Propuesta de Diseño y Arquitectura de Software

Sistema "Click Race"

Alexander Belisario T.

Contenido

1. Introducción

La presente propuesta detalla el diseño y la arquitectura de sistema "Click Race", un juego multijugador en tiempo real donde la velocidad de clic de cada participante determina su avance en una barra de progreso. El objetivo principal es construir un sistema altamente escalable, reactivo y tolerante a fallos, capaz de gestionar un gran volumen de interacciones simultáneas y proporcionar una experiencia de juego fluida y sincronizada para todos los jugadores.

2. Requisitos Clave

Para asegurar el éxito de "Click Race", se han identificado los siguientes requisitos funcionales y no funcionales:

2.1. Requisitos Funcionales

- En el proceso de registro y autenticación de jugadores, los usuarios deben poder crear una cuenta y autenticarse para participar en el juego.
- Para la creación y unión a partidas, los jugadores deben poder iniciar nuevas carreras o unirse a partidas existentes.
- Cada clic del jugador debe incrementar su avance en una barra de progreso individual.
- El progreso de todos los jugadores en una partida debe ser visible y actualizado en tiempo real para todos los participantes.
- En cuanto a la detección y notificación de ganador, el sistema debe detectar al primer jugador que alcanza la meta y notificar a todos los participantes de la victoria.
- El historial de partidas y los puntajes de los jugadores deben ser persistidos, como garantía de la persistencia de datos.

2.2. Requisitos No Funcionales

El sistema debe cumplir con una serie de requisitos no funcionales que garanticen su óptimo desempeño, escalabilidad, disponibilidad, seguridad y mantenibilidad. A continuación, se detallan las principales consideraciones:

En cuanto a rendimiento, para asegurar una experiencia fluida, el sistema debe ofrecer actualizaciones de progreso con una latencia percibida inferior a 100 ms. Asimismo, debe contar con la capacidad de procesar un alto volumen de clics por segundo en cada partida, optimizando tiempos de respuesta y carga.

La arquitectura debe ser capaz de soportar un gran número de jugadores concurrentes, permitiendo la ejecución simultánea de múltiples partidas sin afectar la estabilidad del sistema. Además, todos los servicios clave (backend, base de datos, caché y procesamiento de eventos) deben ser escalables horizontalmente, garantizando un crecimiento dinámico acorde con la demanda.

Se requiere un nivel de disponibilidad del 99.9%, asegurando continuidad operativa del servicio de juego. Para ello, el sistema debe ser resiliente, permitiendo la tolerancia a fallos de componentes individuales sin que esto genere interrupciones en la prestación del servicio.

La protección de la información y la integridad del juego son fundamentales. Se deben implementar mecanismos robustos de autenticación y autorización, asegurando una gestión segura de usuarios. Asimismo, el sistema debe contar con estrategias anti-trampas, integrando validaciones en el backend que permitan mitigar el uso de herramientas como auto-clickers o la manipulación indebida de eventos.

En términos de mantenibilidad y operabilidad, para facilitar la administración y evolución del sistema, se debe garantizar un código modular y bien documentado, promoviendo la claridad y facilidad de modificación. Adicionalmente, los procesos de despliegue, monitoreo y actualización deben estar optimizados para reducir la complejidad operativa y agilizar mejoras futuras.

3. Drivers Arquitectónicos y Restricciones

Las decisiones arquitectónicas clave se guían por los siguientes drivers y restricciones:

- La necesidad de una sincronización de clics y progreso con baja latencia es el driver principal, lo que impulsa el uso de WebSockets y un procesamiento de eventos eficiente.
- El potencial de alta cantidad de jugadores simultáneos exige una arquitectura sin estado (stateless) en los servicios de juego, el uso de cache distribuido y una base de datos escalable.
- La capacidad de añadir más recursos de forma elástica es fundamental para el crecimiento, lo que favorece el uso de microservicios y orquestación con Kubernetes, conforme a la escalabilidad horizontal.
- La tolerancia a fallos y la recuperación rápida son cruciales para la experiencia del usuario, requiriendo bases de datos replicadas y sistemas de mensajería tolerantes a fallos.

4. Vistas Arquitecturales

La arquitectura propuesta se basa en el modelo **4+1 Vistas de Kruchten**, que permite abordar el diseño desde diferentes perspectivas: lógica, de procesos, de desarrollo, física y de escenarios.

