Technical Specification for Digital Transformation

For ABC Racing Company

Author: Jayasri Panneerselvam

Date:06/06/2025

Version:1.0

Index

🧩 Problem Statement

ABC Racing is losing its fan base. To revive interest, they want:

- A modern, engaging website and mobile platform.
- Fans across all ages and devices to access rich content (videos, news, photos).
- Fast, accessible experience globally.
- Personalized experience per region.
- Features like bookmarking, offline viewing, and analytics.

👷 Problem Breakdown – Technical Terminology

Area	Need
Platform	Responsive, single-page app (SPA) on mobile/web
Content	Rich multimedia (images/videos), live updates, articles
Customization	Regional color themes, localization (i18n)
Performance	Optimized load times on slow networks
Accessibility	AAA accessibility compliance
Compatibility	Graceful degradation for old browsers (e.g., IE7)
Scalability	Modular, scalable architecture for future commerce modules
Analytics	Deep user interaction tracking
Offline	Support for offline viewing of bookmarked sections

Assumptions for the Solution

1. **Transactions Per Second (TPS):**
    - The system is designed to handle a **peak TPS of 5000** (example) during major events or sales, with an average TPS of 1000 in regular operation.
    - Backend and database architecture are sized accordingly to handle burst traffic without degradation.
2. **Number of Users:**
    - The platform supports up to **1 million registered users** with **100,000 concurrent active users** during peak times.
    - User base growth expected at 20% annually, so solution is scalable to 2 million users in 2-3 years.

3. **Autoscaling:**
    - Cloud infrastructure uses **autoscaling groups or Kubernetes Horizontal Pod Autoscalers** triggered by CPU, memory, or custom metrics like request latency.
    - Autoscaling thresholds set to maintain response times below 300 ms under peak loads.
    - Minimum of 3 instances running at all times for high availability.
4. **Security:**
    - All data in transit encrypted using TLS 1.3.
    - Authentication via OAuth 2.0 and/or JWT tokens with short expiry for session management.
    - Role-based access control (RBAC) implemented for internal modules.
    - Regular vulnerability scanning and compliance with industry standards like OWASP Top 10.
    - Data at rest encrypted using AES-256.
    - DDoS protection and WAF enabled at network perimeter.
5. **Color/Frontend Theme & Fan Ageing:**
    - The UI theme adapts based on user demographic data — e.g., younger users see more vibrant colors and dynamic animations; older users have an option for high-contrast themes and larger fonts for accessibility.
    - Color palettes chosen to comply with WCAG AA/AAA standards for color contrast to support aging eyes and color vision deficiencies.
    - Accessibility features like adjustable font size, voice commands, and simplified layouts are included.
    - The frontend(web/mobile) supports **dynamic theming** (light/dark mode, custom color palettes) to enhance user experience.
    - Theming is user-configurable and persists across sessions.
6. **Fan Ageing & Offline Viewing**
    - **Fan Ageing:** (Assuming this refers to content popularity decay or user engagement over time)
    - Analytics service tracks content engagement and can trigger recommendations or archiving of stale content.
    - Caching strategies (Redis) are used to optimize access to trending content.
    - **Offline Viewing:**
    - Mobile clients support offline mode, caching content locally using secure storage.
    - Sync mechanisms ensure data consistency when the device reconnects.

- Content delivery is optimized for low bandwidth and intermittent connectivity.

7. **Personalize Data:**

   Utilizing users' demographic data

Summary Table

Area	Assumption
TPS	1,000–10,000+ at peak, scalable
Users	100,000+ concurrent, scalable
Autoscale	Enabled for stateless services, DB read replicas, Kafka/ElasticSearch clusters
Security	API auth, TLS, RBAC, data encryption, regular audits
Colorfront/Theme	Dynamic theming, accessibility, user-configurable
Fan Ageing	Analytics-driven content lifecycle, caching for trending, archiving stale content
Offline Viewing	Local caching on mobile, sync on reconnect, optimized for low bandwidth

---

Note: These are starting assumptions. For production, you should conduct load testing, security reviews, and user research to refine these numbers and features. If you need a more detailed breakdown or want to adjust any of these assumptions to achieve the strategic idea.

To ensure the architecture diagram fully represents these key technical considerations, we can incorporate the following elements:

1. Performance Optimization

- **CDN Integration (Cloudflare, AWS CloudFront)** → Improve content delivery speed.
- **Media Optimization Layer** → Includes WebP conversion, adaptive video streaming.
- **Lazy Loading & Code Splitting** → Reduce initial load time.
- **Caching Mechanism (Redis, Service Workers)** → Improve mobile experience over 3G/4G.

