UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"MODELADO, SIMULACIÓN Y OPTIMIZACIÓN DE UNA ARQUITECTURA CLIENTE-SERVIDOR PARA SISTEMAS DE EVENTOS DISCRETOS"

TRABAJO FIN DE GRADO

Junio – 2026

AUTOR: Adrián Cánovas Acosta

DIRECTOR/ES: Vicente Quiles Zamora

RESUMEN

Este Trabajo de Fin de Grado presenta el diseño, implementación y evaluación empírica de una arquitectura cliente-servidor distribuida aplicada a un sistema de simulación de eventos discretos. El caso de estudio es un motor de combate por turnos asíncrono. Se ha comparado el enfoque de cómputo en cliente frente a un servidor autoritativo, optando por este último por sus garantías de seguridad, coherencia de datos y escalabilidad. El cliente visual, desarrollado en Unity (C#), recoge la entrada del usuario y reproduce los resultados sin ejecutar ninguna lógica de simulación, la cual reside íntegramente en microservicios *serverless* sobre Firebase Cloud Functions (Node.js). El motor implementa un algoritmo determinista basado en un generador de números pseudoaleatorios con semilla, un sistema de Medidor de Turno proporcional a la velocidad de las entidades y funciones de escalado asintótico para el cálculo de atributos secundarios. Los datos se almacenan en Cloud Firestore, organizado en colecciones independientes para optimizar el rendimiento de lectura. Los resultados empíricos muestran una latencia estabilizada de 475 ms de RTT, un *payload* medio de 3,8 KiB por combate completo, y validan la integridad del sistema frente a vulnerabilidades en el lado del cliente, confirmando la superioridad en seguridad de esta arquitectura frente a alternativas descentralizadas.

Palabras clave: Simulación de eventos discretos, arquitectura autoritativa, Firebase serverless, sistemas distribuidos asíncronos, determinismo algorítmico.

ABSTRACT

This Bachelor's Thesis presents the design, implementation and empirical evaluation of a distributed client-server architecture applied to a discrete-event simulation system. The case study is an asynchronous turn-based combat engine. Client-side computation was compared against an authoritative server model; the latter was selected for its security guarantees, data consistency and scalability. The visual client, built in Unity (C#), captures user input and renders results without executing any simulation logic, which resides entirely in serverless microservices deployed on Firebase Cloud Functions (Node.js). The engine implements a deterministic algorithm based on a seeded pseudo-random number generator, a Turn Meter system proportional to each entity's speed, and asymptotic scaling functions for secondary combat attribute calculation. Data is stored in Cloud Firestore, structured into independent collections to optimise read performance. Empirical results show a stabilised RTT of 475 milliseconds, an average response payload of 3.8 KiB per complete combat, and effective resistance to client-side memory manipulation techniques, confirming that the main engineering objectives of this work have been met.

Keywords: Discrete-event simulation, authoritative architecture, Firebase serverless, asynchronous distributed systems, algorithmic determinism.

ÍNDICE DE CONTENIDOS

Contenido

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGOS

GLOSARIO DE ACRÓNIMOS

- **API**: Application Programming Interface (Interfaz de Programación de Aplicaciones). Conjunto de subrutinas, funciones y protocolos que expone un sistema para permitir la comunicación estandarizada con otro software.
- **BaaS**: Backend as a Service (Backend como Servicio). Modelo de computación en la nube donde el proveedor gestiona toda la infraestructura del servidor, permitiendo a los desarrolladores centrarse exclusivamente en la lógica de cliente.
- **Cold Start**: (Arranque en frío). Latencia inicial que se produce en arquitecturas serverless cuando la plataforma debe aprovisionar, inicializar y arrancar un nuevo contenedor para ejecutar una función bajo demanda.
- **CORS**: Cross-Origin Resource Sharing (Intercambio de Recursos de Origen Cruzado). Mecanismo de seguridad de los navegadores web que permite o restringe los recursos solicitados desde un dominio distinto al del servidor de origen.
- **DDoS**: Distributed Denial of Service (Ataque Distribuido de Denegación de Servicio). Ataque cibernético que busca inhabilitar un servidor, servicio o red sobrecargándolo con un flujo masivo de tráfico de Internet malicioso.
- **DES / SED**: Discrete Event Simulation (Simulación de Eventos Discretos). Rama de la modelización computacional donde el estado de un sistema evoluciona de forma discontinua, alterándose únicamente en instantes discretos de tiempo debido a la ocurrencia de eventos.
- **DEVS**: Discrete Event System Specification (Especificación de Sistemas de Eventos Discretos). Formalismo matemático estricto y modular para el modelado y simulación de sistemas de eventos discretos.
- **DOI**: Digital Object Identifier (Identificador de Objeto Digital). Enlace permanente y estandarizado utilizado para identificar unívocamente artículos académicos y publicaciones científicas en la red.

- **DTO**: Data Transfer Object (Objeto de Transferencia de Datos). Patrón de diseño arquitectónico utilizado para encapsular y transportar datos entre procesos o componentes, reduciendo el número de llamadas a métodos remotos.
- **FaaS**: Functions as a Service (Funciones como Servicio). Categoría de servicios en la nube que permite a los desarrolladores ejecutar fragmentos de código aislados (funciones) en respuesta a eventos sin aprovisionar ni gestionar servidores.
- **HTTP**: Hypertext Transfer Protocol (Protocolo de Transferencia de Hipertexto). Protocolo base de la World Wide Web para la transmisión de información, que define la sintaxis y semántica que utilizan el cliente y el servidor en sus comunicaciones.
- **IETF**: Internet Engineering Task Force (Grupo de Trabajo de Ingeniería de Internet). Organización internacional de estandarización abierta que desarrolla y promueve los estándares voluntarios de Internet.
- **JSON**: JavaScript Object Notation (Notación de Objetos de JavaScript). Formato de texto ligero para el intercambio de datos, basado en la sintaxis de JavaScript, altamente legible por humanos y fácilmente interpretable por máquinas.
- **JWT**: JSON Web Token. Estándar abierto de la IETF (RFC 7519) que define una forma compacta y autónoma de transmitir información segura entre partes mediante objetos JSON firmados digitalmente.
- **MiB**: Mebibyte. Unidad de información equivalente a 2^{20} (1.048.576) bytes. Se utiliza en informática para medir memoria con precisión binaria, diferenciándose del Megabyte (MB) decimal estándar.
- **NoSQL**: Not Only SQL (Sistemas de bases de datos no relacionales). Paradigma de sistemas de gestión de bases de datos documentales, orientadas a grafos o clave-valor, que prescinden del modelo tabular relacional clásico para priorizar la flexibilidad y el escalado horizontal.
- **P2P**: Peer-to-Peer (Red entre iguales). Arquitectura de red distribuida en la que los distintos nodos (clientes) actúan como emisores y receptores de información simultáneamente, sin depender de un servidor centralizado.

- **PEM**: Presupuesto de Ejecución Material. En la dirección de proyectos de ingeniería, representa el coste estricto y directo de los materiales, hardware, licencias y mano de obra necesarios para la ejecución física de un proyecto.
- **PRNG**: Pseudorandom Number Generator (Generador de Números Pseudoaleatorios). Algoritmo diseñado para generar secuencias de números que se aproximan a las propiedades del azar puro, utilizando una fórmula matemática dependiente de una semilla inicial determinista.
- **REST**: Representational State Transfer (Transferencia de Estado Representacional). Estilo de arquitectura de software para sistemas hipermedia distribuidos, caracterizado por el uso de métodos HTTP estándar y la ausencia de estado en el servidor (statelessness).
- **RFC**: Request for Comments (Petición de Comentarios). Serie de memorandos técnicos publicados por el IETF que describen métodos, comportamientos, investigaciones y estándares técnicos de Internet.
- **RTT**: Round-Trip Time (Tiempo de ida y vuelta). Latencia total que experimenta un paquete de red desde que es emitido por el cliente, procesado por el servidor, hasta que el cliente recibe la confirmación o respuesta.
- **SDK**: Software Development Kit (Kit de Desarrollo de Software). Conjunto de herramientas, bibliotecas, documentación y ejemplos de código proporcionados por un proveedor para facilitar el desarrollo de aplicaciones sobre una plataforma o servicio específico.
- **UI**: User Interface (Interfaz de Usuario). Punto de interacción física y lógica entre el usuario humano y el sistema informático, englobando los elementos visuales que facilitan el control de la aplicación.
- **UID**: User Identifier (Identificador de Usuario). Cadena alfanumérica única generada por el sistema para identificar de forma inequívoca e inmutable a una entidad o usuario dentro de una base de datos.
- **UML**: Unified Modeling Language (Lenguaje Unificado de Modelado). Estándar internacional y visual en ingeniería de software utilizado para especificar, construir y documentar la estructura de un sistema informático.

- **UX**: User Experience (Experiencia de Usuario). Conjunto de factores relativos a la interacción del usuario con un sistema, determinando su nivel de satisfacción, facilidad de uso, eficiencia y percepción de valor.
- **Warm Start**: (Arranque en caliente). Ejecución de una función serverless que reutiliza un entorno y contenedor previamente instanciados y mantenidos en memoria por la plataforma, eliminando el coste de latencia del aprovisionamiento.

CAPÍTULO 1. INTRODUCCIÓN

1.1. Motivación del proyecto

El crecimiento de las aplicaciones interactivas, especialmente los videojuegos para móvil y web, ha cambiado lo que se le exige al backend que las sostiene. Los dispositivos móviles se han convertido en la plataforma principal de consumo digital, y con ello ha aparecido un ecosistema donde millones de usuarios esperan respuestas rápidas, datos coherentes y experiencias que no fallen ante una mala conexión o un intento de trampa.

En ese contexto, diseñar backends eficientes, seguros y escalables ya no es algo exclusivo de los grandes estudios. La elección del modelo arquitectónico afecta al rendimiento, al coste operativo, a la seguridad y a la capacidad del sistema de crecer junto con el producto.

Este Trabajo de Fin de Grado parte de ese problema: no solo hacer funcionar un prototipo, sino diseñar, implementar y evaluar empíricamente una arquitectura que cumpla con los requisitos técnicos de una aplicación interactiva distribuida real.

1.2. Objetivos

El objetivo principal de este trabajo es el siguiente:

Diseñar, modelar y evaluar empíricamente una arquitectura cliente-servidor distribuida para sistemas de simulación basados en eventos discretos, aplicada al desarrollo de un prototipo funcional de combate por turnos.

Para alcanzar este objetivo principal, se han definido los siguientes objetivos específicos:

Desarrollar un cliente visual en Unity (un motor gráfico multiplataforma y entorno de desarrollo para aplicaciones interactivas en 2D y 3D) que implemente comunicación asíncrona con async/await y actúe como visor puro de los resultados generados por el servidor, sin ejecutar lógica de simulación propia.

Diseñar y desplegar un conjunto de microservicios *serverless* en Node.js sobre Firebase Cloud Functions que centralicen la lógica de simulación, incorporando mecanismos de autenticación, control de entornos, prevención de abusos y escritura en base de datos NoSQL.

Instrumentar el sistema para extraer métricas empíricas (tiempos de ejecución en servidor, latencia RTT y tamaño de *payload*) que permitan evaluar cuantitativamente el rendimiento de la arquitectura.

1.3. Alcance y especificaciones

El alcance del proyecto comprende el diseño e implementación completos de la arquitectura de backend y su integración con un cliente Unity funcional. El sistema abarca la definición del modelo de datos en Firestore, el protocolo de comunicación HTTP REST entre cliente y servidor, el motor de simulación por eventos discretos con soporte para simulaciones probabilísticas reproducibles, y los mecanismos de seguridad asociados a la gestión de identidad y la protección de los endpoints.