4.1. Vista Lógica

La vista lógica se enfoca en la funcionalidad del sistema, los objetos y sus relaciones, sin considerar aspectos de implementación o de despliegue. Aquí se definen los componentes clave que encapsulan la lógica de negocio del juego.

Componentes Lógicos Principales

Player	Representa a un jugador individual con atributos como id, name, currentProgress, gameSessionId
GameSession	Gestiona una instancia específica del juego. Incluye sessionId, status (e.g., waiting, inProgress, finished), players (lista de Players), targetProgress, winnerId.
ClickEvent	Objeto que representa un clic realizado por un jugador, incluyendo playerId, timestamp, clickCount (para agrupar clics).
ProgressBar	Entidad lógica que representa el avance de un jugador en la sesión actual.
GameCoordinator	Componente central de la lógica de negocio que orquesta las sesiones de juego, procesa los ClickEvents, actualiza el progreso de los jugadores y determina el ganador.
AuthService	Maneja la autenticación y autorización de los jugadores.
DataService	Abstracción para la persistencia de datos (interactúa con MongoDB).
CacheService	Abstracción para el almacenamiento en caché de datos (interactúa con Redis).
RealtimeService	Gestiona la comunicación en tiempo real con los clientes (WebSockets, WebRTC).

Diagrama de Clases (Conceptual)

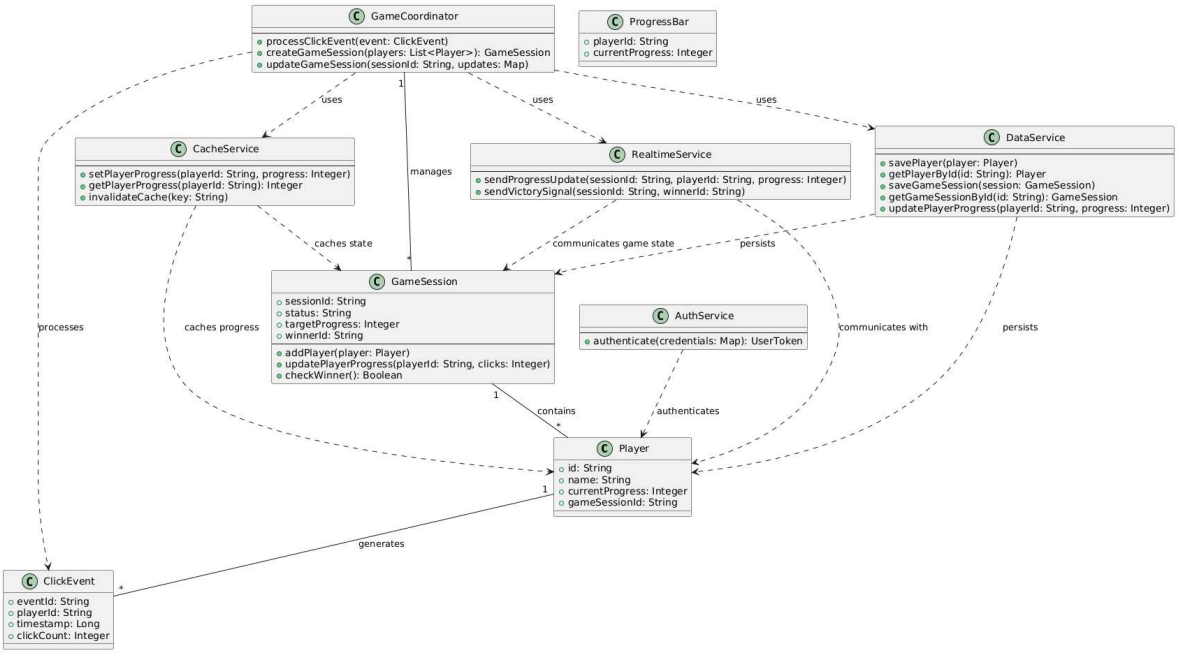

Figura 1. Diagrama de clases (conceptual)

4.2. Vista de Procesos

La vista de procesos describe la concurrencia del sistema, la distribución de los componentes, la comunicación entre ellos y cómo se manejan los hilos y los procesos. A continuación, se muestra el flujo de ejecución en tiempo real.

Flujo de Interacción Clave: "Jugador hace clic"

Este diagrama de secuencia ilustra el flujo de un clic de un jugador desde el frontend hasta la actualización de la base de datos y la notificación en tiempo real a los demás jugadores.