2. Accessibility (AAA Compliance)

- **WCAG Standards Layer** → Represent AAA-level accessibility.
- **UI Components Accessibility Framework** → Semantic HTML5, ARIA roles.

- **Screen Reader Support & Contrast Adjustments** → Improve usability for all audiences.

3. Graceful Degradation (Legacy Browser Support)

- **Polyfill Layer (Babel, ES6 Transpiler)** → Ensures compatibility with older browsers.
- **CSS Fallbacks** → Adapt styles gracefully.
- **Conditional Rendering** → Optimize content based on browser capabilities.

4. Scalability & Extensibility

- **Microservices Architecture** → Horizontal scaling via containerized backend services.
- **Load Balancing Layer (NGINX, AWS ALB)** → Distribute heavy traffic loads.
- **Modular API Gateway (GraphQL, RESTful Services)** → Ensure easy integration for future expansions.

5. Analytics Integration

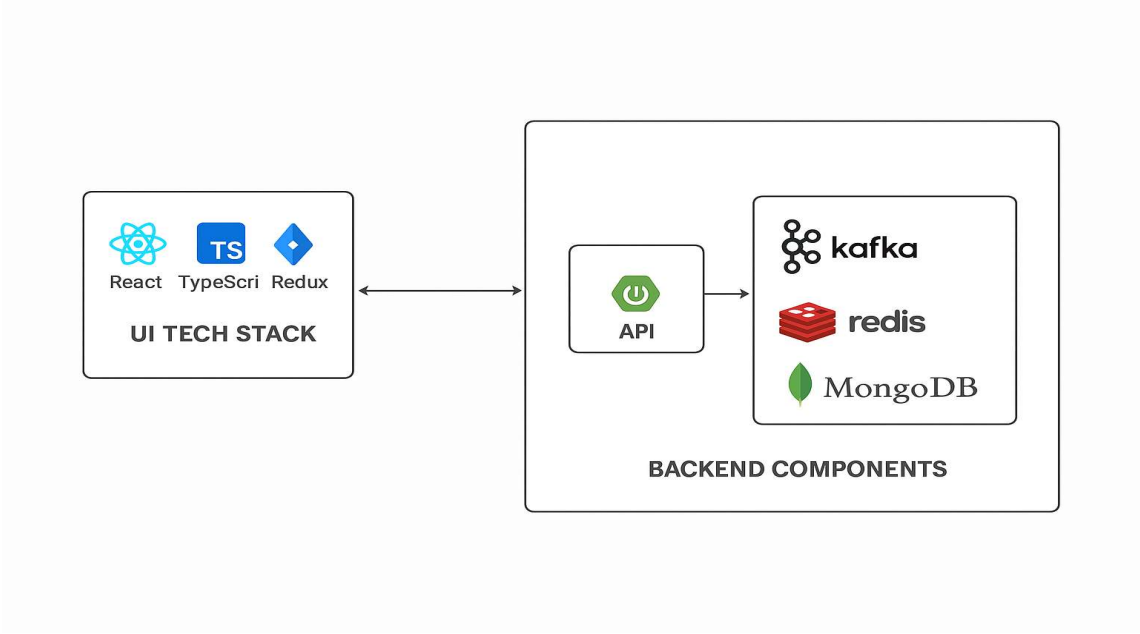
- **Event Tracking Layer (Google Analytics, Mixpanel, Segment)** → Capture user interactions.
- **Behavioral Data Pipeline (Kafka, AWS Kinesis)** → Process analytics efficiently.
- **A/B Testing Framework** → Optimize user experience dynamically.

6. Offline Viewing

- **Service Workers** → Enable caching for offline viewing.
- **PWA Implementation** → Support offline content through Progressive Web Apps.
- **Local Storage & IndexedDB** → Store bookmarked or cached sections.

