WP4-A1. Production of the database for the e-Learning Tool.

"Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Education and Culture Executive Agency (EACEA). Neither the European Union nor EACEA can be held responsible for them."

Contents

1. INTRODUCTION

This report summarizes the final findings of WP4.A1: Production of the database for the e-Learning Tool. In the overall framework of the RockChain project, this activity focuses on the design and realization of the back-end data layer that supports the serious game so that multiplayer sessions, market dynamics and mining events can run in real time on a stable and production-ready infrastructure.

The crucial task of WP4.A1 is to define and develop a database and back-end architecture that will be able to support consistent handling of user profiles, game sessions, rounds, rewards, and in-game decisions according to the learning scenarios defined in the project. The current implementation of such architecture is based on *Firebase Firestore* as the central database, a set of Firebase functions that encapsulate sensitive write operations and game logic, and a dedicated *WebSocket server* deployed on *Google Cloud Run*, managing real-time, low-latency communication between players.

This report has a deliberately narrow and technical focus: it documents how the RockChain data layer is structured and implemented in production. It describes the principal components of the architecture and their hosting, the structure of key Firestore collections and documents, and the most relevant data flows creating and updating this information during a game. It also summarizes essential production considerations related to security standards, performance, and basic database maintenance.

With this concise description of the data layer, WP4.A1 provides a practical reference that developers and technical partners can use to implement, operate, or extend the RockChain infrastructure. The resulting architecture will form the base upon which both the front-end e-learning tool and the pilot activities of subsequent work packages can be reliably built.

2. DATA ARCHITECTURE OVERVIEW

2.1. General concepts

The RockChain e-learning tool runs in production on a three-layer architecture, centered around a single Firebase project and a dedicated real-time service:

- The **React Native mobile client** (Android/iOS) offers the user interface and links learners to the game.
- **Firebase backend** with Firestore, Functions, Authentication - holds all persistent data and encapsulates sensitive operations.
- The **WebSocket server** deployed on Google Cloud Run handles real-time communication between players during a game, with low latency.

At a high level, the architecture follows a hybrid model:

- Firestore serves as an authoritative database that stores games, users, blocks, products, markets, and rewards.
- Firebase Functions act as a controlled gateway for critical writes, such as creating games, joining games, and updating round and reward information, and apply validation and business rules on the server side.
- The WebSocket server keeps in-memory state of each active game and streams real-time events like round start, market data, mining issues, and results to all connected clients, storing relevant results only at well-defined points in Firestore.

Because all components share the same Firebase identity and security context, authentication, authorization, and data access are consistent across the stack. This architecture can be replicated for development and testing with a different Firebase project and implementation of WebSockets, while maintaining the exact structure but using non-production credentials and URLs.

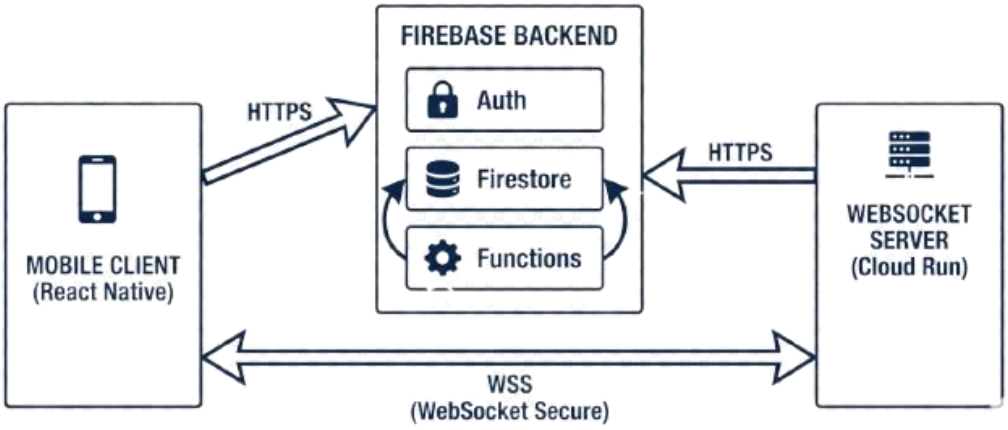

Figure 1: Architecture diagram

This separation of responsibilities lets RockChain combine durable, well-structured storage of what has happened in each session with fast, event-based updates of what is happening at that moment during gameplay.