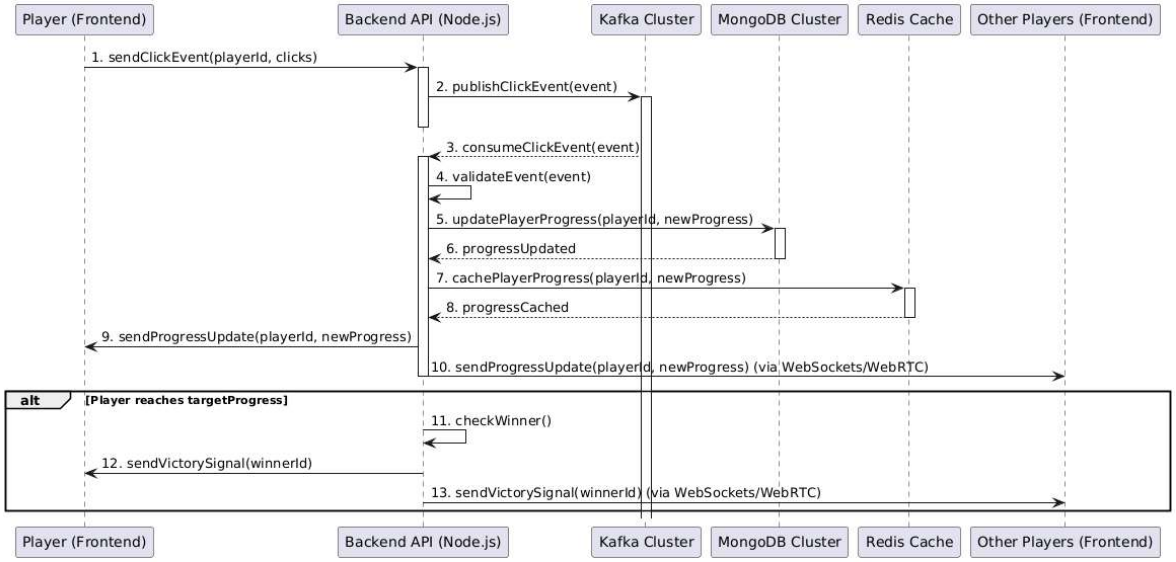

Figura 2. Diagrama de procesos

4.3. Vista de Desarrollo

La vista de desarrollo se centra en la organización del código fuente, los módulos, las librerías y las dependencias. Establece cómo el equipo de desarrollo organizará y construirá el software.

Componentes de Desarrollo

El sistema está conformado por múltiples componentes diseñados para garantizar un funcionamiento eficiente y escalable. A continuación, se presentan las principales tecnologías y módulos empleados en cada capa:

Frontend Application	Desarrollado con HTML, CSS y JavaScript, incorporando Phaser.js para la lógica del juego y renderizado gráfico. Incluye módulos clave como GameScene (gestión visual), NetworkManager (manejo de WebSockets/WebRTC), UIManager (interfaz de usuario) e InputHandler (procesamiento de clics). Se utilizan librerías como socket.io-client para WebSockets y PeerJS para WebRTC, en caso de comunicación directa desde el cliente.
Backend API	Implementado en Node.js y Express.js, estructurado para soportar API REST y WebSockets mediante ws o socket.io. Sus módulos incluyen GameController (gestión de solicitudes de juego), EventProcessor (procesamiento de eventos Kafka), PlayerService (lógica de jugadores), GameSessionService (manejo de sesiones), DatabaseAdapter (interfaz con MongoDB), CacheAdapter (interacción con Redis) y consumidores/productores Kafka. Se integran librerías como mongoose (ODM para MongoDB), ioredis (cliente Redis) y kafkajs (gestión de Kafka).
Kafka Cluster	Compuesto por Kafka Brokers y ZooKeeper, garantizando la coordinación y distribución eficiente de eventos. Los principales tópicos utilizados incluyen click-events (registro de interacciones de jugadores) y game-updates (sincronización de estados de juego).
MongoDB Cluster	Diseñado bajo una arquitectura Replica Set, con un Primary Node y múltiples Secondary Nodes para alta disponibilidad. Contiene colecciones esenciales como players (información de jugadores) y gameSessions (estado y progreso de partidas).
Redis Cache	Utiliza instancias en Cluster Mode, permitiendo escalabilidad. Se emplean estructuras de datos como Strings (almacenamiento del progreso de los jugadores) y Hashes (gestión de estados de sesión).
WebRTC Module	Facilita la comunicación peer-to-peer entre jugadores para sincronización de eventos en tiempo real, reduciendo la carga en el backend. Su funcionamiento requiere un servidor STUN/TURN, aunque no se especifica explícitamente como un componente independiente.