El prototipo no pretende ser un producto comercial terminado, sino un entorno de pruebas sobre el que validar las decisiones de diseño planteadas. Quedan fuera del alcance de este trabajo el desarrollo de un sistema de emparejamiento entre jugadores en tiempo real (*matchmaking*), la implementación de un sistema de monetización y el diseño gráfico completo de la interfaz de usuario, aspectos que corresponderían a fases posteriores del prototipo.

1.4. Justificación académica

Este Trabajo de Fin de Grado se enmarca en el Grado en Ingeniería Electrónica y Automática Industrial y se apoya en dos competencias centrales de la titulación, aplicadas a un problema de ingeniería real y medible.

La primera es el modelado y simulación de sistemas. El motor de simulación por eventos discretos que forma el núcleo de este trabajo requiere formalizar matemáticamente el estado del sistema mediante variables primarias y secundarias, diseñar un algoritmo de control de flujo basado en el Medidor de Turno e implementar un generador pseudoaleatorio determinista con semilla algorítmica. El sistema de

combate no se ha construido como una colección de reglas arbitrarias, sino como un modelo matemático explícito cuyo comportamiento es analizable, reproducible y verificable.

La segunda es la informática industrial y las comunicaciones. El diseño de la API REST, la definición del protocolo de *payloads* y cabeceras HTTP, la autenticación con JWT y la extracción de telemetría de red son tareas directamente relacionadas con los contenidos de comunicaciones industriales y sistemas distribuidos del plan de estudios. La instrumentación del sistema para obtener métricas empíricas añade además una dimensión experimental alineada con la metodología propia de la ingeniería.

Que ambas competencias confluyan en un mismo proyecto es lo que dota de rigor y coherencia académica a este Trabajo Fin de Grado, al trasladar los conocimientos teóricos de la titulación a la resolución de un problema de ingeniería aplicado y medible.

Por último, vale la pena señalar que los sistemas de eventos discretos son el fundamento matemático de los autómatas de estados finitos y los sistemas SCADA de plantas industriales. Un sistema interactivo cliente-servidor es formalmente equivalente a una celda de fabricación con recursos limitados y colas de prioridad: entidades compitiendo por un recurso, transiciones de estado deterministas y un reloj que avanza por eventos. La diferencia está en el dominio de aplicación, no en el modelo matemático.

2. ESTADO DEL ARTE Y MARCO TEÓRICO

Este capítulo recoge la base teórica sobre la que se asienta el proyecto, junto con un repaso al estado actual de las tecnologías empleadas: arquitecturas distribuidas, técnicas de simulación matemática y herramientas de despliegue en la nube. Entender el punto en el que se encuentra cada una de estas áreas resulta imprescindible para justificar las decisiones de diseño tomadas en el desarrollo del sistema de resolución asíncrona de eventos discretos.

2.1. Arquitecturas de sistemas distribuidos interactivos

2.1.1. Computación local frente a topología Cliente-Servidor

Cuando se diseña un sistema distribuido interactivo, una de las primeras preguntas que hay que responder es quién asume la carga del cálculo: ¿el cliente o el servidor? En un extremo están las arquitecturas de computación local (Client-Side Computation) y las redes Peer-to-Peer (P2P), donde cada nodo ejecuta su propia lógica y sincroniza los resultados con el resto. A primera vista puede parecer una solución atractiva, ya que descarga la infraestructura central, pero en la práctica plantea problemas difíciles de ignorar.

El más evidente es la falta de un árbitro externo. Si la simulación corre en el dispositivo del usuario, este tiene acceso directo al estado interno del sistema y puede alterarlo en su beneficio, lo que se conoce como cheating o trampa. A esto se suma que, cuando dos clientes ejecutan la misma lógica de forma independiente, sus estados tienden a diferenciarse con el tiempo, lo que compromete tanto la equidad como la reproducibilidad de los resultados. No se trata de vulnerabilidades teóricas: están ampliamente documentadas en la literatura sobre sistemas de juego en red [1].

Por todo ello, la topología dominante en la industria es la Cliente-Servidor, donde el servidor actúa como única fuente de verdad (*Source of Truth*). En este esquema, los clientes no tienen autoridad sobre el estado global; su papel se reduce a enviar peticiones y representar la información que devuelve el servidor. Esta separación elimina buena parte de los vectores de ataque presentes en las arquitecturas descentralizadas y garantiza que todos los participantes trabajen sobre un estado coherente y verificable [1].

La Tabla 1 resume las diferencias más relevantes entre los tres paradigmas considerados en este proyecto.

Tabla 1. Comparativa técnica de arquitecturas de sistemas distribuidos interactivos.

Parámetro / Arquitectura	***Client-Side Computation***	***Peer-to-Peer* (P2P)**	**Servidor Autoritativo (*Stateless*)**
Seguridad (*Anti-Cheat*)	Muy baja	Baja	Muy Alta
Fuente de la verdad	Cliente local	Consenso/ Host	Servidor Central
Uso de red (Ancho banda)	Bajo	Alto (Malla completa)	Muy bajo (Petición única)
Coste de infraestructura	Nulo	Nulo	Medio (Pago por uso/ FaaS)

2.1.2. Arquitectura de Servidor Autoritativo

Dentro de la topología Cliente-Servidor, el modelo más robusto para aplicaciones de eventos discretos es el Servidor Autoritativo sin estado (Stateless Authoritative Server). En este modelo, el cliente solo envía peticiones que describen las acciones del usuario, sin incluir ningún estado de simulación. El servidor recibe esas peticiones, ejecuta toda la lógica de simulación de forma determinista y devuelve al cliente un registro de eventos (*event log*) con el resultado.

Que el procesamiento sea determinista es un requisito clave: ante el mismo conjunto de entradas, el servidor debe producir siempre la misma secuencia de salida, independientemente de cuándo se procese la petición. Esto permite mantener la consistencia del sistema incluso sin sincronía constante entre cliente y servidor, lo que encaja bien con modalidades de juego asíncronas por turnos [2].

En cuanto a la seguridad, esta arquitectura resuelve los problemas descritos en la sección anterior. Si no hay lógica ejecutable en el cliente, no hay nada que manipular. Además, los eventos generados por el servidor pueden almacenarse y reproducirse, lo que facilita verificar el comportamiento del sistema ante cualquier duda [1].

En términos de red, la comunicación se reduce a mensajes compactos de petición y respuesta, sin necesidad de sincronización continua de estado, lo que reduce el consumo de ancho de banda respecto a otras arquitecturas [2].

En definitiva, la arquitectura autoritativa es el modelo de referencia para sistemas distribuidos interactivos que necesitan combinar seguridad, consistencia y eficiencia. Es también el fundamento sobre el que se apoya el diseño del sistema desarrollado en este trabajo, cuya implementación sobre Unity y Firebase se detalla en los capítulos siguientes.

2.2. Modelado y Simulación de Eventos Discretos (SED)

2.2.1. Conceptos fundamentales de SED

La Simulación de Eventos Discretos (*Discrete-Event Simulation*, DES) es una técnica de modelización computacional que representa sistemas cuyo estado no cambia de forma continua, sino a saltos. Esto la distingue claramente de la simulación continua: mientras que esta última recurre a ecuaciones diferenciales y necesita evaluar el estado del sistema en cada instante, en la SED el modelo permanece congelado entre dos eventos y solo se actualiza cuando ocurre algo relevante [3]. La consecuencia práctica es importante: el coste computacional no depende de cuánto tiempo se simula, sino de cuántos eventos se procesan. Para sistemas con actividad esporádica, esto supone una ventaja considerable.

La literatura clásica identifica tres elementos básicos sobre los que se construye cualquier modelo de SED. Las entidades son los objetos activos de la simulación, los que cambian de estado, interactúan entre sí o consumen recursos [4]. El estado del sistema es, en esencia, el conjunto mínimo de variables que describen el modelo en un momento dado; cualquier modificación en esas variables es, por definición, el resultado de un evento. Y el reloj de simulación es simplemente una variable interna que lleva la

cuenta del tiempo lógico, saltando de un evento al siguiente sin desperdiciar ciclos en los intervalos sin actividad [3].

Vale la pena mencionar también el formalismo DEVS (*Discrete Event System Specification*), que ofrece un marco matemático para especificar este tipo de sistemas de forma modular. Su utilidad reside en que permite dividir un simulador complejo en componentes más pequeños con interfaces bien definidas, lo que facilita tanto su implementación en código como la verificación independiente de cada parte [5].

2.2.2. Aplicación a sistemas de resolución asíncrona

La SED no es exclusiva de la ingeniería industrial. Puede aplicarse a cualquier sistema que pueda describirse como una secuencia ordenada de sucesos discretos, y un sistema de combate por turnos encaja en esa definición con bastante naturalidad.

La correspondencia entre el modelo teórico y el sistema implementado es bastante directa. Los personajes son las entidades: objetos activos con atributos propios sobre los que recaen las consecuencias de cada acción [4]. El estado del sistema lo forman las variables que definen a esos personajes en cada momento; puntos de vida, estadísticas de ataque y defensa, modificadores activos y similares, y permanece invariable entre dos acciones consecutivas, lo que mantiene el carácter discreto de los turnos. Los eventos son las propias acciones: un ataque, una esquiva, el uso de un recurso o la aplicación de un efecto. Cada uno transforma el estado según reglas predefinidas y puede, a su vez, desencadenar nuevos eventos.

Más allá de la descripción, este enfoque tiene consecuencias concretas para el diseño del sistema. Adoptar un modelo formal como DEVS permite descomponer la lógica del simulador en unidades funcionales independientes, cada una responsable de un subconjunto de eventos [5]. Esto resulta especialmente útil en arquitecturas *serverless*, donde las funciones se ejecutan de forma *stateless*: cada invocación recibe el estado actual, procesa los eventos del turno y devuelve el resultado, sin necesidad de mantener contexto entre llamadas. En definitiva, el modelado mediante SED no solo clarifica la lógica del combate, sino que actúa como guía de diseño para su implementación en la arquitectura autoritativa descrita en la sección 2.1.

2.3. Tecnologías de la Información y Comunicaciones (TIC) en la nube

2.3.1. Backend as a Service (BaaS): Firebase

Con los años, la computación en la nube ha evolucionado hacia capas más abstractas, dejando que los proveedores asuman la mayor parte de las tareas de infraestructura. El modelo BaaS (*Backend as a Service)* lleva esa idea al extremo en lo que al servidor se refiere: el aprovisionamiento de hardware, la configuración de red, el sistema operativo y la disponibilidad del servicio dejan de ser responsabilidad del equipo de desarrollo.

Firebase, la plataforma BaaS de Google, ejecuta este modelo con un conjunto integrado de servicios gestionados: bases de datos en tiempo real, autenticación de usuarios, almacenamiento de archivos y funciones de computación bajo demanda, todo accesible mediante SDKs multiplataforma.

El efecto práctico es que el equipo puede centrarse en la lógica de la aplicación sin dedicar tiempo al mantenimiento de servidores. Para proyectos con recursos limitados, o con picos de actividad irregulares como los que presenta una aplicación interactiva, esto supone una ventaja real frente a una infraestructura propia (*on-premise*).

2.3.2. Bases de datos documentales NoSQL (Firestore)

Las bases de datos relacionales nacieron en un contexto donde la consistencia transaccional y la integridad referencial eran las prioridades. Funcionan bien en muchos escenarios, pero las aplicaciones distribuidas modernas les exigen cada vez más: mayor flexibilidad estructural y capacidad de escalar horizontalmente, algo que los sistemas relacionales no siempre resuelven de forma eficiente.

Las bases de datos NoSQL documentales responden a esa necesidad almacenando la información en documentos semiestructurados, habitualmente JSON, agrupados en colecciones sin esquema fijo.

Cloud Firestore, la solución documental de Firebase, aplica este modelo con un enfoque orientado al desarrollo de aplicaciones. Su arquitectura *multi-tenant* aísla proyectos entre sí y absorbe ráfagas de tráfico de forma automática, distribuyendo la carga sobre la infraestructura de Google sin que el desarrollador tenga que intervenir [7]. Eso lo convierte en una opción razonable para sistemas con demanda variable e impredecible.

