# WP4-A1. Producción de la base de datos para la herramienta de e-Learning.



*"Financiado por la Unión Europea. Las opiniones y puntos de vista expresados solo comprometen a su(s) autor(es) y no reflejan necesariamente los de la Unión Europea o los de la Agencia Ejecutiva Europea de Educación y Cultura (EACEA). Ni la Unión Europea ni la EACEA pueden ser considerados responsables de ellos."*

# Índice

# 1. INTRODUCCIÓN

Este informe resume los resultados finales del WP4-A1: Producción de la base de datos para la herramienta de e-Learning. En el marco general del proyecto RockChain, esta actividad se centra en el diseño y la realización de la capa de datos back-end que apoya el juego serio, de modo que las sesiones multijugador, la dinámica de mercado y los eventos de minería puedan ejecutarse en tiempo real sobre una infraestructura estable y lista para producción.

La tarea crucial de WP4-A1 consiste en definir y desarrollar una base de datos y una arquitectura de back-end que pueda soportar la gestión consistente de perfiles de usuario, sesiones de juego, rondas, recompensas y decisiones dentro del juego según los escenarios de aprendizaje definidos en el proyecto. La implementación actual de dicha arquitectura se basa en *Firebase Firestore* como base de datos central, un conjunto de funciones de Firebase que encapsulan operaciones de escritura sensibles y lógica de juegos, y un *servidor WebSocket* dedicado desplegado en *Google Cloud* Run, gestionando la comunicación en tiempo real y baja latencia entre jugadores.

Este informe tiene un enfoque deliberadamente estrecho y técnico: documenta cómo se estructura e implementa la capa de datos RockChain en producción. Describe los componentes principales de la arquitectura y su alojamiento, la estructura de las colecciones y documentos clave de Fisestore, y los flujos de datos más relevantes que crean y actualizan esta información durante una partida. También resume consideraciones esenciales de producción relacionadas con los estándares de seguridad, el rendimiento y el mantenimiento básico de bases de datos.

Con esta descripción concisa de la capa de datos, WP4-A1 proporciona una referencia práctica que desarrolladores y socios técnicos pueden utilizar para implementar, operar o ampliar la infraestructura de RockChain. La arquitectura resultante formará la base sobre la que tanto la herramienta de e-learning de front-end como las actividades piloto de los paquetes de trabajo posteriores podrán construirse de forma fiable.

# 2. VISIÓN GENERAL DE LA ARQUITECTURA DE DATOS

## 2.1. Conceptos generales

La herramienta de e-learning RockChain se ejecuta en producción sobre una arquitectura de tres capas, centrada en un único proyecto Firebase y un servicio dedicado en tiempo real:

- El cliente móvil React Native (Android/iOS) ofrece la interfaz de usuario y conecta a los estudiantes con el juego.
- El backend de Firebase con Firestore, Functions, Authentication - almacena todos los datos persistentes y encapsula operaciones sensibles.
- El servidor WebSocket desplegado en Google Cloud Run gestiona la comunicación en tiempo real entre los jugadores durante una partida, con baja latencia.

A un nivel general, la arquitectura sigue un modelo híbrido:

- Firestore actúa como una base de datos autorizada que almacena juegos, usuarios, bloques, productos, mercados y recompensas.
- Las Firebase Functions actúan como una puerta de enlace controlada para escrituras críticas, como crear juegos, unirse a partidas y actualizar información de rondas y recompensas, y aplican reglas de validación y negocio en el lado del servidor.
- El servidor WebSocket mantiene el estado en memoria de cada partida activa y transmite eventos en tiempo real como inicio de ronda, datos de mercado, problemas de minería y resultados a todos los clientes conectados, almacenando los resultados relevantes solo en puntos bien definidos de Firestore.

Dado que todos los componentes comparten la misma identidad y contexto de seguridad de Firebase, la autenticación, la autorización y el acceso a los datos son consistentes en toda la pila. Esta arquitectura puede replicarse para desarrollo y pruebas con un proyecto y una implementación diferente de WebSockets en Firebase, manteniendo la estructura exacta pero utilizando credenciales y URLs no de producción.

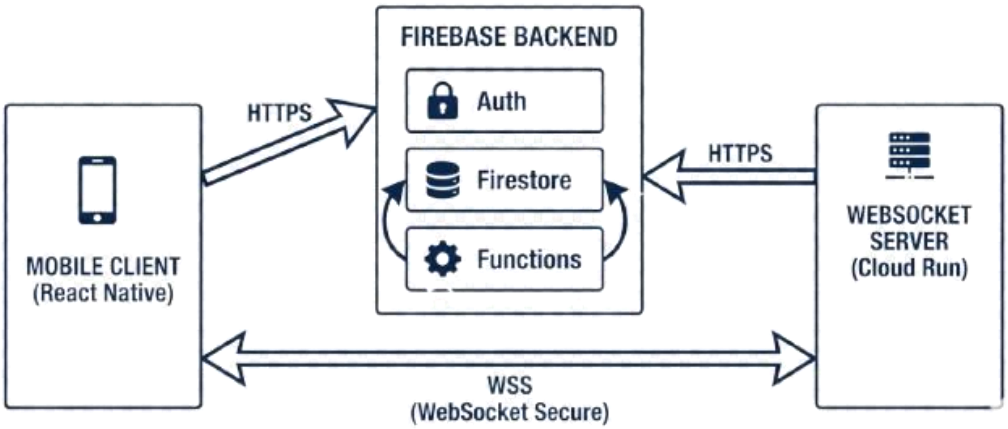


*Figura 1: Diagrama de arquitectura*

Esta separación de responsabilidades permite a RockChain combinar un almacenamiento duradero y bien estructurado de lo ocurrido en cada sesión con actualizaciones rápidas basadas en eventos de lo que ocurre en ese momento durante la partida.

## 2.2. Componentes principales

### 2.2.1. Cliente móvil React Native

La aplicación móvil React Native es **el único punto de entrada para los usuarios**. El propósito es presentar la interfaz del juego y responder al estado del backend, mientras que la propia aplicación no gestiona la persistencia.

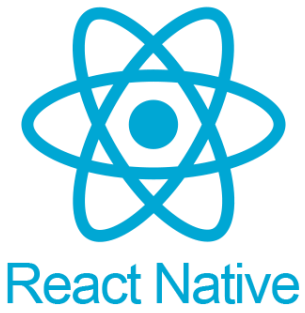


*Figura 2: React Native*

En producción, el cliente móvil:

- Se autentica con Firebase Authentication y obtiene una identidad de usuario (*userId*), que se usa de forma consistente en todo el backend.
- Se une a los juegos mediante llamadas dedicadas a funciones de Firebase, como crear o unirse a una partida, que sirven para validar la solicitud y luego actualizar Firestore.

- Abre una conexión WebSocket al endpoint Cloud Run una vez que el usuario está en una partida, y envía el *gameId,* el userID *autenticado* y la información básica del jugador para que pueda colocar el socket en la sala de juego correspondiente.
- Suscríbete a eventos de backend (ya sea de los oyentes Firestore o de los mensajes WebSocket) y mapéalos al estado local de la aplicación, que luego se renderiza en la pantalla.