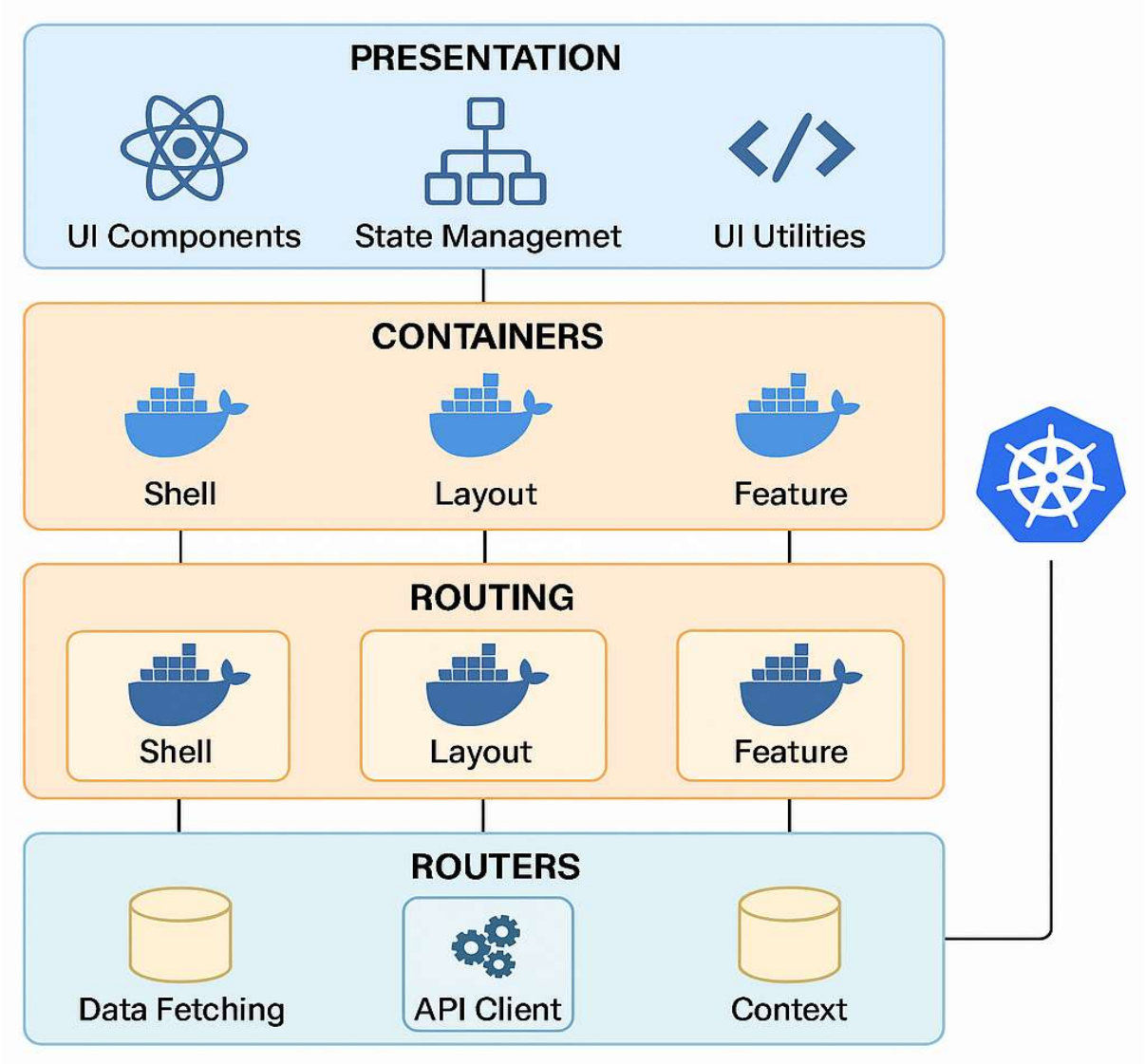
🧱 Technical Architecture

1.High Level design and Solution:



a. Frontend Architecture

- **Framework:** Use **React.js** (for flexibility) or **Next.js** (for improved performance via server-side rendering).
- **Design System:** Implement a **component-based architecture**, leveraging **Styled Components** or **Tailwind CSS** for adaptability.
- **Personalization:** Allow **geo-based theming** with dynamic styles.
- **Bookmarking Feature:** Use **local storage** or a lightweight **NoSQL** database (such as Firebase) for persisting user bookmarks.