2.2. Main components

2.2.1. React Native mobile client

The React Native mobile application is **the only point of entry for users**. The purpose is to present the game interface and respond to backend state, while the app itself doesn't handle persistence.

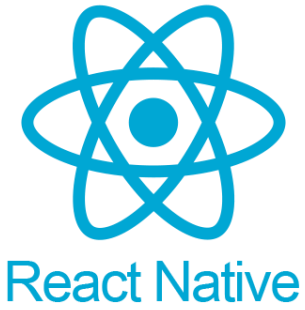

Figure 2: React Native

In production the mobile client:

- Authenticates with Firebase Authentication and gets a user identity (*userId*), that is used consistently throughout the backend.
- Joins games through dedicated Firebase functions calls, such as create or join a game, which serve to validate the request and then update Firestore.
- It opens a WebSocket connection to the Cloud Run endpoint once a user is in a game, and sends the *gameId*, authenticated *userId*, and basic player information so that it can put the socket into the appropriate game room.
- Subscribe to backend events (either from Firestore listeners or WebSocket messages) and map them onto the local app state, which is then rendered onto the screen.

The client does not write directly to Firestore for critical operations. Instead:

- Firebase Functions should be invoked whenever a lasting change to the data model is required, examples include updating game state and writing rewards.
- They emit WebSocket events, such as *'ready to play'*, *'purchase product'*, *'send mining response'*, which are processed on the server and persisted where appropriate.

This design keeps the mobile application as a thin, reactive layer focused on user interaction, with all authoritative decisions being made on the backend about the game state.

2.2.2. Firebase backend: Firestore and Functions

The Firebase backend consists of Cloud Firestore with Firebase Functions, which together provide a secure and scalable data layer.

Figure 3: Firebase

Cloud Firestore (database and authoritative storage)

Firestore stores any information that needs to survive a WebSocket connection and be available for analytics or recoveries. The following are among those stored in RockChain:

- **Game state** in the *games* collection: Each document represents a match. It includes game code, list of players, current round and status, winning industry per round, timers, and assigned rewards.
- **User state** in the *users* collection: each document stores the user profile - display name, avatar - and per-game data: rockCoins, accumulated waste, selected industry, inventory
- **Mining and "blockchain-inspired" data** in *blockchain*/{*gameId*}/*blocks*: for each game, this subcollection stores mining problems, received answers and the final winner of each block.
- **Product catalogue** in the *products* collection: templates for the available marble products (type, marble type, and description), used to build up per-round market offers.
- *Market*: per-game **market configuration**. Output is an organized view of market parameters, by round and product type. This is useful for rebuilding, or displaying, the market in contexts other than the real-time flat lists.

In this case, Firestore is optimized for short, frequent reads by a mobile client-current game status, updated inventory, etc.-and controlled writes from backend code that ensure the integrity and consistency of the game model.

Firebase Functions (business logic and controlled writes)

Firebase Functions centralize the main business rules and sensitive modifications to Firestore. The client does not directly write to critical documents such as games or user game data, but instead calls Functions that:

- **Create game**, *createGame*: creates a document in games with the game code, host, status, maximum players and other standard fields
- Allow a user to **join a game**, for instance: *joinGame* checks if a game exists and joinable; adds user to the list of players; updates *playerCount*, and creates/updates their starting values: *rockCoins*, *waste*, *status*, *timestamps*….
- **Update** the sensitive fields in-game: for example, round info, aggregated results or rewards triggered by client events or as the result of backend processes at the end of a round.

```
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;
}
```

Figure 4: createGame function

By routing these operations through Functions, the project:

- Ensures consistent validation and rules within every session.
- Keeps Firestore Security Rules simpler since complex decisions remain in backend code instead of large sets of rules.
- Reduces the risk of malicious or inconsistent writes from untrusted clients.

In practice, the authoritative path is always: *Client → Firebase Function / backend logic → Firestore*.

2.2.3. WebSocket server on Google Cloud Run (real-time layer)

The WebSocket server is a service built with Node.js and runs as a container on Google Cloud Run. Its main job is to **keep all players connected** during the game, with fast, two-way communication without interruptions.