El cliente no escribe directamente en Firestore para operaciones críticas. En lugar de:

- Las Firebase Functions deben activarse siempre que se requiera un cambio duradero en el modelo de datos; ejemplos incluyen actualizar el estado del juego y escribir recompensas.
- Emiten eventos WebSocket, como *'listo para jugar'*, *'comprar producto'*, *'enviar respuesta de minería'*, que se procesan en el servidor y persisten cuando corresponde.

Este diseño mantiene la aplicación móvil como una capa delgada y reactiva centrada en la interacción del usuario, con todas las decisiones autorizadas tomadas en el backend sobre el estado del juego.

### 2.2.2. Backend de Firebase: Firestore y Funciones

El backend de Firebase consiste en Cloud Firestore con Firebase Functions, que juntos proporcionan una capa de datos segura y escalable.


*Figura 3: Base de fuego*

#### *Cloud Firestore (almacenamiento de bases de datos y de referencia)*

Firestore almacena cualquier información que necesite para sobrevivir a una conexión WebSocket y estar disponible para análisis o recuperaciones. Los siguientes se encuentran entre los almacenados en RockChain:

- Estado del juego en la colección *de juegos*: Cada documento representa una partida. Incluye el código del juego, lista de jugadores, ronda actual y estado, industria ganadora por ronda, temporizadores y recompensas asignadas.
- Estado del usuario en la colección *de usuarios*: cada documento almacena el perfil de usuario - nombre de visualización, avatar - y datos por partida: rockCoins, residuos acumulados, industria seleccionada, inventario

- Minería y datos "inspirados en blockchain" en *blockchain*/{*gameId*}/*bloques*: para cada juego, esta subcolección almacena problemas de minería, respuestas recibidas y el ganador final de cada bloque.
- Catálogo de productos en la *colección de productos: plantillas para los mármols disponibles (tipo, tipo de canica y descripción), utilizadas para aumentar las ofertas de mercado por ronda.*
- *Mercado:* configuración de mercado por partida. La producción es una visión organizada de los parámetros del mercado, por ronda y tipo de producto. Esto es útil para reconstruir o mostrar el mercado en contextos distintos a las listas planas en tiempo real.

En este caso, Firestore está optimizado para lecturas cortas y frecuentes por parte de un cliente móvil: estado actual del juego, inventario actualizado, etc., y escrituras controladas desde el código backend que garantizan la integridad y consistencia del modelo del juego.

### *Firebase Functions (lógica de negocio y escrituras controladas)*

Las Firebase Functions centralizan las principales reglas de negocio y las modificaciones sensibles de Firestore. El cliente no escribe directamente en documentos críticos como juegos o datos de juegos de usuario, sino que llama Funciones que:

- **Crear juego**, *crearJuego*: crea un documento en los juegos con el código del juego, el host, el estado, el número máximo de jugadores y otros campos estándar
- Permitir que un usuario **se una a una partida**, por ejemplo: *joinGame* comprueba si existe un juego y es accesible; añade el usuario a la lista de jugadores; *actualiza el conteo de jugadores* y crea/actualiza sus valores iniciales: *rockCoins*, *desechos*, *estado*, *marcas de tiempo*....
- **Actualiza** los campos sensibles dentro del juego: por ejemplo, información de la ronda, resultados agregados o recompensas activadas por eventos del cliente o como resultado de procesos de backend al final de una ronda.

```
exports.createGame = onCall({
  memory: '256MiB',
  timeoutSeconds: 60,
  region: 'us-central1',
  minInstances: 0,
  maxInstances: 10,
  concurrency: 80,
  retry: false,
  ingressSettings: 'ALLOW_ALL'
}, async (request) => {
  const userId = request.auth?.uid;
  const gameCode = generateGameCode();

  if (!userId) {…
  }

  const gameData = {…
  };

  try {…
  } catch (error) {
    console.error('❌ Error al crear el juego:', error);
    throw new Error(`Failed to create game: ${error.message}`);
  }
});

function generateGameCode() {
  const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
  let code = '';
  for (let i = 0; i < 6; i++) {
    code += characters.charAt(Math.floor(Math.random() * characters.length));
  }
  return code;
}
```

*Figura 4: función createGame*

Al enrutar estas operaciones a través de Funciones, el proyecto:

- Garantiza una validación y reglas consistentes en cada sesión.
- Mantiene las Reglas de Seguridad Firestore más simples, ya que las decisiones complejas permanecen en código backend en lugar de grandes conjuntos de reglas.
- Reduce el riesgo de escrituras maliciosas o inconsistentes por parte de clientes no confiables.

En la práctica, el camino autoritativo siempre es: *Cliente → Función Firebase / lógica backend → Firestore*.

### 2.2.3. Servidor WebSocket en Google Cloud Run (capa en tiempo real)

El servidor WebSocket es un servicio construido con Node.js y se ejecuta como contenedor en Google Cloud Run. Su función principal es mantener a todos los jugadores conectados durante el juego, con una comunicación rápida y bidireccional sin interrupciones.

¿Qué hace exactamente? Aquí tienes:

- Garantiza que la comunicación en cada partida fluya en tiempo real. Esto incluye notificaciones como quién se une, quién está listo, cuándo empieza una ronda, cómo va el mercado, nuevos problemas de minería, resultados y cuándo termina cada ronda o toda la partida.

- Lleva un registro de cada partida activa en memoria. Esto le permite coordinar lo que ocurre sin sobrecargar Firestore. Por ejemplo:
    - Quién participa y si está preparado.
    - En qué ronda están y cuánto tiempo falta para que termine.
    - Cómo van los inventarios y el mercado en ese momento.
    - Qué problemas de minería están activos y qué respuestas se han enviado.
- También guarda los resultados en Firestore, pero solo en momentos clave: cuando alguien gana un bloque, al final de una ronda o cuando se asignan recompensas. A veces lo hace directamente, y otras veces lo pasa a funciones de Firebase.

Además, como funciona en Cloud Run, el servicio se escala según cuántos juegos estén activos. Y todo está conectado a través de un endpoint WebSocket seguro (WSS), dentro del mismo proyecto Firebase donde están Firestore y Functions.

En resumen:

- El servidor WebSocket te dice lo que ocurre en tiempo real dentro del juego.
- Firestore y Funciones almacenan el historial oficial: qué ocurrió, cómo va el juego y en qué estado se encuentran los jugadores.

Gracias a esta división, RockChain puede ofrecer juegos multijugador ágiles y fluidos, sin sacrificar el control ni la grabación de lo ocurrido.

# 3. DISEÑO DE MODELOS DE DATOS Y BASES DE DATOS

En producción, la capa de datos de RockChain gira en torno a unas pocas colecciones clave en Firestore. Cada uno tiene una función bien definida: están diseñados para ser leídos constantemente por la aplicación móvil, mientras que las escrituras se realizan de forma controlada a través de funciones y procesos de Firebase y backend.

Juntas, estas colecciones almacenan:

- El estado general de cada partido.
- El estado individual de cada jugador dentro de una partida.
- La historia de los acontecimientos mineros.
- La configuración de mercado utilizada en cada ronda.

## 3.1. Colecciones y documentos Firestore

### 3.1.1. Juegos – Estado global del juego

La colección de juegos almacena un documento por partida. Cada *juego*/{*gameId*} contiene datos que describen lo que ocurre en ese juego en general.