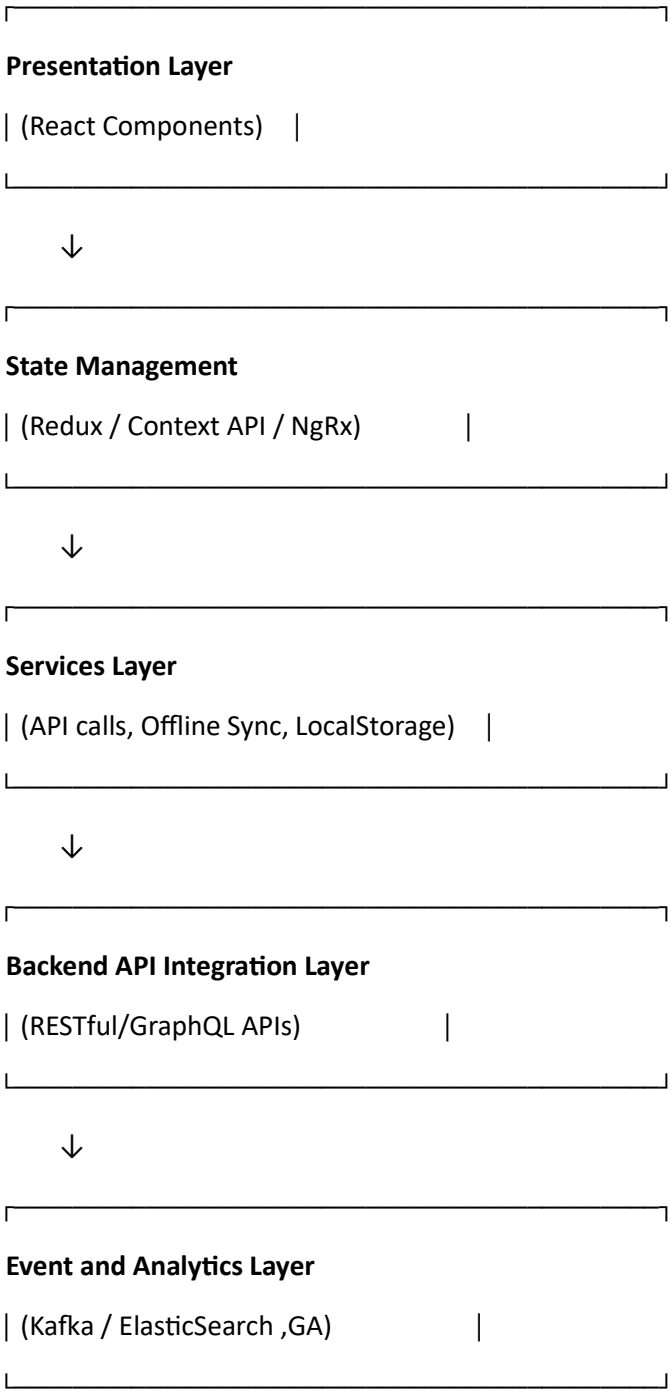
Back-End Architecture

- **Microservices Approach:** Deploy services using **Node.js (Express)** or **Spring Boot**, ensuring modularity.
- **Content Storage:** Use **MongoDB** (for flexible schema handling)
- **CDN & Caching:** Implement **AWS CloudFront** and **Redis** for optimized content delivery.

- **Event Streaming & Analytics:** Use **Kafka** and **GA** for computing interactive metrics.

(Logical View)

- **Type:** Modular, Component-based SPA (Single Page Application)
- **Framework:** React (with Next.js for SSR)
- **Structure:**



- **Key Characteristics**:
    - **Separation of Concerns**: Clear division between UI, state, and services
    - **Maintainability**: Components are modular and testable
    - **Extensibility**: Easy to plug new features like “Live Scores” or “E-Commerce”

---

2. Infrastructure and DevOps

- **CI/CD Pipeline:** Use **GitHub Actions** or **Jenkins** for automated deployments.
- **Containerization:** Deploy via **Docker** and orchestrate with **Kubernetes** for scalability.
- **Testing & Code Quality:** Use **Cypress** (UI testing), **Jest/Mocha** (unit tests), and **SonarQube** for code coverage.

Area	Tool
CI/CD	GitHub Actions / Jenkins
Code Quality	SonarQube
Unit Testing	Jest (React)
E2E Testing	Cypress / Playwright
Code Coverage	Istanbul / Codecov/SonarQube
Containerization	Docker
Deployment	Kubernetes on AWS/GCP (EKS/GKE)
Monitoring	Prometheus + Grafana

---

3. Performance Strategy

- **Lazy Loading** of videos/images
- **Server-Side Rendering** (with Next.js) for faster first paint
- **Progressive Web App (PWA)** support for offline content
- **CDN delivery** (Akamai/Cloudflare)
- **Gzip + Brotli compression**
- **Image optimization** using WebP
- **Lazy Loading & Code Splitting:** Improve load times using **Webpack’s dynamic imports**.