What exactly does it do? Here you go:

- It ensures that **communication** in each game flows **in real time**. This includes notifications such as who is joining, who is ready, when a round starts, how the market is doing, new mining issues, results, and when each round or the entire game ends.
- It keeps track of each active game in memory. This allows it to coordinate what is happening without overloading Firestore. For example:
    - Who is participating and if they are ready.
    - What round they are in and how long until it ends.
    - How inventories and the market are doing at that moment.
    - What mining issues are active and what responses have been sent.
- It also saves the results in Firestore, but only at key moments: when someone wins a block, at the end of a round, or when rewards are assigned. Sometimes it does this directly, and other times it passes it on to Firebase functions.

In addition, since it runs on Cloud Run, the service scales itself based on how many games are active. And everything is connected through a secure WebSocket endpoint (WSS), within the same Firebase project where Firestore and Functions live.

In summary:

- The **WebSocket** server **tells you what is happening in real time** within the game.
- **Firestore and Functions store the official history**: what happened, how the game is going, and what state the players are in.

Thanks to this division, RockChain can offer multiplayer games that feel agile and fluid, without sacrificing control and recording of what happened.

3. DATA MODEL AND DATABASE DESIGN

In production, RockChain's data layer revolves around a few key collections in Firestore. Each one has a well-defined function: they are designed to be constantly read by the mobile app, while writes are performed in a controlled manner through Firebase functions and backend processes.

Together, these collections store:

- The overall status of each game.
- The individual status of each player within a game.
- The history of mining events.
- The market configuration used in each round.

3.1. Firestore collections and documents

3.1.1. games – global game state

The games collection stores one document per game. Each *games*/{*gameId*} contains data describing what is happening in that game in general.

These are some of the key fields:

- *gameCode*: the public code that players use to join.
- *players*: the list of *userIds* that are in the game, along with data such as how many players there are (*playerCount*) and the maximum allowed (*maxPlayers*).
- status: what stage the game is in (can be *waiting*, *starting*, *in_progress*, *roundEnd*, *waitingForNextRound*, or finished).
- round: number of the current round.
- *winningIndustry*: industry that won the last complete round.
- *roundTime*: timestamp indicating when the current round ends; used as a basis for displaying counters and making transitions.
- *rewardsAssigned*: a boolean indicating whether the rewards for the last round have already been written.
- *roundRewards*: summary of each player's rewards in the previous round (by industry, waste removal, mining, etc.).