Estos son algunos de los campos clave:

- *gameCode*: el código público que los jugadores usan para unirse.
- *players*: la lista de *userIds* que hay en el juego, junto con datos como cuántos jugadores hay (*playerCount*) y el máximo permitido (*maxPlayers*).
- estado: en qué fase está el juego (puede estar *esperando*, *iniciando*, *in_progress*, *rondaTerminando*, *esperandoParaSiguienteRonda* o terminado).
- Ronda: Número de la ronda actual.
- *ganadoraIndustria*: industria que ganó la última ronda completa.
- *roundTime*: marca de tiempo que indica cuándo termina la ronda actual; usada como base para mostrar fichas y realizar transiciones.
- *recompensasAssigned*: un booleano que indica si las recompensas de la última ronda ya han sido escritas.
- *roundRewards*: resumen de las recompensas de cada jugador en la ronda anterior (por industria, extracción de residuos, minería, etc.).

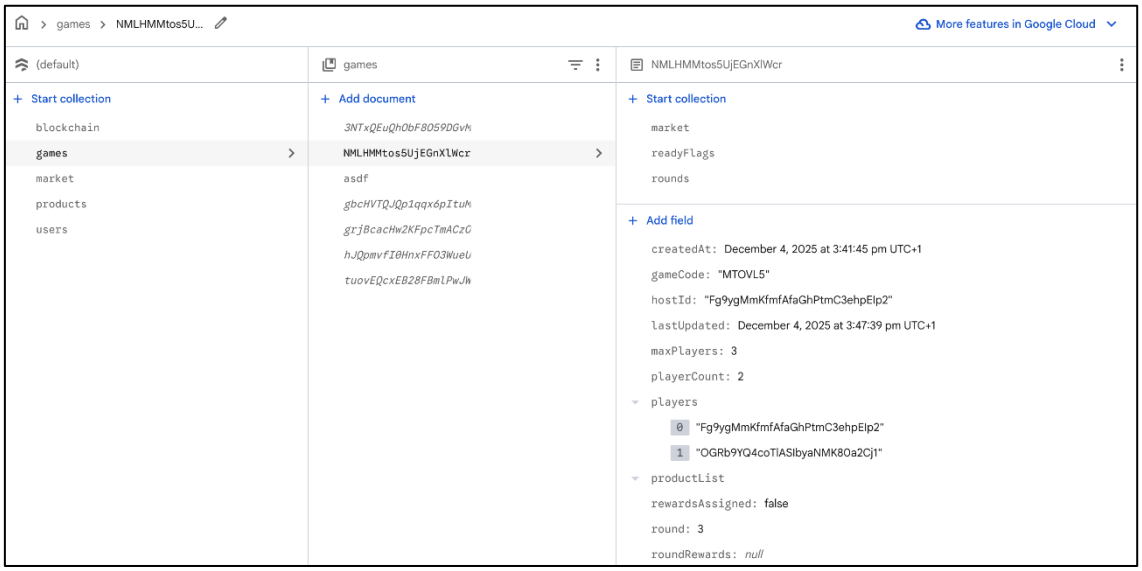


*Figura 5: recopilación de datos de juegos*

Este documento es la principal puerta de entrada al cargar o reanudar una partida. Simplemente leyendo *games*/{*gameId*}, el sistema puede saber quién está jugando, en qué ronda están y si ya se han asignado recompensas.

### 3.1.2. Usuarios – Datos de perfil y por partido

La colección de *los usuarios* almacena un documento por usuario, identificado por su userID de Firebase. Cada documento *usuario*/{*userId*} tiene dos capas de información:

- Perfil de usuario, con campos como *Nombre de usuario*, un avatar opcional (*imagenUrl*) y *lastUpdated*, que sirve como marca temporal para auditorías básicas.
- Datos de juego, organizados dentro de un mapa de juegos, donde cada entrada está asociada a un *gameId*. Para cada juego en el que participa el usuario, se almacenan cosas como las siguientes:
    - Monedas del juego: *rockCoins* y residuos acumulados.
    - La industria que eligieron (*seleccionó la industria*) para obtener recompensas dentro del modelo de economía circular.
    - Su inventario de productos (*productos*), que muestra lo que han comprado en el marketplace.
    - Datos adicionales como *gameCode*, estado, cuándo se unieron (*joinedAt*) y la cola de minería (*miningQueue*).
    - Información sobre el último problema de minería con el que interactuaron (*lastMiningProblem*), incluyendo el *blockId*, detalles del problema y una marca de tiempo.

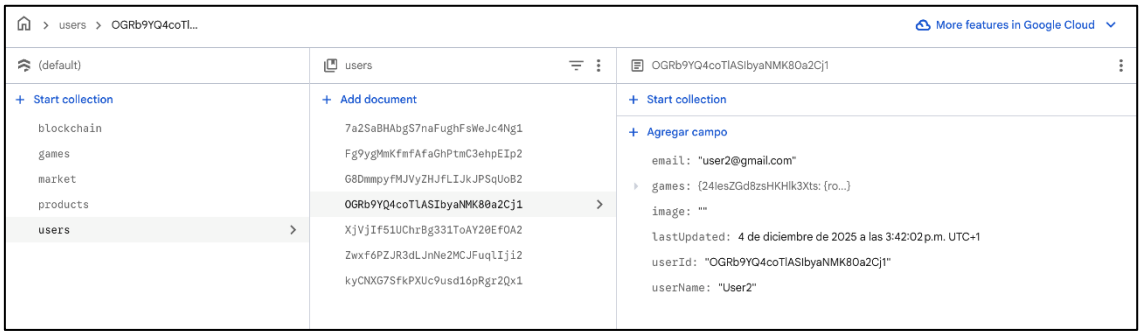


*Figura 6: recogida de datos de los usuarios*

Concentrar toda esta información en un solo documento por usuario facilita mucho las cosas: por ejemplo, puedes conocer instantáneamente la situación actual del jugador en una partida concreta, sin tener que hacer varias consultas o combinaciones complicadas.

### 3.1.3. blockchain/{gameId}/bloques – problemas de minería y bloques ganadores

Para cada juego, la subcolección *blockchain/{gameId}/bloques* lleva un seguimiento de los eventos de minería, utilizando una estructura inspirada en blockchain. Cada *documento blocks/{blockId}* representa un bloque con la siguiente información:

- Problema de minería activa (*activProblema*), que incluye:
  - *problemId*, que coincide con el *blockId*.
  - Un problema matemático sencillo (*mathProblem*) junto con su solución correcta.
  - Fecha de creación (*createdAt*) y detalles de la transacción asociada (como el producto, tipo de operación y usuario implicado).
- Respuestas recibidas (*respuestas*) mientras el problema está activo.
- El número de la ronda (*ronda*) y un *createdAt* que indican cuándo se creó el bloque.
- Un objeto ganador que se llena una vez decidido el ganador de la carrera minera. Incluye: *userId*, *nombre de usuario*, la respuesta dada, si era correcta, el tiempo que tardó en responder (*elapsedTime*) y cuándo respondió (*responseTime*), lo cual es útil para auditar.
- Un indicador *isActive* que indica si el bloque sigue abierto o si ya hay un ganador.