Ahora bien, no todo son ventajas. El modelo de facturación de Firestore se basa en el número de lecturas, escrituras y eliminaciones, independientemente del tamaño de los documentos. Un esquema mal diseñado puede disparar lecturas innecesarias, generar consultas no indexadas o concentrar demasiadas escrituras sobre un mismo documento, los llamados *hotspots*, con el consiguiente aumento de latencia y coste [8]. El diseño del modelo de datos no es, por tanto, una decisión menor.

2.3.3. Microservicios y funciones Serverless (Cloud Functions)

El modelo FaaS (*Functions as a Service*), encuadrado dentro del enfoque *serverless*, lleva la abstracción un paso más allá. El desarrollador despliega funciones independientes que el proveedor ejecuta bajo demanda en respuesta a eventos, sin servidores que aprovisionar ni mantener. La plataforma se encarga del ciclo de vida completo: escala cuando hay carga y desactiva las instancias cuando no las necesita.

Cloud Functions for Firebase es la implementación FaaS del ecosistema Firebase. Permite asociar funciones a eventos de la plataforma, escrituras en Firestore, peticiones HTTP, llamadas desde el cliente, y proporciona una lógica de servidor reactiva y completamente gestionada. El encaje con la arquitectura autoritativa de la sección 2.1 es natural: cada petición de resolución de combate invoca una función que recibe el estado, ejecuta la simulación y almacena el resultado, sin mantener nada entre llamadas.

Dicho esto, FaaS tiene sus limitaciones. La más conocida es el *Cold Start*: cuando no hay ninguna instancia activa, la plataforma necesita crear un nuevo contenedor antes de poder atender la petición, lo que introduce una latencia inicial apreciable. A esto se añade el fenómeno *Spawn Start*, identificado en literatura reciente como un estado intermedio entre el arranque en frío y el *Warm Start*, en el que parte de los recursos ya están inicializados pero el entorno aún no está listo del todo [6]. Dependiendo del entorno, el tamaño del despliegue y las dependencias, esta sobrecarga puede oscilar entre varios cientos de milisegundos y algunos segundos.

En una aplicación de tiempo real, un juego de acción, por ejemplo, eso sería inaceptable. Pero en un sistema de simulación asíncrona por turnos como el que se desarrolla en este trabajo, el compromiso es perfectamente asumible. El usuario no percibe el arranque en frío como un problema, porque la resolución del combate no ocurre en tiempo real sino en diferido. A cambio, el sistema gana una capacidad de

escalado prácticamente ilimitada, un modelo de costes proporcional al uso y cero cargas operativas por gestión de servidores.

2.4. Protocolos de comunicación industrial e informática

2.4.1. API REST y estructura de datos JSON frente a SDKs integrados

La comunicación entre componentes en sistemas distribuidos modernos requiere la adopción de protocolos y estilos arquitectónicos que garanticen interoperabilidad, mantenibilidad y eficiencia en el intercambio de información. En este contexto, el estilo arquitectónico REST (*Representational State Transfer*) es el estándar habitual para la comunicación telemática entre clientes y microservicios. Definido formalmente por Roy Fielding en su tesis doctoral [9], REST no constituye un protocolo en sí mismo, sino un conjunto de restricciones arquitectónicas aplicadas sobre el protocolo HTTP que, cuando se respetan, dotan al sistema de propiedades relevantes como la ausencia de estado en el servidor (*statelessness*), la uniformidad de la interfaz y la capacidad de evolución independiente de cliente y servidor [9]. Cada recurso del sistema queda identificado mediante una URI única, y las operaciones sobre él se expresan a través de los métodos HTTP: GET para la consulta, POST para la creación, PUT o PATCH para la actualización y DELETE para la eliminación. Esta uniformidad facilita el mantenimiento del código y favorece la integración con herramientas de monitorización, prueba y documentación estándar.

El formato de serialización predominante en las APIs REST modernas es JSON (*JavaScript Object Notation*), definido como estándar técnico por el IETF. JSON representa estructuras de datos mediante pares clave-valor y arrays anidados en texto plano codificado en UTF-8, lo que lo hace simultáneamente legible por humanos e interpretable por máquinas sin necesidad de preprocesamiento [10]. Su ligereza respecto a alternativas previas como XML, su compatibilidad nativa con los lenguajes de programación más extendidos y su adecuación al intercambio asíncrono de cargas útiles (*payloads*) de tamaño variable lo convierten en el mecanismo de serialización óptimo para sistemas de simulación asíncrona donde los mensajes deben transportar estados de combate compactos y bien estructurados [10].

El núcleo de la discusión en este apartado reside, no obstante, en la elección entre dos aproximaciones técnicas para implementar la comunicación cliente-servidor: el uso de

SDKs (*Software Development Kits*) comerciales integrados en el cliente, o la implementación de un cliente REST manual sobre HTTP. Plataformas como Firebase ofrecen SDKs nativos para entornos como Unity que encapsulan la totalidad de las operaciones de red (autenticación, serialización, reintentos automáticos, gestión de caché y sincronización en tiempo real) detrás de interfaces de alto nivel. Esta abstracción acelera el desarrollo y reduce el código repetitivo (*boilerplate*), constituyendo una opción razonable en proyectos donde la velocidad de iteración prima sobre el control operativo.

Sin embargo, esa comodidad tiene un precio. Los SDKs actúan como cajas negras: ocultan al desarrollador el tráfico de red real que genera la aplicación. Los reintentos ante fallos, la invalidación de caché, la multiplexación de conexiones o la compresión de datos los gestiona la biblioteca internamente, sin exponer ninguna métrica al exterior. En la práctica, esto significa que no es posible medir con precisión parámetros como el *Round-Trip Time* (RTT) o el tamaño exacto de las tramas intercambiadas, que son indicadores básicos para detectar cuellos de botella en cualquier sistema distribuido.

Implementar un cliente REST manual, en cambio, da visibilidad completa sobre lo que ocurre en la red. Cada petición HTTP se construye y envía directamente desde el código, lo que permite instrumentarla individualmente: registrar marcas de tiempo, calcular el RTT con precisión de milisegundos y medir el volumen de datos de cada intercambio. En entornos de ingeniería, optimizar el rendimiento sin ese tipo de evidencia empírica es difícil. En este trabajo, la decisión de usar un cliente REST manual sobre las APIs nativas de Firebase, en lugar del SDK oficial para Unity, responde exactamente a esa necesidad: poder medir de forma controlada y reproducible la latencia de red y el tamaño de las tramas generadas por la simulación de combate.

CAPÍTULO 3. DISEÑO Y ARQUITECTURA DEL SISTEMA

Con la base teórica establecida en los capítulos anteriores, este capítulo describe cómo se han trasladado esos conceptos al diseño concreto del sistema. Se explica la arquitectura adoptada para implementar un sistema distribuido de simulación asíncrona de combate por turnos, las decisiones que han guiado su construcción y cómo se organiza la información en la base de datos.

3.1. Arquitectura general de la solución

El sistema se apoya en un servidor autoritativo: toda la lógica de simulación reside en el servidor, que es quien tiene la última palabra sobre el estado del combate. Esto evita que el cliente pueda manipular los resultados, algo fundamental en cualquier aplicación en red donde participen varios usuarios.

El cliente está implementado en Unity con C#. Su papel es deliberadamente limitado: recoge los inputs del usuario, los manda al servidor y representa visualmente lo que este devuelve. Unity no ejecuta ninguna lógica de simulación ni decide nada sobre el estado del combate; se limita a mostrar y permitir la interacción.

El servidor, por su parte, se ha construido sobre Firebase Cloud Functions con Node.js. Cada función es un servicio independiente y *stateless* que se lanza bajo demanda, lo que permite escalar automáticamente según la carga sin tener que gestionar ningún servidor dedicado.

La comunicación entre cliente y servidor es completamente asíncrona, mediante peticiones HTTP POST sobre una API REST. El cliente envía una solicitud, concretamente, la selección del rival contra el que el jugador quiere enfrentarse en la arena, y espera la respuesta sin bloquear la ejecución ni necesitar una conexión persistente. El servidor recibe esa petición, resuelve el combate aplicando la lógica de simulación por turnos y devuelve el resultado, todo sin intervención adicional del usuario. Este modelo encaja bien con la naturaleza discreta y asíncrona del sistema, que no requiere comunicación continua sino respuestas puntuales a eventos concretos. La Figura 1 muestra el diagrama de bloques de esta interacción.

Figura 1. Diagrama de arquitectura del sistema Cliente-Servidor Autoritativo.

3.2. Modelo de datos (Esquema en Firestore)

Para guardar la información se ha usado Cloud Firestore, la base de datos documental de Firebase. A diferencia de las bases de datos relacionales, Firestore no tiene tablas ni esquemas fijos: organiza los datos en colecciones de documentos JSON, lo que hace más fácil modificar la estructura sin migraciones costosas y permite leer solo lo que se necesita en cada momento.

La base de datos se ha dividido en colecciones separadas según el tipo de información, evitando meter en un mismo documento datos que no se consultan juntos. Esto es importante en Firestore porque cada lectura cuenta: leer un documento entero para obtener solo un campo es un gasto innecesario tanto en rendimiento como en coste.

En la Tabla 2 se detalla el esquema adoptado y el nivel de acceso definido para cada colección mediante las reglas de seguridad del servidor.

Tabla 2. Estructura y descripción del modelo de datos en Cloud Firestore.

Colección	Descripción y uso en el sistema	Reglas de Acceso (Seguridad)
users	Datos privados (inventario, misiones, gemas, historial). Agrupado por el UID de Firebase Auth.	Lectura/Escritura restringida única y exclusivamente al propietario del UID.
brutosArena	Datos públicos de combate (estadísticas de los personajes para ser usados como rivales).	Lectura pública (autenticada). Escritura restringida al servidor.
rankings	Clasificación global y mensual. Almacena las victorias y la posición de los jugadores.	Lectura pública. Escritura restringida al servidor mediante *Cloud Functions*.
gamedata	Variables de entorno, multiplicadores de daño y plantillas de los torneos.	Lectura pública. Escritura denegada (Solo administrador).
rateLimits	Control de abusos. Registra el *timestamp* y número de peticiones de cada usuario.	Lectura/Escritura exclusiva del servidor.
rivals / userStates	Caché temporal de rivales y estado de la sesión activa del jugador.	Lectura/Escritura restringida al propietario del UID y al servidor.

3.3. Diseño del protocolo de comunicación (Endpoints y Payloads)

Una vez definida la arquitectura general y el modelo de datos, este apartado describe cómo se materializa la comunicación entre el cliente Unity y los microservicios del servidor. El diseño de este protocolo ha sido una de las decisiones de ingeniería más relevantes del proyecto, ya que de él depende tanto la seguridad de las transacciones como la coherencia del estado compartido entre ambos extremos.

Clase controladora: FirebaseFunctionCaller

Para gestionar la totalidad de las comunicaciones de red en el lado del cliente, se ha implementado una clase estática denominada FirebaseFunctionCaller. Esta clase actúa como la única *pasarela de red* (*Gateway*) del cliente Unity, centralizando y encapsulando todas las peticiones HTTP realizadas mediante la API UnityWebRequest. Adoptar este patrón de punto de acceso único presenta ventajas claras desde el punto de vista del mantenimiento: cualquier cambio en el protocolo de comunicación, ya sea en la serialización, en las cabeceras o en el tratamiento de errores, se aplica de forma homogénea a todas las llamadas del sistema sin necesidad de modificar la lógica de las capas superiores.

La serialización y deserialización de los objetos intercambiados entre cliente y servidor se gestiona mediante la librería Newtonsoft.Json, que permite mapear de forma transparente los objetos C# a su representación JSON y viceversa, garantizando la compatibilidad de tipos entre los dos entornos de ejecución (C# en Unity y Node.js en el servidor).