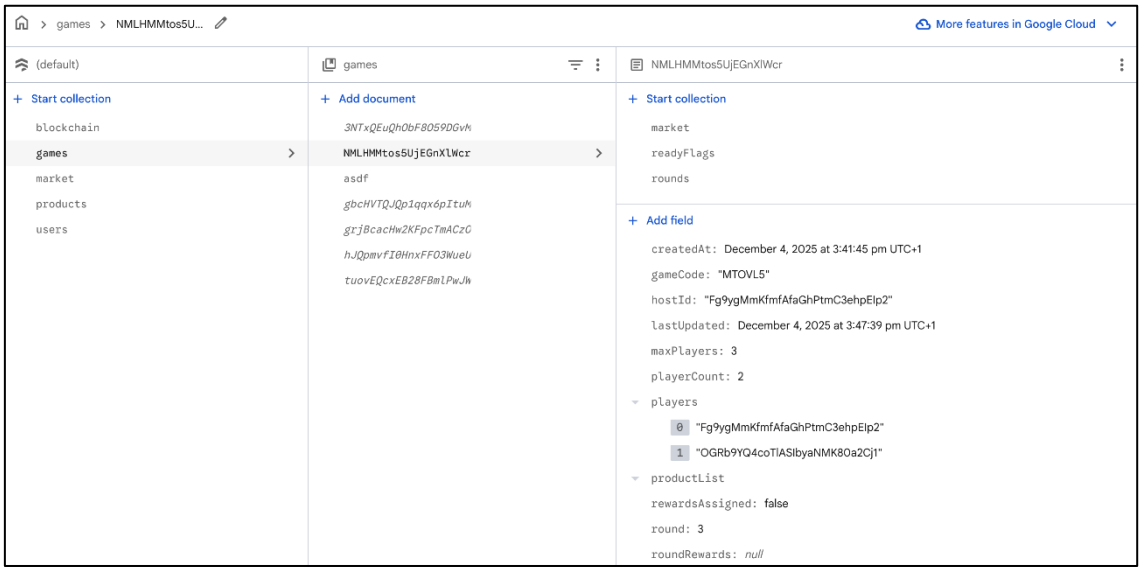

Figure 5: games data collection

This document is the main gateway when loading or resuming a game. By simply reading *games*/{*gameId*}, the system can know who is playing, what round they are in, and whether rewards have already been assigned.

3.1.2. users – profile and per-game data

The *users* collection stores one document per user, identified by their Firebase *userId*. Each *users*/{*userId*} document has two layers of information:

- **User profile**, with fields such as *userName*, an optional avatar (*imageUrl*), and *lastUpdated*, which serves as a timestamp for basic audits.
- **Game data**, organized within a games map, where each entry is associated with a *gameId*. For each game the user participates in, things like the following are stored:
    - Game coins: *rockCoins* and accumulated waste.
    - The industry they chose (*selectedIndustry*) to earn rewards within the circular economy model.
    - Their product inventory (*products*), which shows what they have purchased on the marketplace.
    - Additional data such as *gameCode*, status, when they joined (*joinedAt*), and the mining queue (*miningQueue*).
    - Information about the last mining problem they interacted with (*lastMiningProblem*), including the *blockId*, problem details, and a timestamp.

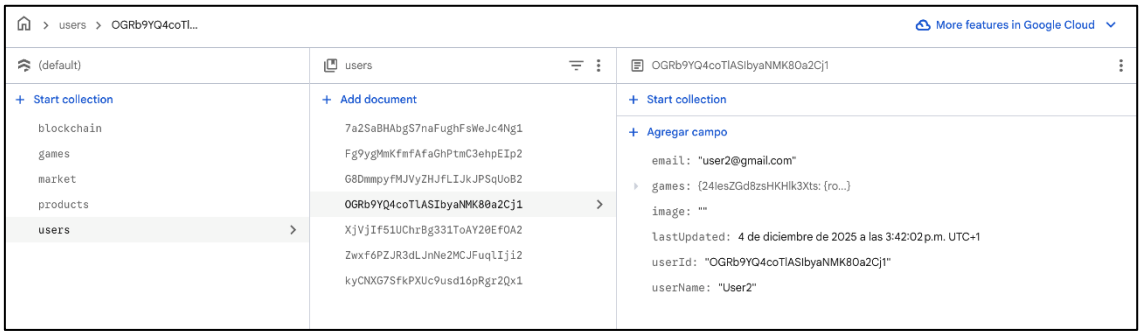

Figure 6: users data collection

Concentrating all this information in a single document per user makes things much easier: for example, you can instantly know the player's current situation in a certain game, without having to make several queries or complicated combinations.

3.1.3. blockchain/{gameId}/blocks – mining problems and winning blocks

For each game, the *blockchain/{gameId}/blocks* subcollection keeps track of mining events, using a structure inspired by blockchain. Each *blocks/{blockId}* document represents a block with the following information:

- Active mining problem (*activeProblem*), which includes:
  - *problemId*, which matches the *blockId*.
  - A simple math problem (*mathProblem*) along with its correct solution.
  - Creation date (*createdAt*) and details of the associated transaction (such as the product, type of operation, and user involved).
- Responses received (*responses*) while the problem is active.
- The round number (*round*) and a *createdAt* indicating when the block was created.
- A winner object that is filled once the winner of the mining race is decided. It includes: *userId*, *username*, the answer given, whether it was correct, the time it took to respond (*elapsedTime*), and when it responded (*responseTime*), which is useful for auditing.
- An *isActive* indicator that signals whether the block is still open or if there is already a winner.