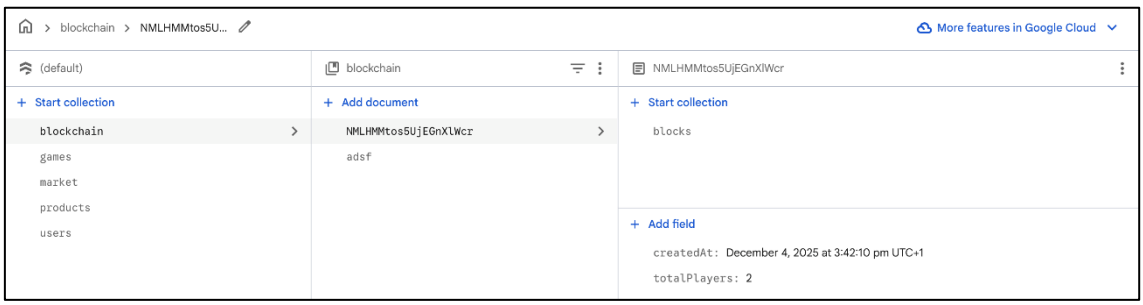


*Figura 7: recopilación de datos de blockchain*

Esta estructura permite a RockChain mantener un registro claro y verificable de los problemas generados, cómo respondieron los jugadores y cómo se asignaron las recompensas relacionadas con la minería.

### 4.1.4. Productos – Catálogo Global de Productos

La colección de productos define el catálogo global de productos de canicas disponibles en el juego. Cada documento funciona como una plantilla estable, con campos como:

- *productId* (por ejemplo: "product001")
- *tipo* ("Bloque" o "Slab")
- *MarbleType* ("Rojo", "Blanco", "Gris", "Negro", "Crema")
- *calidad* ("A", "B", "C")
- Descripción y otros datos estáticos

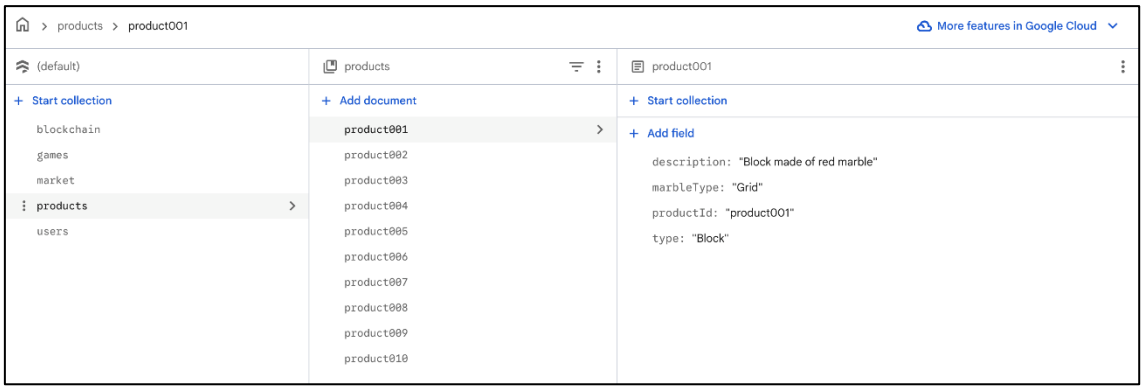


*Figura 8: recogida de datos de productos*

Estas plantillas son utilizadas por la lógica de backend para ensamblar la lista específica de productos que se muestran en el mercado para cada ronda. Sirven como base para generar precios y definir parámetros relacionados con los residuos.

### 4.1.5. Mercado – Estructura del mercado por rondas

Para cada juego, la *colección market/{gameId}* almacena una vista estructurada del mercado. Esta visión se utiliza principalmente para reconstruir cómo se configuró el mercado en diferentes rondas o para análisis posteriores.

El campo principal son rondas: un mapa anidado con las *redondeas estructurales[ronda][mármolDetipo][tipo][calidad] = { precio, desperdicio }*, donde:

- *redondo* es el número redondo,
- *MarbleType* es uno de los tipos de canicas configurados,
- *el tipo* puede ser "Bloque" o "Slab",
- *la calidad* es "A", "B" o "C", cada una con sus valores de precio y nivel de residuos.

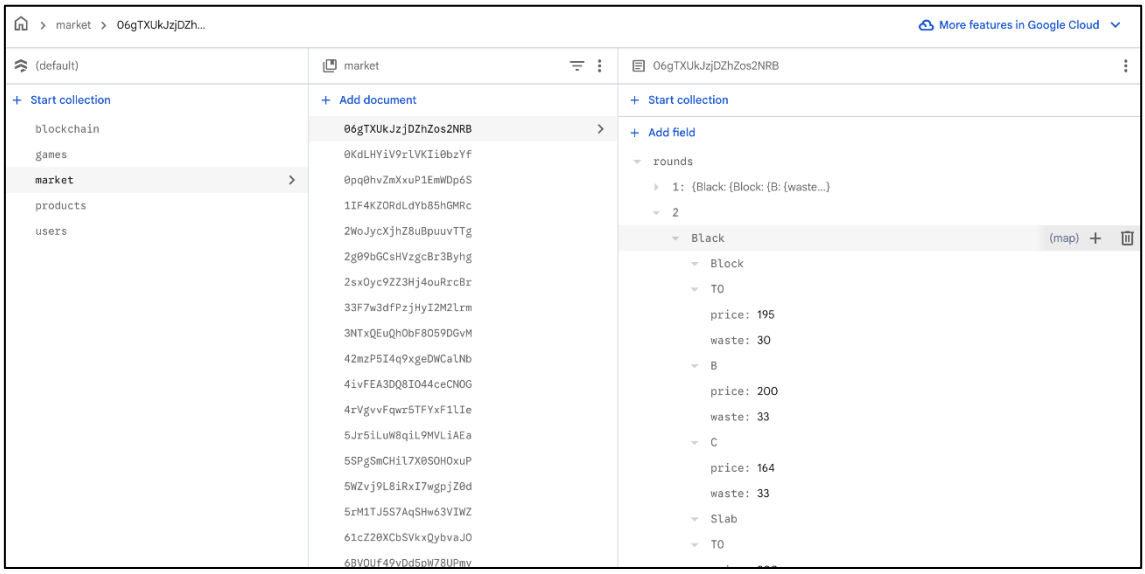


*Figura 9: recopilación de datos de mercado*

Esta estructura funciona como una especie de resumen o visión histórica del mercado. Complementa las listas de productos en tiempo real que se generan y envían vía WebSocket durante el juego. Es especialmente útil para revisar cómo fue el mercado en cada ronda, sin tener que pasar por cada producto individualmente.

## 3.2. Relaciones entre entidades

Aunque Firestore es una base de datos NoSQL orientada a documentos y no gestiona las relaciones como lo haría una base de datos relacional, el modelo RockChain sí define relaciones lógicas claras entre sus colecciones.

### 3.2.1. Juego – Jugadores

Cada documento *games*/{*gameId*} incluye a sus jugadores en el array de un jugador, que contiene los *userIds*.

Al mismo tiempo, cada *documento de usuario*/{*userId*} almacena, dentro del *mapa de juegos*[*gameId*], el estado específico de ese usuario en ese juego. En la práctica, esto representa una relación de muchos a muchos entre juegos y usuarios, implementada con referencias cruzadas.