Control de entornos (Dev/Prod)

FirebaseFunctionCaller implementa un sistema de control de entornos mediante variables estáticas que permiten cambiar de forma sencilla entre las URLs de los microservicios desplegados en entorno de desarrollo y las de producción. Así, por ejemplo, el endpoint de inicio de combate puede apuntar a DEV_START_COMBAT durante la fase de pruebas o a PROD_START_COMBAT en el entorno productivo, sin que sea necesario modificar la lógica de llamada en ningún otro punto del código. Esta separación de entornos es una práctica habitual en ingeniería del software, ya que previene la ejecución accidental de código de prueba sobre datos reales y facilita el proceso de integración continua.

Mecanismo de seguridad: autenticación por token en cabecera

Toda petición HTTP POST emitida por el cliente incorpora en su cabecera (*header*) un Token de Autenticación con el esquema estándar Authorization: Bearer {idToken}. Este token, emitido por Firebase Authentication tras el inicio de sesión del usuario, es un JSON Web Token (JWT) firmado criptográficamente [11]. Antes de procesar el cuerpo (*body*) de cualquier petición, el servidor Node.js verifica la validez e integridad de dicho

token mediante el Firebase Admin SDK. De este modo, se garantiza que únicamente usuarios autenticados pueden invocar las Cloud Functions, y que el servidor puede identificar de forma segura al solicitante sin depender de datos autodeclarados en el cuerpo del mensaje.

Estructura del DTO de respuesta: ServerResponseWrapper

En cuanto al contrato de datos de respuesta (*payload* de retorno), se ha definido un objeto contenedor denominado ServerResponseWrapper. Este *Data Transfer Object* (DTO) encapsula toda la información que el servidor devuelve al cliente tras la simulación de un combate, y su diseño refleja la necesidad de transmitir no solo el resultado final, sino también el detalle del proceso que lo ha generado.

El objeto incluye, entre otros campos, booleanos de estado que indican si la operación se ha completado con éxito, los datos actualizados del jugador (nuevas estadísticas, experiencia o clasificación) y, de forma destacada, el *combatLog*: una lista ordenada de eventos discretos que describe, paso a paso, cada acción ocurrida durante la simulación por turnos en el servidor. Este registro es el elemento fundamental que permite a Unity reproducir la batalla de forma coherente y visualmente detallada, sin haber participado en ningún momento en el cálculo de su resultado. La separación entre la simulación (servidor) y su representación (cliente) queda así materialmente reflejada en la estructura del propio *payload*.

3.3.1. Diagrama de Secuencia de la Comunicación

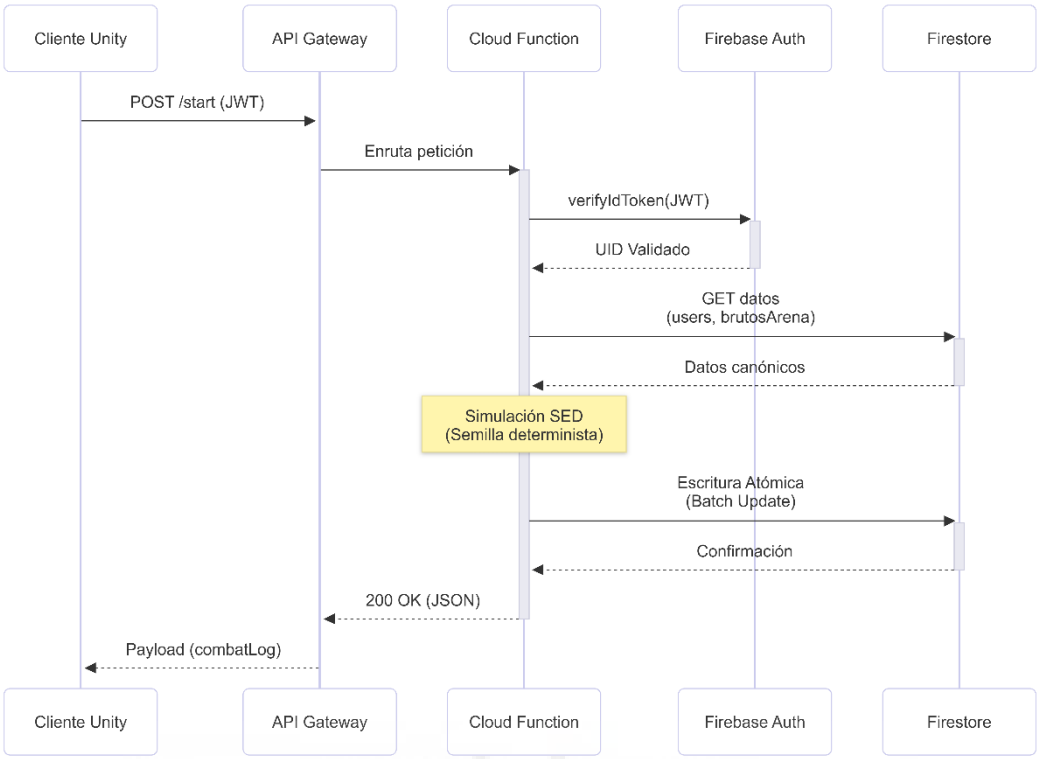

Figura 2. Diagrama de secuencia UML de una transacción completa de simulación.

Como se observa en la Figura 2, el flujo de comunicación comienza cuando el cliente emite una petición POST que incluye un token JWT. El API Gateway enruta esta llamada a la Cloud Function correspondiente, la cual valida primero la identidad del usuario a través de Firebase Auth. Si la verificación es correcta, la función solicita a Firestore los datos necesarios, ejecuta el algoritmo de simulación y registra los resultados de forma atómica en la base de datos. Finalmente, el servidor responde al cliente con un código 200 OK y el JSON que contiene el historial de eventos para su representación visual.

3.4. Modelado del algoritmo de eventos discretos

El núcleo del sistema es el motor de simulación, que corre completamente en el servidor. A diferencia de los motores de juego tradicionales, que evalúan la lógica fotograma a fotograma, este simulador no trabaja en tiempo real: el tiempo avanza a saltos, de un evento al siguiente, siguiendo el modelo de simulación de eventos discretos explicado en el capítulo anterior. Todo el combate se resuelve en el servidor antes de enviar nada al cliente, que simplemente reproduce el resultado. Los subapartados siguientes describen cada componente de este motor en detalle.

3.4.1. Definición de variables de estado

Al inicio de cada simulación, el servidor recupera de Firestore los datos base de las entidades participantes (luchadores y sus mascotas) e instancia sus representaciones en memoria como objetos de estado activos. Estos objetos se componen de un conjunto de atributos primarios: Vida (*HP*), Fuerza, Agilidad y Velocidad, cuyos valores provienen directamente de la base de datos y caracterizan la identidad estadística de cada entidad.

A partir de estos atributos primarios, el motor deriva en tiempo de ejecución un conjunto de atributos secundarios dinámicos, que son los que gobiernan directamente el resultado de cada evento durante el combate. Entre ellos se encuentran la Probabilidad de Crítico, la Esquiva, la Precisión y la Resistencia al daño. El cálculo de estos atributos no es lineal, sino que se apoya en funciones matemáticas diseñadas para modelar rendimientos decrecientes a medida que los valores base crecen.

Un ejemplo representativo es el cálculo de la Resistencia al daño, que escala en función de la vida máxima de la entidad mediante una curva asintótica con una constante de amortiguación. Este tipo de función impide que los valores de resistencia alcancen cotas absolutas que romperían el equilibrio del sistema, por mucha vida que acumule una entidad, su resistencia nunca converge al 100%. De forma complementaria, atributos como la Esquiva están sujetos a funciones de *clamping*, que restringen el valor resultante dentro de un intervalo matemáticamente seguro, en este caso, entre el 5 % y el 80 %, evitando tanto la imposibilidad de impactar como la evasión perfecta, y garantizando que el combate mantenga siempre un componente de incertidumbre controlada. En el Código 1 se muestra la implementación a nivel de código de estas restricciones matemáticas y funciones asintóticas.

```
finalDodgeChance = Math.max(0.05, Math.min(0.80, finalDodgeChance));
```

Código 1. Implementación de funciones de escalado asintótico y clamping para los atributos de combate.

3.4.2. Lógica probabilística y control de flujo

El orden de actuación de las entidades durante el combate no sigue un esquema de turnos alternos estricto del tipo A→B→A→B. En su lugar, se ha implementado un sistema de Medidor de Turno (*Turn Meter*) que introduce un mecanismo de iniciativa dinámica basado en el atributo de Velocidad de cada entidad.

En cada iteración del bucle de simulación, todas las entidades acumulan puntos en su medidor de turno de forma proporcional a su Velocidad. Cuando una entidad supera el umbral de referencia, establecido en 1.000 puntos para operar exclusivamente con números enteros y evitar errores de precisión en coma flotante, se dispara su evento de turno: la entidad ejecuta su acción y su medidor se reinicia. Si en una iteración ninguna entidad alcanza este valor, el sistema incrementa los medidores proporcionalmente hasta que una de ellas lo logre. Este diseño implica que una entidad con el doble de Velocidad que su rival actuará aproximadamente el doble de veces en el mismo intervalo de simulación, replicando con fidelidad el concepto de iniciativa presente en los sistemas de combate de rol por turnos más complejos y añadiendo profundidad estratégica al modelo sin incrementar significativamente el coste computacional.

Toda la lógica de resolución probabilística (incluyendo el cálculo del daño infligido, la determinación de bloqueos y esquivas, y la activación de habilidades especiales) se apoya en un Generador de Números Pseudoaleatorios (PRNG) alimentado por una semilla algorítmica (*seed*) determinada al inicio de la simulación. Esta decisión de diseño tiene una consecuencia técnica importante: dado un mismo estado inicial de las entidades y una misma semilla, el motor de simulación producirá exactamente el mismo resultado en cada ejecución. El sistema es, por tanto, completamente determinista y reproducible, una propiedad esencial en la arquitectura autoritativa, ya que elimina cualquier ambigüedad sobre el resultado oficial del combate y facilita la depuración y verificación del comportamiento del simulador.

3.4.3. Generación del registro de eventos (Log)

La salida del motor algorítmico no se reduce a una variable binaria de victoria o derrota. En su lugar, a medida que el simulador resuelve matemáticamente cada acción, inyecta de forma secuencial un objeto JSON descriptivo en un array denominado *combatLog*. Cada objeto registrado en este array representa un evento discreto ocurrido durante la simulación, e incluye los campos necesarios para que el cliente pueda interpretarlo de forma autónoma. Un ejemplo representativo sería un objeto del tipo {type: 'DamageDealt', damage: 45, isCritical: true}, que codifica no solo la magnitud del daño aplicado sino también su naturaleza, en este caso, un golpe crítico.

Al término de la simulación, este array constituye un registro secuencial de eventos y ordenado de todos los eventos que han conformado el combate, y es el que se incluye en el ServerResponseWrapper transmitido al cliente. Unity, al recibirlo, recorre el registro de forma iterativa e instancia las animaciones, efectos visuales y transiciones de interfaz que corresponden a cada entrada, sin conocer ni haber participado en ningún momento en el cálculo matemático que las originó.

Este mecanismo aplica el principio de desacoplamiento entre lógica y presentación que sustenta toda la arquitectura del sistema: el servidor ejecuta los cálculos subyacentes de la simulación, y el cliente se limita a ser un intérprete fiel de su registro.