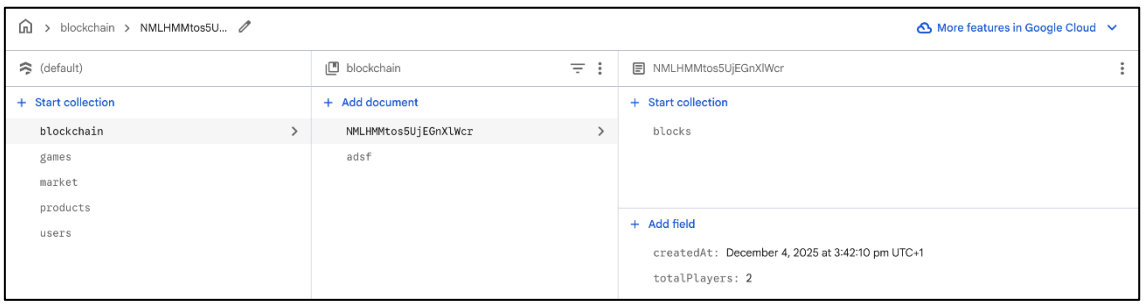

Figure 7: blockchain data collection

This structure allows RockChain to keep a clear and verifiable record of the problems generated, how the players responded, and how the mining-related rewards were assigned.

4.1.4. products – global product catalogue

The products collection defines the global catalogue of marble products available in the game. Each document functions as a stable template, with fields such as:

- *productId* (for example: "product001")
- *type* ("Block" or "Slab")
- *marbleType* ("Red," "White," "Gray," "Black," "Cream")
- *quality* ("A," "B," "C")
- description and other static data

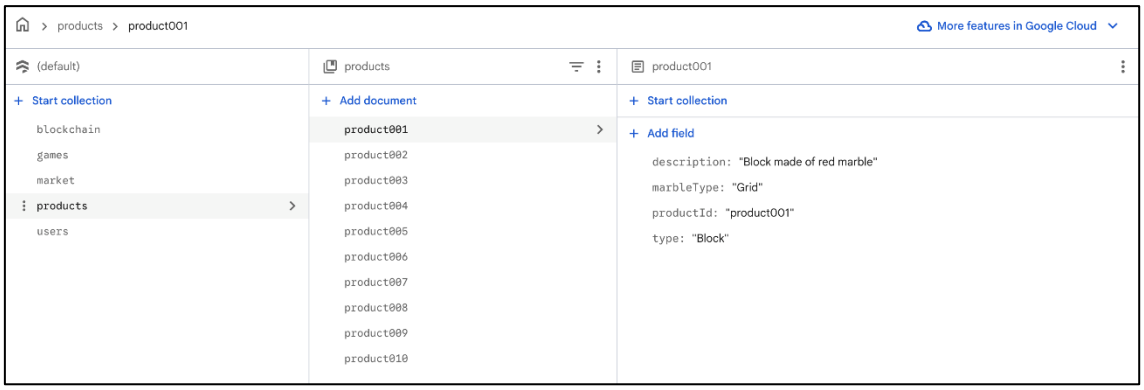

Figure 8: product data collection

These templates are used by the backend logic to assemble the specific list of products displayed in the marketplace for each round. They serve as the basis for generating prices and defining parameters related to waste.

4.1.5. market – market structure by rounds

For each game, the *market*/{*gameId*} collection stores a structured view of the market. This view is mainly used to reconstruct how the market was configured in different rounds or for subsequent analysis.

The main field is rounds: a nested map with the structure *rounds[round][marbleType][type][quality] = { price, waste }*, where:

- *round* is the round number,
- *marbleType* is one of the configured marble types,
- *type* can be "Block" or "Slab,"
- *quality* is "A," "B," or "C," each with its price and waste level values.