### 3.2.2. Juego – Bloques de minería

Cada juego tiene su propia subcolección: *blockchain*/{*gameId*}/*bloques*.

Todos los bloques están vinculados a un único *gameId*, y cada uno incluye el número de ronda al que pertenece. Es una relación de uno a muchos: un juego puede tener varios bloques, pero cada bloque pertenece a un solo juego.

### 3.2.3. Juego – Mercado

Cada juego tiene un documento *único/*{*gameId*}, donde el mercado se organiza por rondas. Aunque Firestore no impone relaciones, lógicamente es una relación uno a uno entre *juegos*/{*gameId*} y *mercado*/{*gameId*}.

### 3.2.4. Productos – Mercado / Inventarios

La colección global de productos define las plantillas base de producto.

Tanto el campo de rondas en *market*/{*gameId*} como los arrays de productos dentro *de games*/{*gameId*} se construyen a partir de los valores y parámetros definidos en esos *productIds*. Es una relación de uno a muchos: un solo producto en el catálogo puede aparecer en diferentes mercados y en varios inventarios de jugadores.

### 3.2.5. ¿Cómo se reconstruyen el estado de un juego y de un jugador?

El backend suele seguir este patrón:

- Lee *los juegos*/{*gameId*} para conocer el estado general del juego.
- Lee *users*/{*userId*} y revisa *games*[*gameId*] para ver cómo va ese jugador.
- Lee *market*/{*gameId*} si necesitas una visión completa de los precios y el desperdicio por ronda.
- Lee *blockchain*/{*gameId*}/*bloques* (y filtra por ronda si es necesario) para analizar el historial de minería.

El modelo aplica una ligera desnormalización para reducir llamadas y simplificar las consultas del cliente, sin perder la trazabilidad que los socios del proyecto necesitan para analizar los datos posteriormente.

# 4. FLUJOS CLAVE DE DATOS EN LA PRODUCCIÓN

Más allá del modelo de datos estático, RockChain funciona gracias a una serie de flujos de datos recurrentes que crean y actualizan documentos en Firestore durante una partida. Estos flujos conectan el cliente móvil, las funciones de Firebase, el servidor WebSocket y las colecciones que hemos descrito antes. Juntos, aseguran que todos los jugadores vean el mismo estado del juego, en tiempo real y con trazabilidad.

## 4.1. Creación y incorporación de juegos

La creación de juegos y la incorporación de jugadores dependen de las funciones de Firebase que realizan las primeras escrituras en los juegos y las colecciones de los usuarios. Esto garantiza que cada partida comience con un estado consistente y que los jugadores estén correctamente registrados tanto en el documento global del juego como en su perfil de usuario.

### 4.1.1. Creación de juegos

Cuando un usuario decide alojar un nuevo juego:

- Tras iniciar sesión, el cliente llama a una función de Firebase (*createGame*).
- La función crea un nuevo documento en *games*/{*gameId*}.
- En *users*/{*userId*}, la Función crea o actualiza la entrada *games*[*gameId*] con valores iniciales.

Este proceso garantiza que tanto el estado general del juego como el estado individual del jugador anfitrión estén bien definidos desde el principio.

### 4.1.2. Unirse a un juego existente

Cuando un jugador se une a una partida existente:

- El jugador proporciona el *código del juego* en el cliente.
- El cliente llama a una Función de Firebase (*joinGame*), que resuelve el *gameCode* en un *gameId*.
- La Función verifica que el juego es accesible y luego:
    - Añade al jugador a *games*/{*gameId*}/*players* e incrementa *playerCount*.
    - Crea o actualiza *usuarios*/{*userId*}/*games*[*gameId*] con sus datos iniciales dentro del juego (monedas, residuos, estado, marcas de tiempo).

```
exports.joinGame = onCall({
  maxInstances: 10,
  memory: '256MiB',
}, async (request) => {
  console.log('🎮 joinGame iniciado');
  console.log('📥 Datos recibidos:', JSON.stringify(request.data, null, 2));

  // Extraer datos de la petición
  const { gameId } = request.data || {};
  const userId = request.auth?.uid;

  console.log(`🎯 Intentando unir usuario ${userId} al juego ${gameId}`);

  // Validaciones básicas
  if (!userId) {…
  }

  if (!gameId) {…
  }

  try {…
  } catch (error) {
    console.error('❌ Error en joinGame:', error);

    // Mensajes de error específicos
    if (error.message.includes('Game not found')) {
      throw new Error('Game not found. Please check the code and try again.');
    } else if (error.message.includes('Game is full') || error.message.includes('No caben más jugadores')) {
      throw new Error('No caben más jugadores en esta sala. La sala está llena.');
    } else if (error.message.includes('not accepting new players')) {
      throw new Error('Game has already started. Please try another game.');
    } else if (error.message.includes('User not found')) {
      throw new Error('User not found. Please try logging in again.');
    } else {
      console.error('❌ Error no manejado:', error);
      throw new Error('Failed to join the game. Please try again.');
    }
  }
});
```

*Figura 10: unirse Función de juego*

Esto evita escrituras directas de clientes en la colección de juegos y garantiza que todos los jugadores estén registrados bajo las mismas reglas.

### 4.1.3. Conexión a la capa de tiempo real

Una vez que existan los documentos de Fiseary:

- Cada cliente abre una conexión WebSocket al punto final Cloud Run.
- El cliente envía *gameId*, *userId* y datos básicos de perfil.
- El servidor WebSocket registra el socket en la entrada correspondiente *de gameRooms* y comienza a rastrear a ese usuario en su estado de memoria.

En este punto, tanto el estado persistente (Firestore) como el estado en tiempo real (mapas en memoria WebSocket) están alineados, y el juego está listo para pasar de esperar a la primera ronda.

## 4.2. Actualizaciones y rondas de mercado

Para cada ronda, el mercado se genera y gestiona principalmente en el servidor WebSocket, que crea la lista de productos en la memoria y la envía a los clientes a través del evento *economy:state*. Opcionalmente, se conserva una vista agregada de esta

configuración en market/{gameId} para que los precios y los valores de los residuos puedan reconstruirse más tarde sin depender del estado en memoria.

### 4.2.1. Generación de mercado en el servidor WebSocket

Para el *gameId* activo y *la ronda*, el servidor:

- Utiliza las plantillas globales de productos (tipos, tipos de canicas, cualidades, rangos de precio) para generar el conjunto completo de combinaciones posibles.
- Aplica la lógica de precio y desperdicio (calidad A/B/C, Bloque vs Losa, diferencias de canicas y otras reglas del juego).
- Selecciona aleatoriamente un subconjunto de productos para esa ronda y los almacena en memoria como la lista actual de mercado para ese juego.

```
const assignProductsToGame = async ({ gameId, round }) => {
  const productsRef = admin.firestore().collection('products');
  const gameRef = admin.firestore().collection('games').doc(gameId);
  const marketRef = admin.firestore().collection('market').doc(gameId);

  const marbleTypes = ['Red', 'White', 'Gray', 'Black', 'Cream'];
  const productTypes = ['Block', 'Slab'];
  const qualities = ['A', 'B', 'C'];

  const qualityMultipliers = {…
  };

  const calculateProductPrice = (basePrice, productType) => {…
  };

  const calculateProductWaste = (baseWaste, productType) => {…
  };

  try {…
  } catch (error) {
    console.error('Error assigning products to game:', error);
    throw new Error('Failed to assign products to game');
  }
};

module.exports = { assignProductsToGame };
```

*Figura 11: asignar función ProductsToGame*

Esto produce un mercado específico por ronda que refleja tanto el catálogo estático como los parámetros dinámicos del juego actual.