CAPÍTULO 4. DESARROLLO E IMPLEMENTACIÓN DEL PROTOTIPO

Este capítulo describe cómo se ha implementado el sistema diseñado en el capítulo anterior. Se recogen las decisiones de codificación tomadas en ambos extremos: el cliente visual en C# sobre Unity y los microservicios en JavaScript sobre Firebase Cloud Functions. El objetivo es explicar no solo qué se ha construido, sino cómo y por qué se han tomado ciertas decisiones técnicas, desde la gestión de la asincronía en el cliente hasta los mecanismos de seguridad y escritura en base de datos en el servidor.

4.1. Implementación del Cliente Visual (Unity / C#)

4.1.1. Gestión de peticiones asíncronas

La comunicación de red en el cliente se ha implementado sin usar el sistema de Corrutinas (*Coroutines*) que tradicionalmente se ha usado en Unity para manejar operaciones asíncronas. En su lugar se ha optado por async/await con Task<T>, el modelo asíncrono estándar de C# actual, que tiene una sintaxis más clara, maneja errores de forma más limpia mediante bloques try/catch y se integra mejor en flujos de código complejos.

Las peticiones al servidor se construyen mediante UnityWebRequest, configurado con dos manejadores especializados. El UploadHandlerRaw se encarga de serializar el objeto de petición como una cadena JSON codificada en UTF-8 e inyectarla en el cuerpo (*body*) del mensaje HTTP POST, estableciendo la cabecera Content-Type:

application/json. El DownloadHandlerBuffer gestiona la recepción de la respuesta del servidor, almacenando el contenido binario para su posterior conversión a cadena de texto y deserialización. Adicionalmente, antes de enviar la petición, se adjunta en la cabecera el token de autenticación bajo el esquema Authorization: Bearer {idToken}, cerrando así el ciclo de seguridad descrito en el apartado anterior. La construcción de esta petición asíncrona mediante la clase controladora se detalla en el Código 2.

```
private static UnityWebRequest CreatePostRequest(string url, string
jsonBody, string idToken) {

        var request = new UnityWebRequest(url, "POST");

        byte[] bodyRaw = Encoding.UTF8.GetBytes(jsonBody);

        request.uploadHandler = new UploadHandlerRaw(bodyRaw);

        request.downloadHandler = new DownloadHandlerBuffer();

        request.SetRequestHeader("Content-Type", "application/json");

        if (!string.IsNullOrEmpty(idToken))
request.SetRequestHeader("Authorization", $"Bearer {idToken}");

        return request; }
```

Código 2. Construcción y envío de la petición HTTP asíncrona mediante UnityWebRequest.

4.1.2. Deserialización e interpretación de eventos

Una vez recibida la respuesta del servidor, el string JSON es deserializado a su correspondiente objeto C# mediante Newtonsoft.Json, instanciando el ServerResponseWrapper descrito en el apartado 3.3. Este DTO contiene, entre otros campos, el *combatLog*: la lista ordenada de eventos discretos que componen la simulación.

A partir de este momento, el control pasa a un controlador visual dedicado en Unity, cuya responsabilidad es iterar secuencialmente sobre los elementos del *combatLog* e instanciar, para cada entrada, la respuesta visual correspondiente. Dado que cada evento debe reproducirse de forma ordenada y con la cadencia temporal adecuada para que el resultado sea legible para el jugador, el controlador emplea un sistema de colas o temporizadores que regula el ritmo de reproducción. Así, eventos como un golpe crítico, una esquiva o la derrota de una entidad desencadenan sus animaciones y actualizaciones

de interfaz de usuario (UI) en el orden exacto en que fueron calculados por el servidor, replicando fielmente la simulación sin que el cliente haya intervenido en su cómputo.

4.1.3. Representación Visual e Interfaz de Usuario (UI)

Aunque el núcleo del sistema está en el servidor autoritativo, el cliente también tiene su papel: traducir las estructuras de datos JSON en algo que el jugador pueda ver e interactuar. Las siguientes capturas muestran cómo Unity representa las distintas fases de la interacción, desde la consulta del estado hasta la reproducción del combate.

1. Menú Principal y Estado General

Al iniciar sesión, el cliente hace una primera sincronización para obtener el estado del jugador. Como se ve en la Figura 3, el Menú Principal actúa como pantalla central del usuario, donde se muestra la información de la colección users de Firestore: nombre del jugador, nivel actual, recursos (gemas) y el combatiente seleccionado con su nivel. Todos estos valores son de solo lectura; el cliente no tiene autoridad para modificar gemas ni nivel.

Figura 3. Interfaz del Menú Principal con la información general de la cuenta y el combatiente activo.

2. Menú de Estadísticas y Equipamiento

Para consultar las capacidades del personaje en detalle, la aplicación tiene un menú dedicado (Figura 4) donde se muestran las estadísticas primarias y secundarias del combatiente: Vida, Fuerza, Agilidad, armas, etc. Esta pantalla es en realidad la

representación visual del modelo de datos que el motor de simulación en Node.js usará después en memoria para resolver el combate. Mostrar estas estadísticas de forma transparente permite al jugador saber con qué variables entra al combate, aunque el cálculo siempre lo haga el servidor.

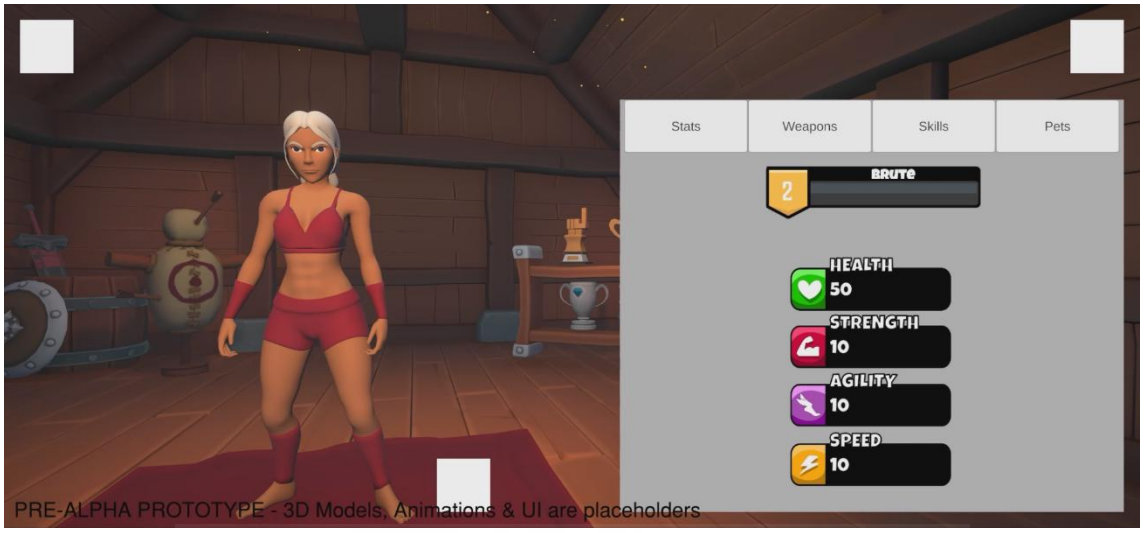

Figura 4. Menú de estadísticas detalladas con los atributos que el servidor usará durante la simulación.

3. Selección de Rival y Disparo de la Petición Asíncrona

El flujo de simulación hacia el combate se inicia en la interfaz de la Arena o selección de rival. En esta pantalla (Figura 5), el cliente lista posibles oponentes basándose en los datos públicos consultados en Firestore.

El momento crítico de la comunicación cliente-servidor ocurre en esta vista: cuando el usuario selecciona a un oponente y pulsa el botón de luchar, el cliente Unity bloquea temporalmente la interfaz y utiliza la pasarela de red (FirebaseFunctionCaller) para emitir la petición HTTP POST asíncrona hacia la Cloud Function.

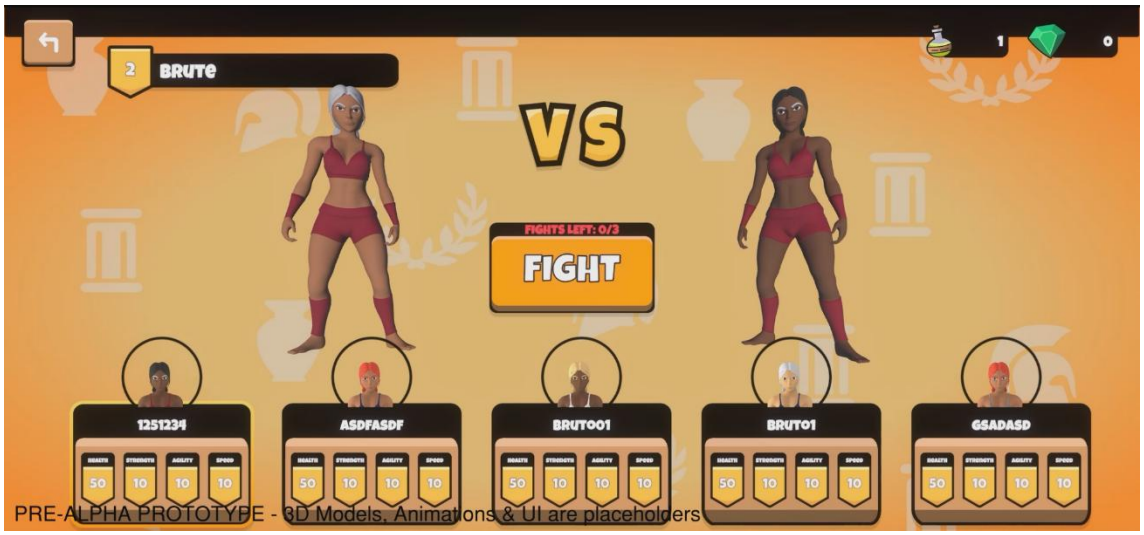

Figura 5. Pantalla de selección de rival, desde donde se envía la petición al servidor autoritativo.

4. Representación del Registro de Eventos (Combate)

Cuando el servidor termina la simulación y devuelve el DTO con código HTTP 200 OK, el cliente procesa la respuesta y pasa a la escena de combate (Figura 6).

Esta escena no ejecuta ninguna lógica probabilística ni de colisiones. Un controlador visual lee secuencialmente el array combatLog del JSON recibido y, según el tipo de evento (golpe crítico, esquiva, etc.), lanza la animación correspondiente en los modelos y actualiza las barras de vida. El cliente es un reproductor: el resultado del combate ya estaba calculado y validado en el servidor antes de que aparezca nada en pantalla.

Figura 6. Representación visual del combate en Unity. El cliente se limita a animar en orden los eventos calculados por el servidor.

4.2. Implementación de la Lógica de Servidor (Node.js / JavaScript)

4.2.1. Configuración de Cloud Functions y política CORS

Los microservicios del servidor se han implementado como funciones HTTP de Firebase mediante el método onRequest, que expone cada función como un endpoint REST independiente accesible a través de su URL pública. Con el objetivo de optimizar los costes operativos, cada función se ha declarado explícitamente con recursos controlados: se ha especificado la generación de entorno de ejecución gcf_gen1 y un límite de memoria RAM de 256 MiB, valores suficientes para la carga computacional de la simulación y que minimizan el coste por invocación frente a configuraciones más permisivas.

Por defecto, los navegadores web bloquean por seguridad las peticiones hechas hacia un dominio distinto al de origen. Para solucionarlo, se ha implementado el estándar CORS. Cuando el cliente intenta comunicarse, el navegador envía primero una petición de consulta (OPTIONS). El servidor la intercepta y responde con cabeceras de autorización (Access-Control-Allow-Origin), dando permiso explícito al navegador para enviar la petición real.

4.2.2. Capas de Seguridad Implementadas

A diferencia de los modelos distribuidos genéricos, en esta arquitectura no se confía en ningún dato enviado por el cliente. La verificación de identidad constituye el primer bloque de ejecución en todo *endpoint* del sistema. Para proteger la integridad de la base de datos, se han diseñado e implementado las siguientes capas de seguridad consecutivas en el servidor:

- 1. Filtrado de Protocolo y CORS: Se comprueba que el método de la petición entrante sea POST, rechazando con un error 405 cualquier otro verbo HTTP. Además, se emplea el *middleware cors* configurado explícitamente para limitar y validar los orígenes de las peticiones (*preflight* OPTIONS).