- **Media Optimization:** Convert images to **WebP**, videos to **adaptive bit-rate streaming (HLS)**.
- **Load Balancing:** Use **NGINX** or **AWS ALB** to distribute traffic.

---

4. Localization Strategy

- Use **react-intl or ngx-translate or i18next** for multi-language support
- Auto-detect language based on user's browser
- Dynamic theme switching using CSS variables
- Geo-detection via IP for regional content/themes
- **Geo-Based Customization:** Dynamically adjust colors and content based on user location via **GeoIP API**.

---

5. Tech Stack

Layer	Technology	Reason
Frontend	React (Next.js)	Component-based SPA with SSR support
Backend	Node.js + Express / Spring Boot (optional)	Scalable, RESTful APIs
API Management	Apollo Server / GraphQL / REST	Flexibility in querying
DB	MongoDB (unstructured)	Hybrid data needs
Cache	Redis	Faster read ops (bookmarks, configs)
Search	ElasticSearch	Quick full-text search
CDN	Cloudflare / Akamai	Fast static content delivery
Offline	Service Workers	Cache and sync content offline
Analytics	GA, Segment, Custom Kafka Events	Detailed interaction metrics
Accessibility	axe-core, Lighthouse	AAA compliance testing
Testing	Jest, Mocha, Cypress	High confidence, automated
State Management	Redux/Context API	Efficient handling of app state

Infrastructure	AWS/GCP/Azure	Cloud-native hosting

6. Sample Feature

Vertical Code (Bookmarking System)

On the frontend (React + Redux Toolkit):

tsx

```
// store/bookmarksSlice.ts
import { createSlice } from '@reduxjs/toolkit';

const bookmarksSlice = createSlice({
  name: 'bookmarks',
  initialState: [],
  reducers: {
    addBookmark: (state, action) => { state.push(action.payload); },
    removeBookmark: (state, action) => state.filter(item => item.id !== action.payload),
  }
});

export const { addBookmark, removeBookmark } = bookmarksSlice.actions;
export default bookmarksSlice.reducer;
```

Backend (Node.js):

js

```
// routes/bookmark.js
app.post('/api/bookmark', (req, res) => {
  const { userId, contentId } = req.body;
  db.saveBookmark(userId, contentId);
  res.send({ success: true });
});
```

Service

```
// public/sw.js
```

```
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open('v1').then(cache =>
      cache.addAll([
        '/',
        '/index.html',
        '/styles.css',
        '/offline.html'
      ])
    )
  );
});

self.addEventListener('fetch', event => {
  event.respondWith(
    fetch(event.request).catch(() => caches.match(event.request))
  );
});
```

server.js

```
// server.js
const express = require('express');
const app = express();
const PORT = 5000;

const articles = [
  { id: 1, title: "Race Day Highlights", summary: "Top moments from the weekend." },
  { id: 2, title: "Behind the Pit Wall", summary: "A look at the team strategy." }
];

app.get('/api/articles', (req, res) => {
```

```
  res.json(articles);
});

app.listen(PORT, () => {
  console.log(Server running on http://localhost:${PORT});
});

// HomePage.jsx
import React, { useEffect, useState } from "react";
import axios from "axios";

const HomePage = () => {
  const [articles, setArticles] = useState([]);

  useEffect(() => {
    axios.get("/api/articles")
      .then(res => setArticles(res.data))
      .catch(err => console.error(err));
  }, []);

  return (
    <div className="p-6">
      <h1 className="text-2xl font-bold">Latest Racing Content</h1>
      <ul>
        {articles.map(article => (
          <li key={article.id} className="border p-2 my-2">
            <h2 className="text-lg font-semibold">{article.title}</h2>
            <p>{article.summary}</p>
          </li>
        ))}
      </ul>
```

```
  </div>
 );
};

export default HomePage;
```

snippet for dynamic UI theming based on user's **geo-location**:

```
const getTheme = (region) => { const themes = { US: { primary: "#FF0000", secondary: "#000000" }, EU: { primary: "#0000FF", secondary: "#FFFFFF" }, ASIA: { primary: "#00FF00", secondary: "#333333" } }; return themes[region] || themes["US"]; }; // Apply theme dynamically const region = userGeoLocation.region; const currentTheme = getTheme(region); document.body.style.backgroundColor = currentTheme.primary;
```