### 4.2.2. Emisión en tiempo real y progresión de rondas

Para cada *ronda*:

- El servidor WebSocket envía un *evento economy*:*state* a todos los jugadores en el juego con una carga útil que contiene la lista de productos disponibles en esa ronda y cualquier metadato relevante.
- En el cliente, el *GameContext* (*GameContextHybridRobust*) almacena esta información en el estado local y la expone a las diferentes pantallas (*mercado*, estadísticas, *cabecera*).

- Los jugadores interactúan con este mercado (comprando productos, cambiando inventarios) a través de eventos WebSocket, mientras que el servidor mantiene la copia funcional del mercado y los inventarios en memoria.

Esta clara división significa que Firestore mantiene una visión duradera y agregada del mercado por ronda, mientras que la capa WebSocket entrega la lista de productos concretos y gestiona interacciones rápidas durante cada ronda.

## 4.3. Minería, validación y recompensas

La minería en RockChain combina interacción en tiempo real con almacenamiento persistente. El servidor WebSocket genera y distribuye problemas de minería, mientras que la *subcolección blockchain*/{*gameId*}/*bloques* mantiene un registro permanente de cada problema, las respuestas recibidas y quién fue el ganador.

Al final de cada ronda, toda esta información se consolida en un conjunto de recompensas por jugador, que luego se escriben de nuevo en Firestore.

### 4.3.1. Creación de bloques y problemas

Cuando se activa un evento minero para una partida y ronda determinada:

- El servidor WebSocket crea un nuevo problema de minería y lo asocia con un *blockId*.
- Crea o actualiza un documento en *blockchain*/{*gameId*}/*blocks*/{*blockId*}.

Esta escritura inicial garantiza que cada evento minero tenga un ancla persistente en Firestore desde el momento en que se crea.

### 4.3.2. Distribución en tiempo real y envío de respuestas

Una vez creado el bloque:

- El servidor emite un *evento mining*:*problem* a todos los jugadores del juego, que contiene el problema y el contexto relevante.
- Los clientes muestran el problema y permiten a los usuarios enviar respuestas dentro de un plazo limitado.
- Cada jugador envía su respuesta mediante un *evento mining*:*submit* WebSocket, que el servidor registra en memoria y puede añadir al array de respuestas en el documento de bloque correspondiente.

Esta interacción modela una **"carrera minera"** donde los jugadores compiten para proporcionar la solución correcta lo antes posible.

### 4.3.3. Validación y persistencia de los resultados de la minería

A medida que llegan las respuestas:

- El servidor evalúa cada intento con la solución correcta.
- Cuando se detecta un ganador válido según las reglas del juego (la **primera respuesta correcta**), el servidor:
  - Determina al jugador ganador.
  - Marca el bloque como cerrado (*isActive* = false).
  - Actualiza *blockchain*/{*gameId*}/*bloques*/{*blockId*} con un objeto ganador que contiene: *userId*, *userName*, su respuesta y si fue correcta, *elapsedTime* y *responseTime* para auditoría.
- El servidor emite entonces un *evento mining*:*result* a todos los clientes, incluyendo información sobre el ganador y cualquier retroalimentación inmediata.

```
// Función para manejar respuestas de minado con tiempo del cliente
const handleMiningSubmit = (blockId, userId, response, userName, clientResponseTime) => {
  console.log(`[MINING][SUBMIT] blockId=${blockId} user=${userId} response=${response} clientResponseTime=${clientResponseTime}`);

  // Definir timestamp actual al inicio de la función
  const now = Date.now();

  // Usar el tiempo del cliente si está disponible, sino calcular del servidor
  let elapsed;
  if (clientResponseTime !== undefined && clientResponseTime > 0) {···
  } else {···
  }

  // Obtener el problema para verificar la respuesta correcta
  const problemData = miningProblems.get(blockId);
  if (!problemData) {···
  }

  const correctAnswer = parseFloat(problemData.activeProblem.solution);
  const userAnswer = parseFloat(response);
  const isCorrect = userAnswer === correctAnswer;

  console.log(`[MINING][SUBMIT] blockId=${blockId} user=${userId} isCorrect=${isCorrect} elapsed=${elapsed}ms`);

  // Inicializar o actualizar respuestas del bloque
  if (!miningResponses.has(blockId)) {···
  }

  const blockData = miningResponses.get(blockId);

  // Verificar si el usuario ya respondió
  const existingResponseIndex = blockData.responses.findIndex(r => r.userId === userId);
  if (existingResponseIndex !== -1) {···
  } else {
    // Añadir nueva respuesta
  const newResponse = {···
  };
  blockData.responses.push(newResponse);
  }

  // Determinar ganador y estado de completado
  const correctResponses = blockData.responses.filter(r => r.isCorrect);
  let winner = null;
  let isCompleted = false;

  if (correctResponses.length > 0) {···
  }
```

*Figura 12: función handleMiningSubmit*

Este flujo garantiza que cada evento de minería tenga un registro rastreable y auditable en Firestore mientras sigue siendo gestionado en tiempo real por el servidor WebSocket.

### 4.3.4. Consolidación de recompensas al final de la ronda

Al final de cada ronda, RockChain recopila todo lo que ocurrió en la fase de mercado y minería para calcular un conjunto único de recompensas por jugador. Estas recompensas se escriben luego en Firestore, actualizando tanto el documento global del juego como el estado individual de cada jugador.

*Desencadenar el final de la ronda*

Cuando expira el tiempo oficial de la ronda (según el *campo roundTime y las marcas de tiempo en* gameAuthorities*), o cuando se cumplen todas las condiciones necesarias, el servidor WebSocket cambia el estado del juego a la fase de cierre de la ronda.*

En ese momento, recopila datos que tiene en memoria, como:

- Bloques extraídos y sus ganadores.
- Inventarios de jugadores.
- Industrias seleccionadas.
- Niveles de residuos y otros indicadores.

*Cálculo de recompensas por jugador*

El backend calcula, para cada *userId*:

- Recompensas de minería: por ejemplo, una cantidad fija de RockCoins por cada bloque minado correctamente.
- Reducción de residuos: basada en la eficiencia de la industria elegida y las reglas del juego (por ejemplo, cuánto desperdicio se elimina adoptando estrategias circulares).

*Escribiendo resultados en Firestore*

En *games*/{*gameId*}, se actualizan los siguientes:

- *ganandoIndustria para la ronda.*
- *roundRewards*[*userId*] = { *industryReward*, *wasteRemoved*, *miningReward*, ... } para todos los jugadores.
- *recompensasAsignado* = verdadero.
- *estado*, que se configura como roundEnd o waitingForNextRound, dependiendo de lo que siga.