- 2. Autenticación Criptográfica (JWT): El servidor extrae el token de la cabecera Authorization parseando el esquema Bearer {idToken}. Utilizando las librerías *jsonwebtoken* y *jwks-rsa* (o *firebase-admin*), se descarga la clave pública oficial y se valida la firma digital, la fecha de expiración y el

emisor [14]. Este enfoque dinámico previene eficazmente la fuga de permisos y la suplantación de identidad en arquitecturas *serverless* [12].

- 3. Aislamiento por UID: Si la verificación criptográfica es exitosa, se extrae el campo uid del token decodificado, obteniendo el identificador inequívoco del usuario solicitante. Este identificador se utiliza obligatoriamente en todas las operaciones posteriores, garantizando que ningún usuario pueda operar sobre los datos de otro, ni siquiera conociendo su ID.

- 4. Transacciones Atómicas Anti-Colisión: En operaciones críticas (como la inscripción a torneos o la subida de nivel), se utiliza el método db.runTransaction(). Esto implementa bloqueos optimistas que previenen condiciones de carrera (*race conditions*), evitando que un atacante envíe solicitudes simultáneas para duplicar recompensas o corromper el estado [15].

Si cualquiera de estas validaciones falla en cualquier punto del flujo (token expirado, manipulado o ausente), la ejecución se interrumpe de inmediato devolviendo un error *401 Unauthorized*, sin llegar a procesar el cuerpo de la petición. El núcleo de esta validación se expone en el Código 3.

```
const decodedToken = await admin.auth().verifyIdToken(token);
```

Código 3. Interceptación y validación criptográfica del token JWT mediante Firebase Admin SDK.

4.2.3. Ejecución de la simulación y escritura en base de datos

Superada la capa de seguridad, el servidor procede a validar y limpiar los datos de entrada recibidos en el cuerpo de la petición (*data validation*), descartando campos inesperados o malformados antes de operar con ellos. A continuación, se importan los modelos de entidad del dominio (*Fighter* y *AttackData*) que encapsulan la lógica de construcción de los combatientes a partir de los datos recuperados de Firestore.

El motor de simulación se inicializa inyectando una semilla algorítmica mediante la librería seedrandom, que sustituye el generador de números pseudoaleatorios nativo de JavaScript por uno determinista y reproducible, tal como se describió en el apartado 3.4.2. Con el estado inicial completamente definido, la simulación se ejecuta de forma

síncrona en el hilo de la función, completándose en un tiempo de cómputo del orden de milisegundos dado el carácter discreto y acotado del algoritmo.

Una vez finalizada la ejecución, el servidor dispone del resultado completo: el *combatLog*, las estadísticas actualizadas del jugador y los cambios en la clasificación. Estos datos se vuelcan de forma estructurada en las colecciones correspondientes de Firestore (*brutosArena, users, rankings*) mediante operaciones de escritura atómica, antes de construir y devolver el ServerResponseWrapper al cliente.

4.3. Vulnerabilidades, problemas técnicos y soluciones adoptadas

Todo sistema en red expuesto a usuarios finales debe contemplar, desde la fase de diseño, los posibles fallos y puntos de ataque que pueden comprometer la integridad de los datos o la disponibilidad del servicio. Este apartado describe las tres amenazas principales identificadas durante el desarrollo del prototipo y las soluciones técnicas aplicadas para mitigarlas.

Prevención de trampas y manipulación de memoria (Anti-Cheat)

En el ecosistema de los juegos móviles es habitual el uso de herramientas de edición de memoria, como Cheat Engine, o de versiones modificadas de la aplicación (*APKs* manipuladas) con las que un atacante puede alterar en tiempo de ejecución los valores almacenados localmente en el cliente: la vida de su personaje, el daño de su arma o cualquier otro atributo relevante para el combate.

La arquitectura autoritativa implementada neutraliza este problema de raíz, sin necesidad de ninguna medida adicional en el cliente. El servidor nunca procesa ni confía en los parámetros de juego enviados en el cuerpo de la petición. En su lugar, una vez verificado el token de autenticación y extraído el uid del usuario, el servidor recupera directamente de Firestore los valores oficiales de las entidades asociadas a ese identificador. Cualquier manipulación realizada en el cliente es, por tanto, completamente irrelevante: los datos alterados no alcanzan nunca el motor de simulación.

Resiliencia ante desconexiones (Rage Quit)

Una de las problemáticas clásicas en juegos móviles con componentes competitivos es la pérdida de conectividad durante una partida, ya sea involuntaria, por pérdida de cobertura Wi-Fi o 4G, o deliberada, en la práctica conocida como *rage quit*: el jugador interrumpe la conexión de forma intencionada para evitar que una derrota inminente quede registrada en el sistema.

Este problema se mitiga mediante el diseño asíncrono y sin estado (*stateless*) del sistema. Por un lado, si la pérdida de conexión se produce antes de que el cliente logre emitir la petición POST, el servidor nunca recibe el evento y el estado de la base de datos permanece inalterado. Por otro lado, en el instante en que el servidor recibe y valida una petición, completa la simulación íntegra de forma independiente a la disponibilidad del cliente. El resultado se escribe de forma atómica en Firestore (es decir, garantizando que la operación de guardado se ejecuta en su totalidad o se cancela por completo, evitando datos corruptos) antes de intentar devolver respuesta alguna. Si el jugador pierde la conexión tras enviar la petición, el combate ya ha sido calculado y registrado, por lo que la desconexión no tiene ningún efecto sobre el resultado.

Control de abuso de la API (Rate Limiting)

La exposición de endpoints públicos mediante onRequest introduce un riesgo operativo relevante en entornos de nube de pago por uso: un atacante podría lanzar bucles automatizados de peticiones, ya sea con fines de denegación de servicio (DDoS) o de abuso sistemático de la lógica de simulación, provocando un número de invocaciones de Cloud Functions muy superior al esperado y disparando los costes de facturación en Google Cloud de forma descontrolada.

Para mitigar este riesgo se ha implementado un mecanismo de limitación de frecuencia de peticiones (*Rate Limiting*) apoyado en la colección auxiliar *rateLimits* de Firestore, introducida en el apartado 3.2. Cada vez que un usuario invoca un endpoint sujeto a esta política, el servidor consulta dicha colección para verificar si ha transcurrido el tiempo de espera mínimo (*cooldown*) desde la última petición registrada para ese uid. Si el intervalo no se ha cumplido, la función rechaza la petición antes de ejecutar ninguna lógica de simulación ni acceso adicional a la base de datos. En caso contrario, actualiza el *timestamp* correspondiente y continúa con el flujo normal. Este mecanismo favorece

que, incluso en presencia de clientes maliciosos o disfuncionales, el número de simulaciones ejecutadas por usuario queda acotado, protegiendo tanto la disponibilidad del servicio como la previsibilidad de los costes operativos.

CAPÍTULO 5. EVALUACIÓN EMPÍRICA Y ANÁLISIS DE RENDIMIENTO

Que el sistema funcione no es suficiente: hay que medir cómo se comporta en condiciones reales. El presente capítulo analiza el rendimiento del sistema a partir de datos empíricos obtenidos directamente de los registros (*logs*) del servidor y del cliente Unity, evaluando tres dimensiones clave: la escalabilidad económica del despliegue en la nube, la eficiencia computacional del motor de simulación y las características de la comunicación de red entre cliente y servidor.

5.1. Diseño del entorno de pruebas y escalabilidad de costes

El backend del sistema se ha desplegado en producción sobre Google Cloud Platform bajo el plan de facturación Blaze (*Pay-as-you-go*), que es el requerido por Firebase para habilitar las Cloud Functions con conectividad externa. No obstante, la naturaleza sin estado y de ejecución bajo demanda de la arquitectura serverless adoptada permite que el sistema opere con holgura dentro de los límites de la capa gratuita (*Free Tier*) que Google ofrece incluso dentro de dicho plan.

Concretamente, el Free Tier cubre 2.000.000 de invocaciones mensuales de Cloud Functions sin coste, así como 50.000 lecturas y 20.000 escrituras diarias en Firestore. Para un prototipo con una base de usuarios en fase de validación, estas cuotas resultan ampliamente suficientes, lo que implica que el coste operativo real del sistema durante las pruebas ha sido de cero euros. Esto tiene relevancia práctica: el sistema es viable desde el primer día sin inversión en infraestructura.

5.2. Análisis de rendimiento computacional en servidor

Las mediciones de rendimiento se han obtenido instrumentando directamente los registros de ejecución de las *Cloud Functions*. Un combate promedio abarca una duración algorítmica aproximada de 8 turnos, lo que constituye una carga de simulación altamente compacta. Para garantizar la robustez y validez de los resultados, se ejecutó una batería automatizada de N=30 combates completos consecutivos (en adelante referidos como simulaciones). Se eligió este tamaño muestral para satisfacer el criterio del Teorema del Límite Central, asegurando que la distribución de las medias muestrales

de latencia y tiempo de ejecución se aproxime a una distribución normal, lo que permite extraer conclusiones estadísticamente significativas del rendimiento del sistema.

El análisis de los *logs* revela la presencia del fenómeno de *Cold Start*, inherente a las arquitecturas *serverless*. En la invocación inicial tras un periodo de inactividad, la plataforma debe aprovisionar y levantar un nuevo contenedor (evento *AUTOSCALING*). En estas condiciones, el tiempo de ejecución total medido ascendió a 2,23 segundos.

En contraste, durante las ejecuciones consecutivas (en lo que se denomina *Warm Start*, con el contenedor ya instanciado en memoria), el tiempo medio de ejecución síncrona y escritura atómica en Firestore se redujo drásticamente, estabilizándose en una media de 235 milisegundos (0,23 s). Este tiempo de procesamiento para un algoritmo que incluye lectura del perfil, resolución pseudoaleatoria del combate y múltiples operaciones de escritura distribuida, representa un tiempo de procesamiento eficiente. La evidencia empírica de estos tiempos de respuesta y eventos de auto-escalado extraída de la consola de monitorización de Google Cloud se detalla visualmente en la Figura 7.

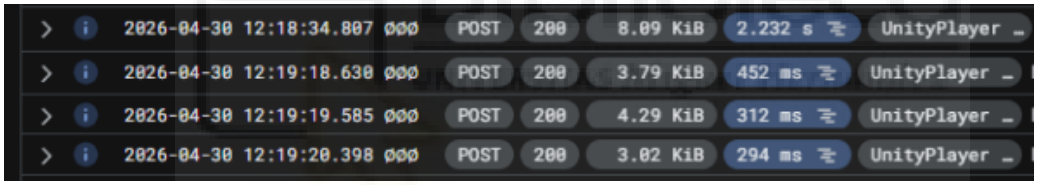

Figura 7. Registros empíricos de telemetría en Google Cloud mostrando el evento de Cold Start, tiempos de ejecución y tamaño de payloads.

5.3. Análisis de comunicaciones y rendimiento secuencial

Para evaluar cómo responde la arquitectura *serverless* frente a un uso continuado y validar el impacto del *Cold Start*, se ha ejecutado una batería de pruebas de carga automatizadas desde el cliente Unity.

Eficiencia del Payload y Tiempos de Servidor

La telemetría interna de Google Cloud muestra que el objeto JSON devuelto por el servidor tiene un tamaño variable de entre 2,26 KiB y 6,03 KiB, con una media aproximada de 3,8 KiB por combate. El tiempo de procesamiento de la Cloud Function se mantiene eficiente, resolviendo la lógica y la escritura atómica en Firestore en torno a los 150-190 milisegundos.