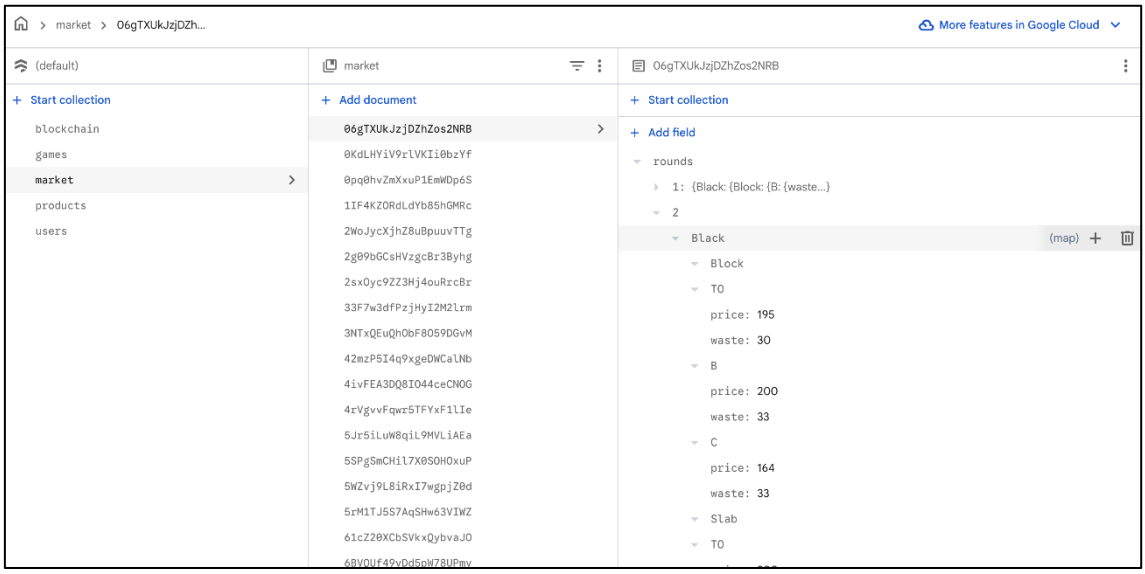

Figure 9: market data collection

This structure works as a kind of summary or historical view of the market. It complements the real-time product lists that are generated and sent via WebSocket during the game. It is especially useful for reviewing how the market was in each round, without having to go through each product individually.

3.2. Relationships between entities

Although Firestore is a document-oriented NoSQL database and does not handle relationships as a relational database would, the RockChain model does define clear logical relationships between its collections.

3.2.1. Game – Players

Each *games*/{*gameId*} document includes its players in a player's array, which contains the *userIds*.

At the same time, each *users*/{*userId*} document stores, within the *games*[*gameId*] map, the specific status of that user in that game. In practice, this represents a many-to-many relationship between games and users, implemented with cross-references.

3.2.2. Game – Mining blocks

Each game has its own subcollection: *blockchain*/{*gameId*}/*blocks*.

All blocks are linked to a single *gameId*, and each includes the round number to which it belongs. It is a one-to-many relationship: a game can have several blocks, but each block belongs to only one game.

3.2.3. Game – Market

Each game has a unique *market*/{*gameId*} document, where the market is organized by rounds. Although Firestore does not impose relationships, this is logically a one-to-one relationship between *games*/{*gameId*} and *market*/{*gameId*}.

3.2.4. Products – Market / Inventories

The global products collection defines the base product templates.

Both the rounds field in *market*/{*gameId*} and the products arrays within *games*/{*gameId*} are constructed from the values and parameters defined in those *productIds*. It is a one-to-many relationship: a single product in the catalog can appear in different markets and in several player inventories.

3.2.5. How are the status of a game and a player reconstructed?

The backend usually follows this pattern:

- Read *games*/{*gameId*} to get the overall status of the game.
- Read *users*/{*userId*} and check *games*[*gameId*] to see how that player is doing.
- Read *market*/{*gameId*} if you need a complete view of prices and waste per round.
- Read *blockchain*/{*gameId*}/*blocks* (and filter by round if necessary) to analyze the mining history.

The model applies slight denormalization to reduce calls and simplify queries from the client, without losing the traceability that project partners need to analyze the data later.

4. KEY DATA FLOWS IN PRODUCTION

Beyond the static data model, RockChain works thanks to a series of recurring data streams that create and update documents in Firestore during a game. These streams connect the mobile client, Firebase functions, the WebSocket server, and the collections we described earlier. Together, they ensure that all players see the same game state, in real time and with traceability.

4.1. Game creation and joining

Game creation and player onboarding depend on Firebase functions that make the first writes to the games and users' collections. This ensures that each game starts with a consistent state and that players are correctly registered in both the global game document and their user profile.