En *users*/{*userId*}, dentro de *games*[*gameId*], se actualiza lo siguiente:

- *RockCoins*, sumando las recompensas ganadas en la ronda.
- *Desperdicio*, restando lo que se eliminó (sin bajar de cero, claro).
- Y asegura que el estado persistente del jugador refleje con precisión el resultado de la ronda.

*Notificación a los clientes*

- Una vez que los datos terminan de escribirse, el servidor WebSocket envía un *evento round_end* con un resumen de las recompensas y la industria ganadora.
- Los clientes actualizan su estado local (recompensas, inventario, sector ganador) y muestran los resultados antes de pasar a la siguiente ronda.

Gracias a este flujo combinado, Firestore actúa como el registro definitivo de lo que ganó cada jugador y cómo cambiaron sus recursos, mientras que WebSocket asegura que todo se sienta inmediato y sincronizado en todos los dispositivos.

# 5. SEGURIDAD, PRIVACIDAD Y PREPARACIÓN PARA LA PRODUCCIÓN

El backend de RockChain fue diseñado para que solo los componentes confiables puedan modificar los datos oficiales del sistema, y para que la información sobre usuarios y controladores siga siendo limitada, protegida y fácil de gestionar a lo largo del tiempo. Esta sección resume cómo se gestionan la seguridad, la privacidad y las operaciones básicas en la capa de datos en producción.

## 5.1. Reglas de seguridad y autenticación de Firestore

El entorno de producción RockChain combina autenticación de Firebase, reglas de seguridad Firestore y funciones de Firebase para garantizar que solo los usuarios y servicios autorizados puedan acceder o modificar los datos.

### 5.1.1. Autenticación como puerta de entrada

- Todo acceso al backend requiere autenticación previa con Firebase Authentication (ya sea por correo electrónico/contraseña u otro proveedor compatible).
- A cada usuario se le asigna un userID estable, que se utiliza en Firestore y en el servidor WebSocket.

### 5.1.2. Acceso limitado a datos de perfil y de juegos

- Las reglas de seguridad Firestore aseguran que cada usuario autenticado solo pueda:
    - Lee y actualiza sus propios *usuarios*/{*userId*} documento.
    - Lee datos de los juegos en los que están registrados.
    - No pueden escribir directamente en documentos críticos como *games*/{*gameId*} o *blockchain*/{*gameId*}/*blocks*/{*blockId*} desde el cliente.
- El acceso a los datos de otros jugadores está restringido al mínimo necesario para que el juego funcione (por ejemplo, nombres públicos o clasificaciones).

### 5.1.3. Escrituras controladas por funciones en Firebase

- La mayoría de las operaciones que afectan a la jugabilidad (crear o unirse a juegos, actualizar el estado del juego, guardar recompensas, etc.) solo pueden ejecutarse desde funciones de Firebase con privilegios elevados.
- Las reglas de Firestore se mantienen simples y básicamente verifican que:
    - Los clientes autenticados pueden leer ciertos documentos si tienen permiso.

- Solo las funciones de backend o cuentas de servicio pueden escribir en rutas protegidas (*juegos*, *blockchain*, ciertos campos en *usuarios*, etc.).

#### 5.1.4. Servidor WebSocket como servicio de confianza

- El servidor WebSocket funciona en Cloud Run y utiliza una cuenta de servicio con permisos limitados para:
  - Lee y escribe los documentos necesarios para gestionar rondas, minería y recompensas.
  - Llama a funciones de backend cuando se requiera validación adicional.
- Toda la comunicación entre el cliente y el servidor WebSocket se realiza a través de conexiones seguras (WSS), y cada mensaje recibido se valida en relación con el estado actual en memoria y en Firestore antes de que se apliquen cambios permanentes

Con estas medidas, RockChain garantiza que ningún usuario no autenticado pueda acceder a los datos, y que incluso los clientes autenticados solo puedan leer y enviar intenciones. Las modificaciones reales a la base de datos siempre las controla el backend, asegurando la integridad del juego.

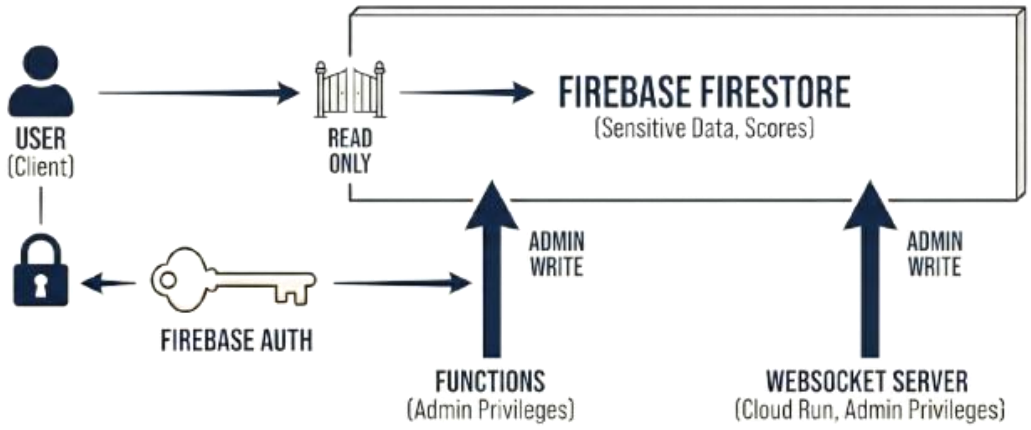


*Figura 13: Cómo funciona la seguridad en Rockchain.*

## 5.2. RGPD y protección de datos

Dado que RockChain se utiliza en actividades piloto con personas reales, su capa de datos fue diseñada de acuerdo con los principios del **Reglamento General de Protección de Datos (RGPD) de la Unión Europea** y las leyes nacionales aplicables.

#### 5.2.1. Minimización de datos y limitación de propósito

- El sistema solo almacena información estrictamente necesaria para:

  - Dirigir sesiones de juego (IDs de usuario, participación, recursos dentro del juego).
  - Evaluar y mejorar las actividades de formación (estadísticas agregadas, registros anonimizados).
- No se recopilan ni almacenan categorías especiales de datos personales (como salud, religión u opiniones políticas).

### 5.2.2. Seudónimo de los jugadores

- A nivel técnico, los usuarios se identifican principalmente por su *userID de* Firebase. Opcionalmente, pueden usar un nombre visible o avatar durante la experiencia de aprendizaje.
- Los socios del proyecto en cada piloto son responsables de gestionar, de forma separada y local, cualquier correspondencia entre un *userID* y la identidad real del participante. Así, el sistema opera usando identificadores seudónimos.

### 5.2.3. Base legal e información para los participantes

- RockChain se utiliza en contextos de formación estructurada (educación de adultos, formación profesional, desarrollo profesional continuo).
- Cada sitio piloto proporciona a los participantes:
  - Un aviso de privacidad que explica qué datos se recopilan en RockChain, para qué fines y durante cuánto tiempo.
  - Datos de contacto claros para ejercer sus derechos como sujetos de datos (acceso, corrección, eliminación, etc.).
- Dependiendo del marco legal del país, la base legal puede ser el consentimiento o el interés legítimo en ofrecer y evaluar formación. Esto queda documentado a nivel de cada piloto por el socio correspondiente.