Evaluación del fenómeno Cold Start

La primera prueba consistió en medir la latencia total (RTT) de la primera petición tras un periodo de inactividad. El resultado fue de 933,90 ms. Este retardo inicial, cercano a un segundo, se debe al aprovisionamiento de un nuevo contenedor en la infraestructura de Google, lo que se conoce como *Cold Start* y es una característica inherente del modelo FaaS.

Pruebas de Escalabilidad Horizontal (Warm Start)

Para aislar el rendimiento real y simular un entorno en producción, se ejecutó una fase de calentamiento previa a cuatro baterías consecutivas de distinto tamaño muestral (N=10, N=30, N=50 y N=100), manteniendo el contenedor en memoria durante todas ellas.

Tabla 3. Resumen estadístico de las pruebas de carga (Latencia RTT en el cliente).

Muestra Evaluada	RTT Medio (ms)	RTT Mínimo (ms)	RTT Máximo (ms)
N = 10 combates	584,50	484,17	690,59
N = 30 combates	475,04	329,01	741,74
N = 50 combates	493,40	412,91	1045,24
N = 100 combates	474,08	315,70	1037,76

Análisis de los resultados

Como muestra la Tabla 3, el RTT medio presenta una clara tendencia a estabilizarse. A mayor volumen de peticiones, el rendimiento mejora y se asienta en torno a los 474 milisegundos. Esto confirma que, una vez superado el arranque inicial, el sistema absorbe la carga sin mostrar síntomas de saturación.

Los picos máximos registrados, en torno a los 1045 ms en casos puntuales, corresponden a fluctuaciones transitorias de red (*jitter*) y no a problemas de procesamiento del backend. La Figura 8 muestra esta estabilidad a lo largo de 100 transacciones consecutivas.

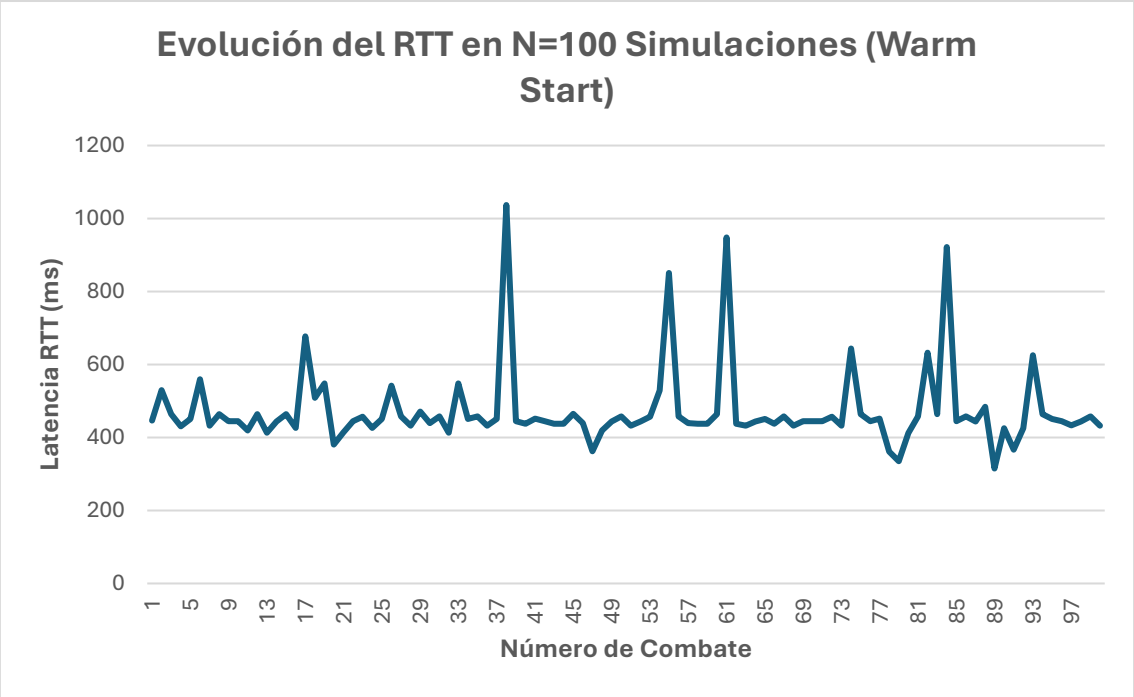

Figura 8. Dispersión del RTT a lo largo de 100 simulaciones de combate consecutivas (Warm Start).

Una latencia media por debajo de medio segundo es un resultado bueno para un juego asíncrono. Al estar cubierta por la transición visual y el bloqueo momentáneo de la interfaz tras seleccionar rival, el usuario percibe la respuesta como inmediata. En definitiva, las métricas confirman que el modelo *serverless* mantiene la latencia dentro de los márgenes aceptables incluso bajo carga continuada.

5.4. Análisis de concurrencia y cuellos de botella (Load Testing)

Tras validar el rendimiento secuencial del sistema, se planteó un escenario de pruebas de estrés (*Load Testing*) para ver cómo se comporta la arquitectura bajo carga concurrente masiva, simulando decenas de jugadores interactuando en el mismo instante.

Al diseñar esta prueba aparecieron dos limitaciones propias de los sistemas *Cloud* y las redes locales que condicionan el escalado masivo:

1. Saturación del cliente vs. servidor (El problema de la telemetría local)
Al intentar emitir ráfagas de 50 o 100 peticiones HTTP POST simultáneas desde una única instancia del cliente Unity, los tiempos de respuesta (RTT) subieron a más de 3.000 milisegundos. Sin embargo, al contrastar esta latencia con la telemetría interna de Google Cloud, se vio que las Cloud Functions resolvían la simulación y la escritura en

apenas ~200 milisegundos. El retardo no venía del servidor, sino del encolamiento de conexiones TCP en la tarjeta de red del propio ordenador de pruebas y en el ancho de banda del router local. Esto deja claro que para medir concurrencia real hace falta usar herramientas distribuidas especializadas (como *Apache JMeter* o *Gatling*) desde múltiples nodos externos, en lugar de un único cliente.

2. Limitaciones de cuota y 'Rate Limiting'

El segundo hallazgo fue el mecanismo de autodefensa de Firebase. Al inyectar tráfico masivo desde una única dirección IP, el proveedor bloqueó las peticiones devolviendo el error *Quota Exceeded*. Esto confirma que las políticas de protección frente a ataques de denegación de servicio (DDoS), implícitas en las capas gratuitas de Google, funcionan: un uso malicioso o un error de programación en el cliente no debería generar facturas descontroladas por invocaciones infinitas.

Análisis teórico del cuello de botella: Contención en Firestore

A partir de la arquitectura diseñada, se puede anticipar dónde estaría el principal cuello de botella si 1.000 jugadores reales jugaran un combate al mismo tiempo. Como la simulación es un proceso *stateless* aislado, Node.js escalaría levantando cientos de contenedores sin problema. El fallo ocurriría al guardar los rankings. Al terminar el combate, todas las Cloud Functions concurrentes intentarían ejecutar la rutina encargada de actualizar la tabla de clasificación (incrementRankingStats()), apuntando al mismo documento global de Firestore. Este motor NoSQL escala bien en lecturas, pero tiene un límite documentado de aproximadamente una escritura por segundo sobre un mismo documento. Las 1.000 peticiones colisionarían entre sí, y la mayoría se rechazarían o quedarían atrapadas en reintentos, con un impacto serio en la latencia.

Solución arquitectónica propuesta

Para que este prototipo pudiera soportar miles de combates por segundo sin que la base de datos se colapse, la estrategia habitual es desacoplar mediante colas de mensajes y agrupamiento (*batching*).

La idea sería que la Cloud Function de combate no actualizara el ranking directamente, sino que publicara un evento ligero de "victoria" en un bus en memoria, como *Google* Cloud Pub/Sub o Redis. Un único microservicio trabajador (*worker*) leería esa cola de forma asíncrona, agruparía las victorias de los últimos cinco segundos en un solo paquete y actualizaría Firestore con una única escritura. Esto eliminaría la contención y

permitiría que la simulación de combate siga siendo determinista, inmediata y escalable, sin depender de la carga sobre el sistema de clasificaciones.

5.5. Análisis comparativo de alternativas algorítmicas

Atendiendo a la necesidad de justificar las decisiones de diseño arquitectónico y evidenciar la optimización aportada sobre las funciones estándar de la plataforma, se ha realizado una comparativa empírica evaluando dos enfoques distintos para la persistencia del registro de combate (*combatLog*).

Dado que Firebase expone múltiples patrones para interactuar con su base de datos documental (Firestore), se ha implementado de forma temporal una algoritmia iterativa genérica (Alternativa A: ejecutar una petición asíncrona independiente de escritura por cada evento discreto generado en el turno, simulando un guardado continuo). Esta alternativa se ha comparado frente al enfoque optimizado desarrollado en este proyecto (Alternativa B: procesar la totalidad de la simulación en memoria a nivel de servidor O(1) en operaciones de red, y consolidar los resultados mediante una única escritura atómica o *Batch Update* final).

Para evaluar el impacto real, se inyectó la Alternativa A en el entorno de producción y se lanzó una batería automatizada de N=30 combates desde el cliente Unity, capturando las métricas de red y los registros internos del contenedor *serverless*. Los resultados comparativos se detallan en la Tabla 4.

Tabla 4. Comparativa de rendimiento de persistencia (Media de N=30 combates completos).

Métrica Evaluada	**Alternativa A (Escritura Secuencial de Eventos)**	**Alternativa B (Simulación en Memoria + Batch Atómico)**	**Mejora Obtenida**
Tiempo int. de escritura (Server)	~ 1.845 ms	~ 235 ms	- 87,2 %
Latencia Total de Red (RTT)	2.423 ms	475 ms	- 80,4 %
Volumen de transacciones a DB	42 a 85 por combate	1 operación atómica	O(1) vs O(N)

Valoración de la alternativa escogida:

Los datos de telemetría muestran que usar un patrón de guardado iterativo estándar (Alternativa A) penaliza seriamente el rendimiento. Al bloquear el hilo de ejecución esperando la confirmación de Firestore en cada iteración, el tiempo interno de procesamiento llegó a superar los 4,1 segundos en combates densos (85 eventos discretos). Como consecuencia, la latencia media percibida por el usuario (RTT) fue de 2,42 segundos, con picos de 8,8 segundos. A esto se suma que este enfoque multiplicaría el coste de facturación de forma proporcional al número de lecturas y escrituras en Cloud.

La Alternativa B justifica tanto técnica como económicamente el modelo desarrollado. Resolver toda la lógica en memoria y limitar la comunicación con la base de datos a un único paquete de datos ha permitido reducir la latencia cerca de un 80%. Esto confirma que uno de los factores más importantes del modelo es el diseño de la estructura de datos y el control del flujo de comunicación, especialmente cuando se trabaja con servicios gestionados que tienen limitaciones estructurales propias.

5.6. Evaluación práctica de las medidas de seguridad (Prueba de abuso de API)

Para evaluar prácticamente la robustez de las mitigaciones implementadas frente a ataques maliciosos o fallos del cliente, se diseñó un escenario de prueba de estrés. En los sistemas interactivos, una vulnerabilidad común ocurre cuando un cliente modificado envía peticiones masivas al servidor para saturar la base de datos o agotar la cuota de facturación (ataque de denegación de servicio a nivel de aplicación).

Para medir la eficacia de la "Capa de Mitigación de Abuso" aportada al proyecto, se instrumentó el cliente Unity para emitir 50 peticiones simultáneas de actualización y búsqueda de rivales al endpoint correspondiente, simulando un comportamiento anómalo.

Resultados obtenidos:

- Sin Rate Limiting: El servidor intentaba procesar las 50 peticiones, ejecutando 50 lecturas a la base de datos y multiplicando el tiempo de procesamiento concurrente y el gasto de cuota.