Diagrama de Componentes

Este diagrama ofrece una vista estructurada de los módulos clave del sistema y sus interacciones. Muestra la integración entre frontend, backend, persistencia y comunicación, destacando su modularidad, escalabilidad y eficiencia. Cada componente cumple un rol específico para garantizar un funcionamiento óptimo y mantenible.

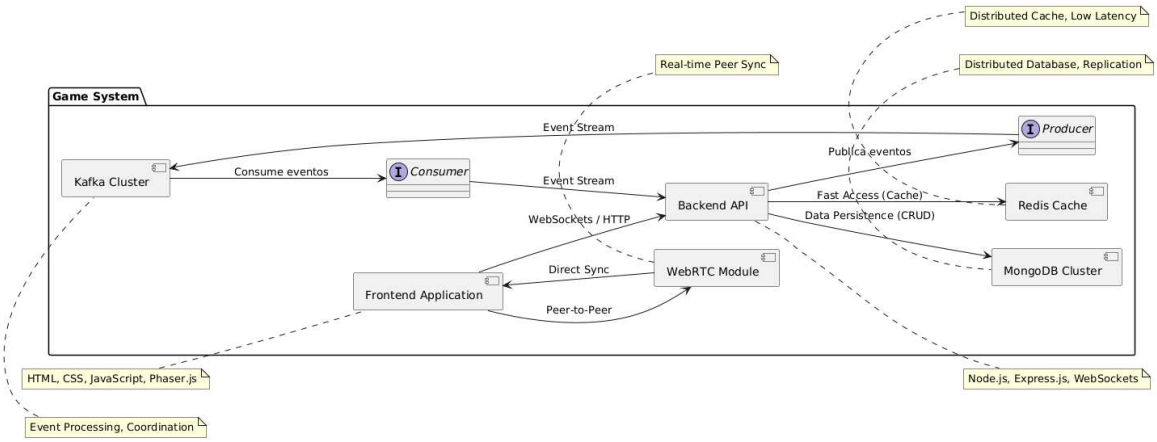

Figura 3. Diagrama de componentes

4.4. Vista Física

La vista física describe el mapeo de los componentes de software a la infraestructura de hardware. Se enfoca en la topología de la red, la ubicación física de los servidores y cómo se distribuyen los procesos en ellos.

Infraestructura General

La infraestructura general del sistema está diseñada para garantizar eficiencia, escalabilidad y disponibilidad. En el núcleo se encuentra el **clúster de Kubernetes**, responsable de la orquestación de microservicios, la gestión del escalado y la administración de recursos. Dentro del clúster, los **Master Nodes** supervisan la operación y aseguran la estabilidad, mientras que los **Worker Nodes** ejecutan los pods que alojan las aplicaciones.

Para optimizar el rendimiento y la distribución de tráfico, se incorporan **Load Balancers**, permitiendo un enrutamiento eficiente entre los servicios de frontend y backend. Asimismo, la infraestructura de **red** proporciona conexiones de alta velocidad, facilitando la comunicación entre los distintos nodos y servicios, garantizando una integración estable y fluida en el ecosistema del sistema.

Distribución de Componentes en Kubernetes

La arquitectura propuesta se basa en la implementación de diversos componentes dentro de un entorno Kubernetes, con el objetivo de garantizar escalabilidad, alta disponibilidad y eficiencia operativa.

Frontend Pods	Contenedores responsables de servir la aplicación cliente, compuesta por HTML, CSS y JavaScript. Se recomienda el uso de Nginx o un servidor HTTP ligero dentro del pod para optimizar la distribución de los recursos estáticos.
Backend API Pods	Contenedores diseñados para ejecutar la aplicación basada en Node.js. Se implementarán múltiples réplicas con el propósito de mejorar la disponibilidad del servicio y facilitar su escalabilidad.
Kafka Pods	Conjunto de contenedores destinados a la operación de los Kafka Brokers y sus dependencias, incluyendo ZooKeeper. Se sugiere un despliegue stateful para garantizar la persistencia y estabilidad de la plataforma de mensajería.
MongoDB Pods	Contenedores que alojan los nodos del Replica Set de MongoDB. Al igual que con Kafka, se recomienda un despliegue stateful para preservar la integridad de los datos y asegurar la continuidad del servicio.
Redis Pods	Contenedores dedicados a la gestión de las instancias de Redis, con posibilidad de configuración en modo clúster para optimizar la distribución de la carga y mejorar el rendimiento del sistema de almacenamiento en memoria.