### 5.2.4. Ubicación de almacenamiento y medidas de seguridad

- Toda la comunicación con el backend (Firestore, funciones, WebSocket) se cifra durante el tránsito usando HTTPS o WSS.
- Los datos también se cifran en reposo, utilizando los mecanismos estándar de Firebase y Google Cloud.
- El acceso a la consola y al entorno de producción de Firebase está limitado a un pequeño grupo de administradores del consorcio, que utilizan autenticación robusta y permisos basados en roles.

### 5.2.5. Retención, anonimización y eliminación

- Los datos del juego se conservan solo mientras sea necesario para:
  - Ejecuta los pilotos.
  - Genera los resultados acordados del proyecto (como informes y estadísticas generales de uso y aprendizaje).
- Una vez finalizado ese periodo, los socios pueden:

- Anonimiza aún más los datos, eliminando cualquier vínculo directo entre el *userId* y las listas de identidad locales.
- Elimina juegos, usuarios o conjuntos de datos completos de la instancia de producción de Firestore (FIRESTORE de producción).

- Si un participante lo solicita, su información puede ser eliminada o pseudonimizada modificando o eliminando el *documento* de sus usuarios/{*userId*} y datos relacionados, tal y como se explica en el aviso de privacidad del piloto.

En resumen, la capa de datos RockChain fue diseñada para operar de manera transparente, limitada y reversible, apoyando los objetivos educativos del proyecto sin perder el control sobre los datos personales, que siempre permanecen en manos de los socios piloto locales.

## 5.3. Respaldos, limpieza y mantenimiento básico

Para estar listo para la producción, el backend de RockChain no solo debe ser seguro, sino también fácil de operar y mantener. La configuración actual incluye procedimientos ligeros para copias de seguridad, limpieza de datos obsoletos y mantenimiento diario.

### 5.3.1. Copias de seguridad y opciones de recuperación

- Firestore ya ofrece redundancia y replicación integradas a nivel de almacenamiento.
- Además, los administradores pueden exportar periódicamente colecciones de claves (como juegos, *usuarios*, *blockchain*, *productos*, *mercado*) a almacenamiento en la nube, ya sea mediante scripts programados o funciones en la nube.
- Estas copias de seguridad sirven tanto para:
  - Recuperar datos en caso de eliminaciones accidentales.
  - Conservar instantáneas congeladas de ciertas fases piloto para fines de documentación o análisis, incluso si posteriormente se eliminan de la base de datos activa.

### 5.3.2. Limpieza de partidas antiguas y datos de pilotos

- Para evitar que los datos crezcan de forma incontrolada y para apoyar la minimización:
  - Los juegos antiguos pueden marcarse como archivados y eliminados, incluyendo sus *subcolecciones* blockchain/{gameId} y *documentos de mercado*/{*gameId*}.
  - Entradas relacionadas en *users*/{*userId*}.*games*[*gameId*] también puede ser eliminado o anonimizado.

- Esta limpieza puede realizarse a través de:
  - Tareas programadas (funciones en la nube o scripts externos).
  - Acciones manuales coordinadas por el socio técnico.

### 5.3.3. Monitorización y controles operativos básicos

- Los paneles de Firebase y Cloud Run ofrecen:
  - Métricas de uso (lecturas, escrituras, ancho de banda).
  - Registros de errores para funciones y el servidor WebSocket.
  - Indicadores de rendimiento que ayudan a identificar consultas que pueden necesitar nuevos índices.
- Durante los pilotos, se realizan revisiones periódicas para:
  - Confirma que no se han superado los límites o cuotas.
  - Detectar configuraciones incorrectas (como reglas de seguridad mal definidas o índices ausentes).
  - Ajusta los recursos o las estrategias de indexación si aumenta el número de usuarios concurrentes.

Gracias a estas medidas, el backend de RockChain sigue siendo ligero y manejable, permitiendo a los socios operar la herramienta tanto durante como después del proyecto sin necesidad de un equipo DevOps dedicado, asegurando al mismo tiempo la integridad y disponibilidad de los datos.

# 6. PRÓXIMOS PASOS Y CONCLUSIONES

El trabajo realizado en WP4-A1 ha dado lugar a una capa de datos concreta y funcional para la herramienta de aprendizaje RockChain. Esto se basa en una arquitectura híbrida que combina Firestore, funciones de Firebase y un servidor WebSocket dedicado que funciona en Google Cloud Run, consumido desde una aplicación móvil integrada en React Native. Esta pila tecnológica ya proporciona una base sólida para gestionar juegos, usuarios, mercados y eventos mineros en tiempo real, con responsabilidades claras para cada componente y un modelo compacto Firestore que refleja los elementos clave del juego y el sistema de recompensas.

A partir de aquí, los siguientes pasos no consisten tanto en realizar cambios estructurales importantes, sino en consolidar y reforzar lo que ya se ha construido. Por un lado, las reglas de seguridad de Firestore para las colecciones principales (juegos, usuarios, blockchain) se están ajustando y probando bajo escenarios realistas para garantizar que el acceso esté bien controlado. También estamos verificando que la separación entre los entornos de desarrollo y producción esté correctamente implementada en todas las partes del sistema: desde compilaciones móviles hasta funciones y el servidor WebSocket. Como parte de esta preparación, se están llevando a cabo sesiones piloto a pequeña escala con usuarios internos y socios del proyecto para confirmar que los flujos de datos funcionan correctamente en condiciones reales de juego.

Además, se están incorporando mecanismos ligeros de monitorización y registro para las operaciones más sensibles, como cambios de rondas, cálculos de recompensas y validación de minería, lo que ayudará a depurar posibles errores durante los pilotos. Los índices de las colecciones más activas también están siendo revisados y ajustados, basándose en patrones reales de consulta, buscando un buen equilibrio entre rendimiento y coste de escritura. También se están definiendo rutinas de mantenimiento sencillas para limpiar sesiones antiguas y exportar colecciones de claves como juegos, usuarios y blockchain, tanto para copias de seguridad como para análisis.

De cara al futuro, planeamos exponer puntos de entrada bien documentados, ya sea a través de funciones o de la API WebSocket, para que otras actividades dentro del mismo WP4 puedan conectarse a esta capa de datos sin tener que duplicar la lógica. También buscamos aprovechar la información que ya se almacena (como el historial de minería, configuraciones de mercado y recompensas) para informar decisiones pedagógicas en el diseño de escenarios o el análisis de aprendizaje en los siguientes paquetes de trabajo

En resumen, WP4-A1 ha llevado RockChain de un diseño abstracto sobre "qué almacenar" a una infraestructura de datos concreta lista para producción y capaz de soportar juegos multijugador en dispositivos móviles. La arquitectura actual es intencionadamente simple y fácil de entender, pero lo suficientemente robusta como para ejecutar sesiones reales, hacer cumplir reglas básicas de seguridad y registrar de

forma rastreable todo lo que ocurre en cada partida. Al centrar los siguientes pasos en la estabilización, la monitorización y pequeñas extensiones incrementales, en lugar de grandes rediseños, los socios del proyecto pueden tratar esta capa de datos como la columna vertebral fiable de la herramienta: algo que puede desplegarse, mantenerse y adaptarse según sea necesario, tanto durante los pilotos como en una posible evolución futura más allá de la vida útil del Proyecto.