.1 View and Style

Our architecture adopts a **Component and Connector View** to clearly outline the interaction between different modules. We further implement an **Event-Driven Microservices Architecture** to ensure scalability, loose coupling, and resilience in handling real-time updates.

2.1.1 Component and Connector View

This view decomposes the system into discrete components and illustrates the connections between them using well-defined protocols and interfaces.

2.1.2 Event-Driven Microservices Architecture

The event-driven design supports asynchronous processing and real-time notifications. Events such as score updates and match status changes trigger microservice communications, ensuring minimal latency in delivering live data to end-users.

2.2 Architecture

2.2.1 Architecture Diagram

Below is the high-level architecture diagram represented by the image file `SA.drawio.png`:

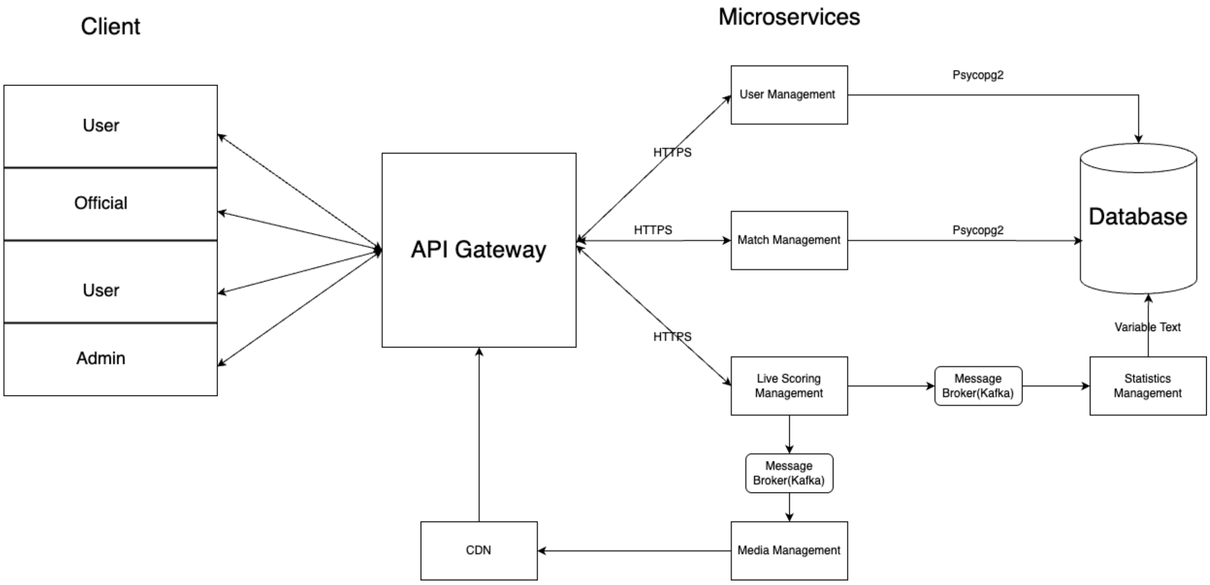

Figure 2.1: High-Level Architecture Diagram

2.2.2 Architecture Description

The architecture is modularized into independent microservices that communicate via an API Gateway and asynchronous messaging. This enables:

- **Scalability:** Individual services can be scaled independently based on load.
- **Resilience:** Service failures are isolated, minimizing system-wide impact.
- **Maintainability:** Modular design simplifies updates and debugging.

2.2.3 Components Table

Component	**Description**
User Management Service	Handles user related management including user authentication, email verification etc
Match Management Service	Manages match scheduling, team management, and match-related data processing.
Live Scoring Service	Processes real-time score updates, live statistics, and disseminates scores to end-users.

2.2.4 Connections Table

Connection	**Description**
HTTPS	Secure communication protocol for client-server interactions.
Message Broker	Facilitates asynchronous communication between microservices via event messages.
Psycopg2 Access	Interface connector used for database interactions (PostgreSQL) from various services.

2.3 Other Proposed Architectures Considered

Monolithic Architecture and **Traditional Client-Server Models** were evaluated:

- **Monolithic Approach:** Rejected due to limited scalability and increased difficulty in maintaining real-time updates.
- **Traditional Client-Server Model:** While simpler, it does not adequately support high-load and event-driven scenarios.

The current microservices-based, event-driven approach was chosen for its ability to scale horizontally and handle the real-time nature of live score updates.

ATAM Analysis

3.1 Scenarios

Key scenarios include:

- A user requests live score updates during high-traffic match events.
- A match organiser submits real-time score updates via the system.
- System recovery when one or more microservices fail.

3.2 Requirements/Constraints

The system must meet the following non-functional requirements:

- **Performance:** Sub-second response times for live updates.
- **Scalability:** Ability to handle spikes in user load during high-profile matches.
- **Security:** Robust authentication and data protection mechanisms.
- **Availability:** High uptime with redundancy across microservices.

3.3 Tradeoff Analysis

The tradeoffs considered include:

- **Complexity vs. Scalability:** Microservices increase system complexity but offer significant scalability benefits.
- **Latency vs. Resilience:** Asynchronous messaging minimizes latency while improving fault isolation.
- **Deployment Frequency vs. Integration Overhead:** Modular services allow for continuous deployment at the cost of more sophisticated integration and monitoring.

3.4 Evaluation of Non-Functional Attributes

The architecture was evaluated based on:

- **Response Time:** Ensuring rapid update cycles for live scoring.
- **Availability:** Redundant services and failover mechanisms.
- **Security:** End-to-end encryption and token-based authentication.
- **Maintainability:** Code modularity and clear API contracts between services.

3.5 Sensitivities and Tradeoffs

Critical sensitivities include:

- The system's ability to handle unexpected traffic spikes.
- Balancing rapid deployment with thorough integration testing.
- Maintaining consistency between microservices when data is updated in real time.

These factors guided the final design decisions and informed our tradeoff analysis during ATAM.