Diagrama de Despliegue (Físico)

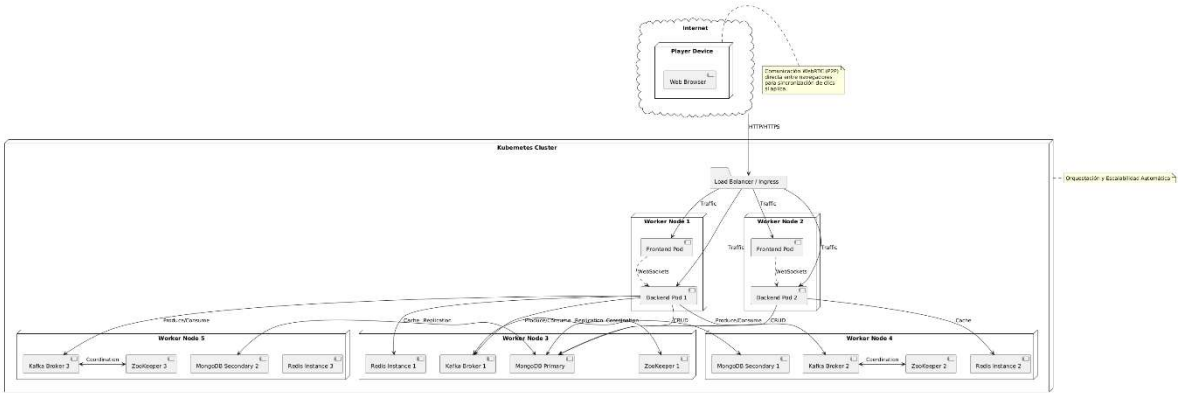

Figura 4. Diagrama de despliegue

4.5. Vista de Escenarios

La vista de escenarios describe los casos de uso más importantes del sistema, sirviendo como una validación de las otras vistas arquitecturales. Aquí se detallan las interacciones clave que impulsan el diseño.