4.1.1. Game creation

When a user decides to host a new game:

- After logging in, the client calls a Firebase Function (*createGame*).
- The Function creates a new document in *games*/{*gameId*}.
- In *users*/{*userId*}, the Function creates or updates the entry *games*[*gameId*] with starting values.

This process ensures that both the overall state of the game and the individual state of the host player are well defined from the start.

4.1.2. Joining an existing game

When a player joins an existing match:

- The player provides the *gameCode* in the client.
- The client calls a Firebase Function (*joinGame*), which resolves the *gameCode* to a *gameId*.
- The Function verifies that the game is joinable and then:
    - Adds the player to *games*/{*gameId*}/*players* and increments *playerCount*.
    - Creates or updates *users*/{*userId*}/*games*[*gameId*] with their starting in-game data (currencies, waste, status, timestamps).

```
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.');
    }
  }
});
```

Figure 10: joinGame function

This avoids direct client writes to the games collection and ensures that all players are registered under the same rules.

4.1.3. Connecting to the real-time layer

Once the Firestore documents exist:

- Each client opens a WebSocket connection to the Cloud Run endpoint.
- The client sends *gameId*, *userId* and basic profile data.
- The WebSocket server registers the socket in the corresponding *gameRooms* entry and begins tracking that user in its in-memory state.

At this point, both persistent state (Firestore) and real-time state (WebSocket in-memory maps) are aligned, and the game is ready to move from waiting to the first round.

4.2. Market updates and rounds

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. Market generation on the WebSocket server

For the active *gameId* and *round*, the server:

- Uses the global product templates (types, marble types, qualities, price ranges) to generate the full set of possible combinations.
- Applies the price and waste logic (quality A/B/C, Block vs Slab, marble differences and other game rules).
- Randomly selects a subset of products for that round and stores them in memory as the current market list for that game.

```
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 };
```

Figure 11: assignProductsToGame function

This produces a round-specific market that reflects both the static catalogue and the dynamic parameters of the current game.

4.2.2. Real-time emission and round progression

For each *round*:

- The WebSocket server sends an *economy*:*state* event to all players in the game with a payload containing the list of products available in that round and any relevant metadata.
- On the client, the *GameContext* (*GameContextHybridRobust*) stores this information in local state and exposes it to the different screens (*market*, *stats*, *header*).

- Players interact with this market (buying products, changing inventories) through WebSocket events, while the server keeps the working copy of the market and inventories in memory.

This clear split means that Firestore keeps a durable, aggregated view of the market per round, while the WebSocket layer delivers the concrete product list and handles fast interactions during each round.

4.3. Mining, validation and rewards

Mining in RockChain combines real-time interaction with persistent storage. The WebSocket server generates and distributes mining problems, while the *blockchain*/{*gameId*}/*blocks* subcollection keeps a permanent record of each problem, the responses received, and who was the winner.

At the end of each round, all this information is consolidated into a set of rewards per player, which are then written back to Firestore.

4.3.1. Block and problem creation

When a mining event is triggered for a given *game* and *round*:

- The WebSocket server creates a new mining problem and associates it with a *blockId*.
- It creates or updates a document in *blockchain*/{*gameId*}/*blocks*/{*blockId*}.

This initial write ensures that every mining event has a persistent anchor in Firestore from the moment it is created.

4.3.2. Real-time distribution and answer submission

Once the block is created:

- The server emits a *mining*:*problem* event to all players in the game, containing the problem and relevant context.
- Clients display the problem and allow users to submit answers within a limited time window.
- Each player sends their answer via a *mining*:*submit* WebSocket event, which the server records in memory and may append to the responses array in the corresponding block document.

This interaction models a **“mining race”** where players compete to provide the correct solution as quickly as possible.

4.3.3. Validation and persistence of mining results

As answers arrive:

- The server evaluates each attempt against the correct solution.
- When a valid winner is detected according to the game rules (the **first correct answer**), the server:
  - Determines the winning player.
  - Marks the block as closed (*isActive* = false).
  - Updates *blockchain*/{*gameId*}/*blocks*/{*blockId*} with a winner object containing: *userId*, *userName*, their response and whether it was correct, *elapsedTime* and *responseTime* for auditing.
- The server then emits a *mining*:*result* event to all clients, including information about the winner and any immediate feedback.

```
// 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) {···
  }
```

Figure 12: handleMiningSubmit function

This flow guarantees that each mining event has a traceable, auditable record in Firestore while still being handled at real-time speed by the WebSocket server.