- Con la solución implementada: El *middleware* de seguridad validó el *timestamp* del UID en milisegundos. Solo la primera petición fue procesada hasta la base de datos. Las 49 peticiones restantes fueron interceptadas y abortadas en la capa de red con un código HTTP 429 Too Many Requests.

Esta comparativa práctica demuestra que la lógica de validación añadida no solo protege la coherencia de los datos del juego, sino que es un mecanismo indispensable para blindar la viabilidad económica del sistema en una arquitectura *Cloud* de pago por uso.

5.7. Limitaciones del sistema

El prototipo funciona bien en las condiciones probadas, pero hay limitaciones estructurales que conviene señalar pensando en un escalado mayor.

La primera es el riesgo de *Vendor Lock-in*, es decir, la dependencia del proveedor. Al acoplar la lógica de base de datos a las particularidades de Cloud Firestore y Firebase Cloud Functions, una futura migración hacia otra infraestructura (como AWS Lambda o Microsoft Azure) requeriría reescribir una parte considerable del código de acceso a datos y seguridad.

La segunda tiene que ver con los costes. El plan gratuito de Firebase cubre sin problemas la carga en fase de validación, pero el modelo de precios de Firestore, basado en el número de lecturas y escrituras, puede volverse caro si el sistema escala a cientos de miles de usuarios concurrentes. En ese escenario, habría que replantear el diseño: usar bases de datos en memoria como Redis o agrupar operaciones en cola (*batching*) para reducir el número de escrituras individuales.

Por último, las métricas obtenidas se han medido en un entorno controlado con un número limitado de combates (N=30), por lo que los resultados son indicativos. Variables como la calidad de la red móvil del usuario, la carga global de Google Cloud o la distancia geográfica al servidor no se han podido aislar en este análisis.

CAPÍTULO 6. CONCLUSIONES Y TRABAJO FUTURO

6.1. Conclusiones generales

Los resultados obtenidos confirman que la arquitectura autoritativa es una opción adecuada para este tipo de sistemas. Las decisiones de diseño tomadas en cada capa, desde el modelo de datos hasta el protocolo de comunicación, han respondido bien a los requisitos planteados.

En cuanto a la seguridad, centralizar toda la lógica de simulación en el servidor ha eliminado directamente la posibilidad de que el cliente manipule el resultado. Al no confiar en ningún parámetro enviado por Unity y trabajar solo con los datos almacenados en Firestore, el sistema no es vulnerable a las técnicas de edición de memoria o modificación de cliente habituales en juegos móviles. Esta protección no es una medida añadida encima del sistema: es una consecuencia directa de cómo está diseñado.

En cuanto a la eficiencia de red, usar un único intercambio asíncrono HTTP POST ha demostrado ser suficiente y eficiente. El *payload* completo del *combatLog*, con todo el historial de la simulación, pesa de media 3,8 KiB por combate (entre 2,26 y 6,03 KiB), sin necesidad de conexión persistente ni flujo continuo de datos como ocurre en arquitecturas basadas en sockets.

En cuanto a costes, la arquitectura *serverless* sobre Firebase ha mantenido el coste de infraestructura en cero durante toda la fase de desarrollo y validación, dentro de los límites del plan gratuito de Google Cloud. Combinado con el escalado automático propio de las funciones bajo demanda, esto hace que la solución pueda crecer sin tener que rediseñar la infraestructura.

6.2. Cumplimiento de objetivos

Los resultados obtenidos confirman que se han cumplido todos los objetivos de ingeniería planteados al inicio del proyecto.

El objetivo principal, diseñar, modelar y evaluar empíricamente una arquitectura cliente-servidor distribuida para sistemas de simulación basados en eventos discretos, se ha completado. Se ha diseñado una arquitectura funcional, implementado un motor de

simulación determinista basado en eventos discretos y medido el comportamiento real del sistema en términos de rendimiento, latencia y uso de red.

En cuanto a los objetivos específicos, el cliente visual en Unity ha sido desarrollado e integrado correctamente, con async/await y un controlador de reproducción secuencial. El servidor está desplegado en Firebase Cloud Functions con mecanismos de seguridad (JWT) y control de *rate limiting*. Por último, la instrumentación de red ha permitido obtener métricas concretas: 235 ms de tiempo de ejecución en servidor (*Warm Start*), 475 ms de latencia total (RTT) y cargas medias de 3,8 KiB, lo que valida cuantitativamente que el sistema funciona bien en condiciones reales de producción.

6.3. Limitaciones de la arquitectura y alternativas en tiempo real

Como línea de evaluación adicional, se ha planteado una comparativa teórica entre el modelo *serverless* asíncrono implementado y una topología de servidor dedicado en tiempo real (*stateful* mediante WebSockets o UDP).

Todo diseño de sistema interactivo implica *trade-offs*: priorizar unas métricas obliga a sacrificar otras. Si el caso de estudio requiriera combate en tiempo real, resolución de impactos en milisegundos, la arquitectura actual basada en Firebase Cloud Functions no sería viable, por las limitaciones de latencia del protocolo HTTP y el fenómeno de *Cold Start*. En ese escenario, habría que migrar a un servidor dedicado persistente. La Tabla 5 resume las implicaciones de ese cambio.

Tabla 5. Comparativa arquitectónica teórica: Serverless vs Servidor Dedicado.

Parámetro Evaluado	**Arquitectura Actual (Serverless / REST)**	**Alternativa Tiempo Real (Dedicado / WebSockets)**
Conexión de red	Peticiones HTTP aisladas (Sin estado).	Conexión TCP/UDP persistente.
Latencia esperada	Alta (~500 ms). Apta para turnos.	Muy baja (<50 ms). Apta para tiempo real.
Escalabilidad	Automática y elástica (Escala a cero).	Manual. Requiere sobre-aprovisionamiento.
Modelo de costes	Pago por uso (Por invocación/lectura).	Coste fijo por tiempo de servidor encendido (Uptime).
Complejidad de código	Baja. La plataforma gestiona la concurrencia.	Alta. Requiere predicción del cliente y *rollback*.

Valoración del análisis:
La principal desventaja de una arquitectura en tiempo real es la ineficiencia económica en periodos de baja concurrencia. Un servidor dedicado consume recursos (CPU/RAM) y genera costes fijos, aunque no haya jugadores conectados. Además, mantener el estado sincrónico de la simulación exige técnicas complejas de predicción en el cliente (*Client-Side Prediction*) para enmascarar el lag.

Dado que el motor de combate funciona por turnos, la elección de una arquitectura *serverless* tiene sentido. Se sacrifica la inmediatez del tiempo real a cambio de una escalabilidad prácticamente ilimitada, un coste proporcional al uso real y una reducción importante en la complejidad de mantenimiento de la infraestructura.

6.4. Líneas de trabajo futuro

El prototipo desarrollado es una base sólida sobre la que se pueden plantear mejoras en futuras versiones del sistema. Se identifican a continuación dos líneas de trabajo de especial interés.

La primera es migrar el código de servidor de JavaScript a TypeScript. La base de código actual de las Cloud Functions, escrita en JavaScript puro, no tiene tipado estático, lo que puede provocar errores silenciosos a medida que el número de eventos del *combatLog* y la complejidad de los DTOs crezca. Adoptar TypeScript permitiría

definir interfaces explícitas para cada tipo de evento, detectar incompatibilidades en tiempo de compilación y mejorar la mantenibilidad del código de servidor.

La segunda propuesta mira hacia un escenario de despliegue comercial real. Como se vio en la sección 5.4 tras las pruebas de carga, la solución pasa por aplicar mensajería asíncrona (con Google Cloud Pub/Sub o Redis) y agrupar las escrituras en lotes (*batching*). Esto desacoplaría el motor de combate del sistema de clasificaciones, eliminando el cuello de botella de Firestore y dejando la arquitectura preparada para soportar muchos más jugadores simultáneos.

Por último, se propone un análisis de escalabilidad formal usando Teoría de Colas. El sistema puede modelarse como una cola M/G/k con escalado dinámico, donde k representa el número de instancias concurrentes de Cloud Functions, asumiendo llegadas de tipo Poisson y tiempos de servicio de distribución general. Aplicar este modelo matemático [13] para predecir tiempos de espera bajo tasas extremas de peticiones añadiría una base teórica complementaria al análisis empírico realizado en este trabajo.

7. BIBLIOGRAFÍA

[1] A. Tošić y J. Vičič, "A Decentralized Authoritative Multiplayer Architecture for Games on the Edge," *Computing and Informatics*, vol. 40, no. 3, pp. 522-542, 2021. Disponible en: https://cai.type.sk/content/2021/3/a-decentralized-authoritative-multiplayer-architecture-for-games-on-the-edge/5108.pdf

[2] H. Viitanen, "Deterministic and Synchronous Computation Between Client and Server in Mobile Games", Tesis de Máster, School of Science, Aalto University, Finlandia, 2025. Disponible en: https://aaltodoc.aalto.fi/server/api/core/bitstreams/783d7d7c-8168-451b-a7e9-8de5a67db5a4/content

[3] A. M. Law y W. D. Kelton, *Simulation Modeling and Analysis*, 4.ª ed. McGraw-Hill, 2000.

[4] J. Banks, J. S. Carson II, B. L. Nelson, y D. M. Nicol, *Discrete-Event System Simulation*, 5.ª ed. Prentice Hall, 2010.

[5] R. Goldstein, G. A. Wainer, y A. Khan, "The DEVS Formalism," en *Modeling and Simulation Fundamentals*, Wiley, 2010. Disponible en: https://cell-devs-02.sce.carleton.ca/publications/2013/GWK13/DEVSintro.pdf

[6] S. Ristov, C. Hollaus, and M. Hautz, “Colder Than the Warm Start and Warmer Than the Cold Start! Experience the Spawn Start in FaaS Providers,” *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and Platforms for Implementing and Evaluating Algorithms for Distributed Systems (ApPLIED '22)*, pp. 35–39, 2022. Disponible en: https://dl.acm.org/doi/10.1145/3524053.3542751

[7] Google Cloud, "Firestore: The NoSQL Serverless Database for the Application Developer," *Google Technical Whitepaper*, 2019.

[8] A. B. Semma, M. Ali, M. Saerozi, M. Mansur, y K. Kusrini, "Cloud computing: google firebase firestore optimization analysis," *International Journal of Electrical and Computer Engineering (IJEECS)*, vol. 29, no. 3, pp. 1719-1728, 2023. DOI: 10.11591/ijeecs.v29.i3.pp1719-1728

[9] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Tesis Doctoral, Universidad de California, Irvine, 2000.

[10] T. Bray, Ed., "The JavaScript Object Notation (JSON) Data Interchange Format," IETF RFC 8259, diciembre 2017. Disponible en: https://datatracker.ietf.org/doc/html/rfc8259

[11] M. Jones, J. Bradley, y N. Sakimura, "JSON Web Token (JWT)", *RFC 7519*, Internet Engineering Task Force (IETF), Mayo 2015. Disponible en: https://datatracker.ietf.org/doc/html/rfc7519

[12] Y. Liu, F. Li, y C. Sun, "Dynamic token encryption for preventing permission leakage in serverless architectures," *PeerJ Computer Science*, vol. 11, e2137, 2025. Disponible en: https://peerj.com/articles/cs-3029/

[13] L. Kleinrock, *Queueing Systems, Volume 1: Theory*. Nueva York, NY, EE.UU.: Wiley-Interscience, 1975. Disponible en: https://ia601403.us.archive.org/13/items/in.ernet.dli.2015.134547/2015.134547.Queueing-Systems-Volume-1-Theory.pdf

[14] D. Fett, C. Meyer, y J. Schwenk, "A Comprehensive Formal Security Analysis of OAuth 2.0," en *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Viena, Austria, 2016, pp. 1204–1215. DOI: 10.1145/2976749.2978385

[15] OWASP Foundation, "REST Security Cheat Sheet," *Open Worldwide Application Security Project*. Disponible en: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html