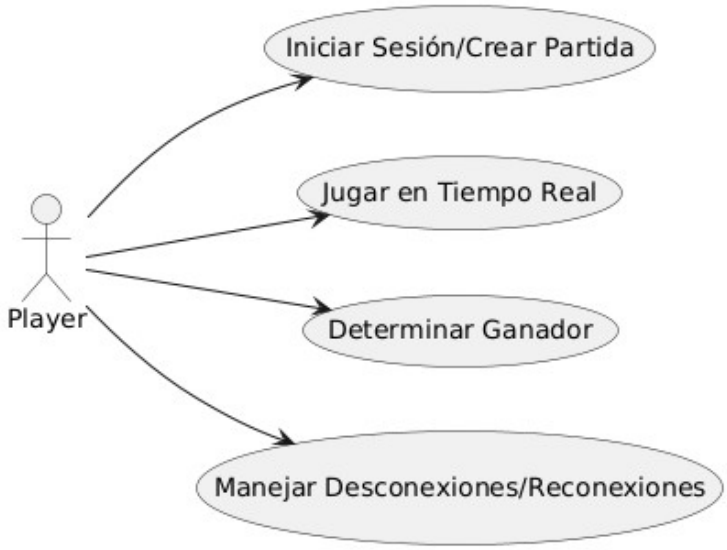

Figura 5. Diagrama de Casos de Uso

1. Inicio de Sesión y Creación/Unión a Partida:
    - Un jugador accede a la aplicación frontend.
    - (Opcional) Se autentica o se le asigna un ID de invitado.
    - El jugador elige crear una nueva sesión de juego o unirse a una existente.
    - El frontend envía una solicitud al Backend API.
    - El Backend API gestiona la sesión, la registra en MongoDB y la cachea en Redis.
    - El RealtimeService notifica a los jugadores en la sesión sobre nuevos participantes.
2. Juego en Tiempo Real (Flujo de Clics):
    - Un jugador hace clic en la interfaz del juego.
    - El frontend genera un ClickEvent y lo envía al Backend API.
    - El Backend API publica el ClickEvent en el tópico click-events de Kafka.
    - Los consumidores del Backend API en Kafka procesan el evento, validan el clic y actualizan el currentProgress del jugador en MongoDB.
    - El nuevo progreso se cachea en Redis para lecturas rápidas.
    - El RealtimeService (WebSockets/WebRTC) envía la actualización de progreso a todos los jugadores en la misma sesión.
    - El frontend de cada jugador actualiza la ProgressBar correspondiente en tiempo real.
3. Determinación del Ganador:
    - Cuando un ClickEvent procesado hace que un jugador alcance el targetProgress, el GameCoordinator en el Backend API detecta esta condición.
    - El GameCoordinator marca la sesión como finished en MongoDB y Redis.
    - El RealtimeService envía una victorySignal a todos los jugadores de la sesión.
    - El frontend de los jugadores muestra el mensaje de victoria y el ID del ganador.

4. Manejo de Desconexiones y Reconexiones:
    - Si un jugador se desconecta, el RealtimeService lo detecta.
    - El Backend API puede actualizar el estado del jugador en la sesión (e.g., disconnected).
    - Si el jugador se reconecta, el frontend intenta restablecer la conexión WebSocket/WebRTC.
    - El Backend API recupera el último estado del jugador y la sesión desde Redis/MongoDB y lo envía al frontend para una sincronización rápida.

5. Justificación de la Tecnología Seleccionada

La selección de tecnologías para la implementación de *Click Race* responde a criterios clave como escalabilidad, rendimiento, tolerancia a fallos y experiencia de usuario en tiempo real. A continuación, se presentan los fundamentos de cada elección tecnológica:

Frontend (HTML, CSS, JavaScript, Phaser.js)

Se han adoptado tecnologías web estándar para garantizar la compatibilidad con una amplia gama de navegadores y la creación de interfaces interactivas. *Phaser.js* ha sido seleccionado como framework para el desarrollo del entorno gráfico y la lógica del juego, optimizando la gestión de escenas y la renderización para mejorar el rendimiento en el navegador.

Backend (Node.js, Express.js, WebSockets)

Se ha optado por *Node.js* debido a su capacidad para manejar un alto volumen de conexiones concurrentes de manera eficiente, gracias a su arquitectura no bloqueante y orientada a eventos. *Express.js* facilita la implementación de la API REST y la gestión de solicitudes HTTP, mientras que *WebSockets* permite la comunicación en tiempo real entre el servidor y los clientes, garantizando una transmisión de datos bidireccional con baja latencia, crucial para la dinámica del juego.

Mensajería (Kafka)

Kafka ha sido incorporado como sistema de mensajería distribuido para desacoplar la API del backend de los consumidores de eventos, optimizando la escalabilidad y la tolerancia a fallos. Además, permite gestionar grandes volúmenes de eventos de manera eficiente y confiable, asegurando que los datos no se pierdan en caso de fallos en el sistema.

Base de Datos (MongoDB)

Se ha elegido *MongoDB* por su flexibilidad y capacidad de escalabilidad. Su modelo de documentos se adapta adecuadamente a la representación de datos relacionados con jugadores y sesiones de juego. La base de datos soporta replicación y particionamiento (*sharding*), lo que permite su expansión horizontal y garantiza la disponibilidad y eficiencia en el manejo de grandes volúmenes de información.

Caché (Redis)

Redis ha sido seleccionado como sistema de almacenamiento en memoria para reducir la carga sobre la base de datos, ofreciendo tiempos de respuesta más rápidos para datos de acceso frecuente, como el progreso de los jugadores y el estado de las sesiones de juego. Su capacidad para realizar operaciones de lectura y escritura de alta velocidad lo convierte en una opción idónea para la gestión de datos en tiempo real.

Orquestación de Contenedores (Kubernetes)

Se ha implementado *Kubernetes* para automatizar el despliegue, escalado y gestión de los contenedores de la aplicación. Esta tecnología facilita la adopción de una arquitectura basada en microservicios, mejorando la modularidad, escalabilidad y mantenibilidad del sistema.

Comunicación Peer-to-Peer (WebRTC)

WebRTC se considera una opción viable para la comunicación directa entre navegadores, reduciendo la latencia y la carga del servidor al permitir la transferencia de datos entre los jugadores sin intermediación. Aunque su implementación supone un mayor grado de complejidad, su integración puede mejorar significativamente la experiencia en tiempo real dentro del juego.