👥 7. Scrum Team Structure

Role	Skills
Product Owner	Domain knowledge (Sports Media), Story grooming
Scrum Master	Agile facilitator
UI/UX Designer	Figma, Accessibility, Responsive Design
Frontend Developers (3)	React, Redux, PWA, Accessibility
Backend Developers (2)	Node.js/Spring Boot, API Design, DB
QA Engineers (2)	Manual + Automation (Cypress/Jest)
DevOps Engineer	CI/CD, Monitoring, Infra
Content Strategist	SEO, Tagging, Metadata
Localization Expert	i18n, region-specific assets
Data Analyst	GA, dashboards, insights
Business Analyst	Requirement Analysis, Agile Planning

💡 **Innovation Idea – Fan Reconnect Feature**

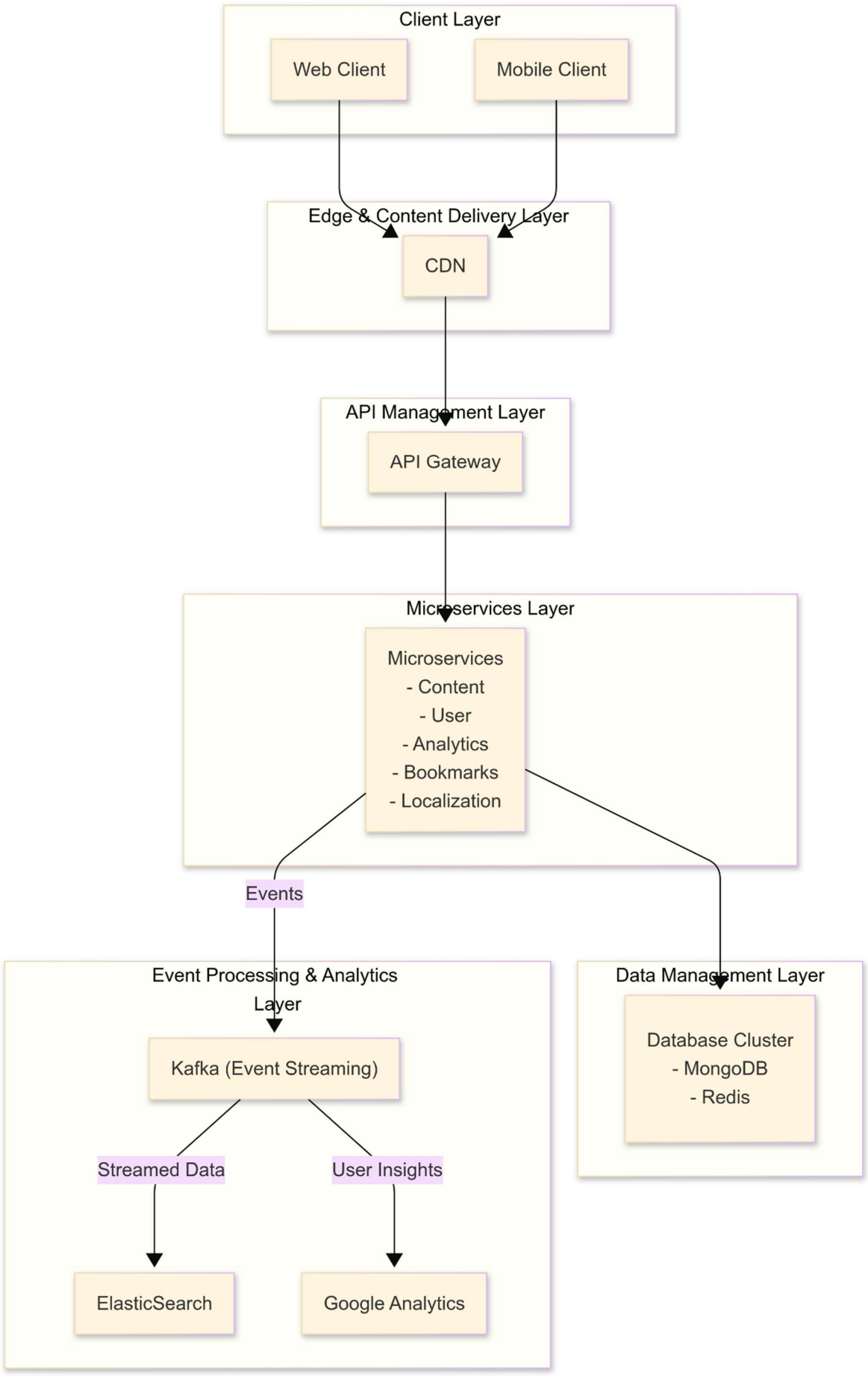
"Virtual Pitstop"
A gamified experience where fans can:

- Predict race outcomes
- Win digital badges or race points
- Share leaderboards with friends
- Unlock exclusive content (behind-the-scenes, interviews)

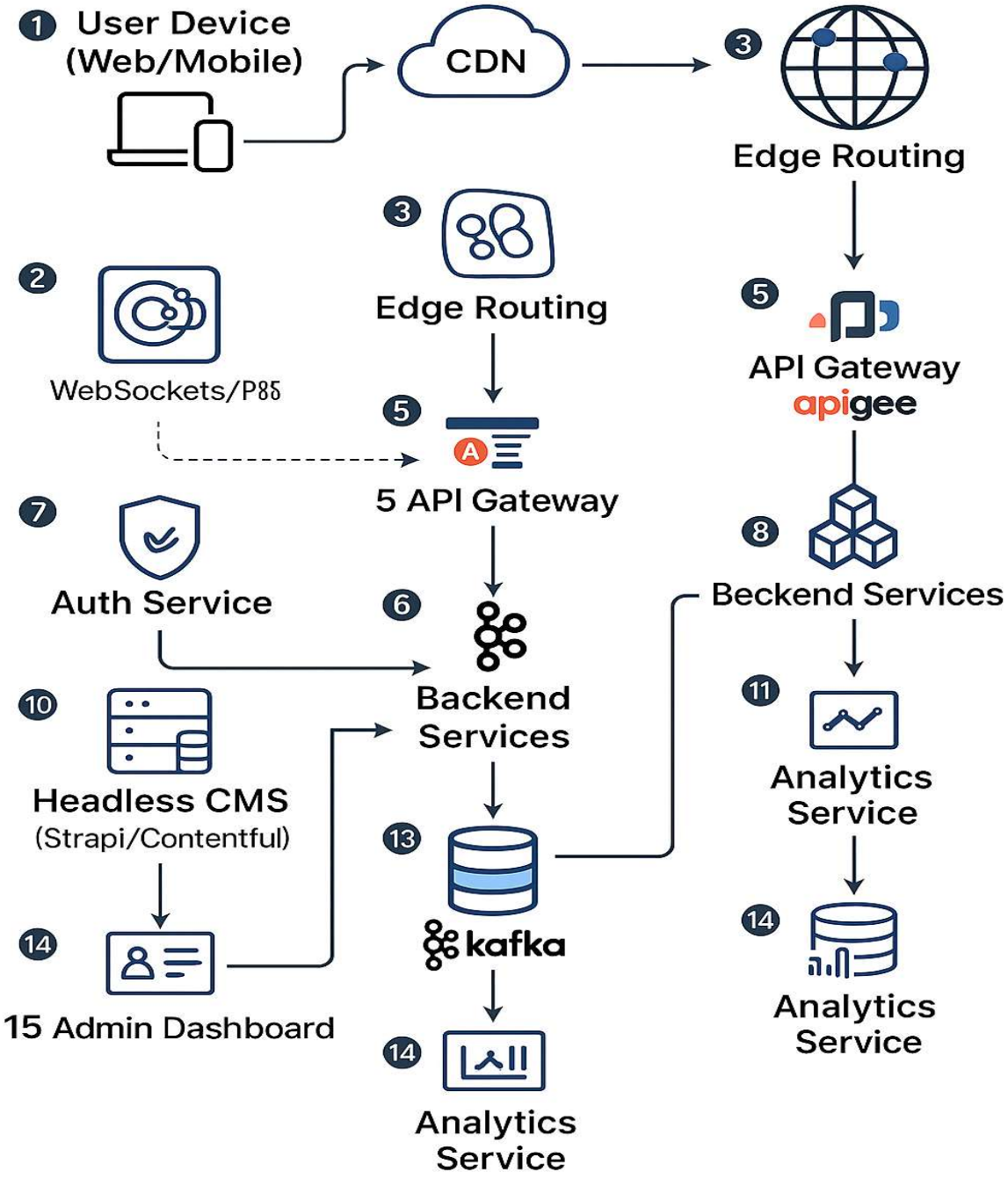
✅ Increases engagement
✅ Creates a social, sticky fan experience
✅ Can later be monetized with merchandise and sponsor tie-ins

Component Interaction diagram: -

Component interaction diagram is mainly integrating all the requested elements such as **performance optimization, accessibility compliance, graceful degradation, scalability, event processing and analytics tracking, offline viewing, localization, CI/CD**, and the recommended **tech stack**


Client Layer
Web Client
Mobile Client
Edge & Content Delivery Layer
CDN
API Management Layer
API Gateway
Microservices Layer
Microservices
- Content
- User
- Analytics
- Bookmarks
- Localization
Events
Event Processing & Analytics Layer
Kafka (Event Streaming)
Streamed Data
User Insights
ElasticSearch
Google Analytics
Data Management Layer
Database Cluster
- MongoDB
- Redis

This **end-to-end solution** ensures **high performance, scalability, localization, and accessibility**, perfectly addressing ABC Racing Company's business challenges.



Here's a step-by-step explanation of the end-to-end architecture based on the **sequence numbers and components** in the diagram:

🔁 End-to-End Architecture Breakdown (Modern Web/Mobile Application)

1️⃣ User Device (Web/Mobile)

- Users interact via **Progressive Web App (PWA)** or **Native Mobile Apps** (iOS/Android).
- Devices initiate requests like content fetching, login, and updates.

2️⃣ CDN (Content Delivery Network)

- Static assets (HTML, CSS, JS, Images, Videos) are served from **CDNs** like **Cloudflare/Akamai**.
- Reduces latency and improves performance by delivering content closer to users.

3️⃣ Edge Routing

- Traffic is intelligently routed using **GeoDNS and Edge Functions**.
- CDN + Edge Routing ensures requests are directed to the nearest and fastest location.
- Improves reliability and regional failover.

4️⃣ Frontend App (React/Next.js, React Native/Flutter)

- Frontend apps communicate with the server via **API Gateway**.
- Handles user interactions, displays personalized content, and integrates WebSockets for real-time data (e.g., notifications, live updates).

5️⃣ API Gateway (Apigee)

- Central entry point for all API requests.
- Provides **authentication**, **rate limiting**, **logging**, and **routing**.
- Connects frontend clients to backend microservices securely and efficiently.

6️⃣ Backend Services

- Responsible for domain-specific logic:
    - User Management
    - Content Delivery
    - Recommendations
    - Notification Dispatch

- Built to be **stateless**, deployed in containers (e.g., Docker), and orchestrated via **Kubernetes**.

---

7 Auth Service

- Performs authentication & authorization using **OAuth2/JWT**.
- Validates identity and roles before accessing protected APIs.
- Integrates with API Gateway to block unauthorized traffic.

---

8 Backend Services Interaction

- All backend services interact through APIs or **event-driven communication (Kafka)**.
- Data persistence via **MongoDB**, Redis (for caching).
- Stateless, ensuring horizontal scalability.

---

9 Analytics Service

- Backend emits tracking and telemetry data to analytics service.
- Tracks:
  - Page views
  - User behavior
  - Feature usage
  - Errors
- Ensures product and performance insights.

---

10 Headless CMS (Strapi/Contentful)

- Used by editors/admins to manage content (articles, banners, metadata).
- Exposes APIs consumed by frontend/backend for rendering and personalization.

---

11 Analytics Service (GA / Custom)

- Aggregates metrics and events, transforms them into structured insights.
- Data pipelines may use tools like **Kafka**, **Spark**, or **ElasticSearch**.

---

12 External APIs

- Backend may call 3rd-party services like:

- Payment Gateways
- Weather APIs
- Location/Map Services
- Social Media Integrations

1️⃣3️⃣ Kafka (Event Streaming Platform)

- Microservices emit **events** to **Apache Kafka** topics.
- Enables:
  - Decoupled service communication
  - Asynchronous processing
  - Real-time pipelines for analytics and personalization

1️⃣4️⃣ Analytics Service (Kafka + Dashboard)

- Event data from Kafka is processed and visualized in dashboards.
- Helps business stakeholders monitor KPIs and app performance in real time.

1️⃣5️⃣ Admin Dashboard

- Admins use this UI to:
  - Manage users, content, permissions
  - Monitor logs and usage
  - Trigger backend operations (e.g., reindex content, publish features)

✅ Additional Notes:

- **WebSockets/PubSub** enables push-based updates from server to client (notifications, live updates).
- The entire stack is typically **containerized (Docker)** and deployed using **Kubernetes**.
- Each service is independently deployable (**Microservices Architecture**), allowing **resilience, scalability, and maintainability**.