4.3.4. End-of-round reward consolidation

At the end of each round, RockChain gathers everything that happened in the market and mining phase to calculate a unique set of rewards per player. These rewards are

then written to Firestore, updating both the global game document and the individual status of each player.

Triggering the end of round

When the official round time expires (according to the *roundTime* field and timestamps in *gameAuthorities*), or when all necessary conditions are met, the WebSocket server changes the game status to the round closing phase.

At that point, it gathers data it has in memory, such as:

- Mined blocks and their winners.
- Player inventories.
- Selected industries.
- Waste levels and other indicators.

Computing rewards per player

The backend calculates, for each *userId*:

- **Mining rewards**: for example, a fixed amount of RockCoins for each block mined correctly.
- **Waste reduction**: based on the efficiency of the chosen industry and the rules of the game (e.g., how much waste is eliminated by adopting circular strategies).

Writing results to Firestore

In *games*/{*gameId*}, the following are updated:

- *winningIndustry for the round.*
- *roundRewards*[*userId*] = { *industryReward*, *wasteRemoved*, *miningReward*, ... } for all players.
- *rewardsAssigned* = true.
- *status*, which is set to roundEnd or waitingForNextRound, depending on what follows.

In *users*/{*userId*}, within *games*[*gameId*], the following is updated:

- *rockCoins*, adding the rewards earned in the round.
- *waste*, subtracting what was removed (without going below zero, of course).
- And it ensures that the player's persistent state accurately reflects the outcome of the round.

Notifying clients

- Once the data is finished writing, the WebSocket server sends a *round_end* event with a summary of rewards and the winning industry.
- Clients update their local state (rewards, inventory, winning industry) and display the results before moving on to the next round.

Thanks to this combined flow, Firestore acts as the definitive record of what each player won and how their resources changed, while WebSocket ensures that everything feels immediate and synchronized across all devices.

5. SECURITY, PRIVACY AND PRODUCTION READINESS

The RockChain backend was designed so that only trusted components can modify official system data, and so that information about users and drivers remains limited, protected, and easy to manage over time. This section summarizes how security, privacy, and basic operations are handled at the data layer in production.

5.1. Firestore security rules and authentication

The RockChain production environment combines Firebase Authentication, Firestore security rules, and Firebase functions to ensure that only authorized users and services can access or modify data.

5.1.1. Authentication as an entry gate

- All access to the backend requires prior authentication with Firebase Authentication (either by email/password or another compatible provider).
- Each user is assigned a stable *userId*, which is used in Firestore and on the WebSocket server.

5.1.2. Limited access to profile and game data

- Firestore security rules ensure that each authenticated user can only:
  - Read and update their own *users*/{*userId*} document.
  - Read data from games in which they are registered.
  - They cannot write directly to critical documents such as *games*/{*gameId*} or *blockchain*/{*gameId*}/*blocks*/{*blockId*} from the client.
- Access to other players' data is restricted to the minimum necessary for the game to function (e.g., public names or rankings).

5.1.3. Firebase functions-controlled writes

- Most operations that affect gameplay (creating or joining games, updating game status, saving rewards, etc.) can only be executed from Firebase functions with elevated privileges.
- Firestore rules are kept simple, and basically verify that:
  - Authenticated clients can read certain documents if they have permission.
  - Only backend functions or service accounts can write to protected paths (*games*, *blockchain*, certain fields in *users*, etc.).

5.1.4. WebSocket server as a trusted service

- The WebSocket server runs on Cloud Run and uses a service account with limited permissions to:

- Read and write documents necessary to manage rounds, mining, and rewards.
- Call backend functions when additional validation is required.

- All communication between the client and the WebSocket server is done over secure connections (WSS), and each message received is validated against the current state in memory and in Firestore before any permanent changes are applied

With these measures, RockChain ensures that no unauthenticated user can access the data, and that even authenticated clients can only read and send intentions. Actual modifications to the database are always controlled by the backend, ensuring the integrity of the game.

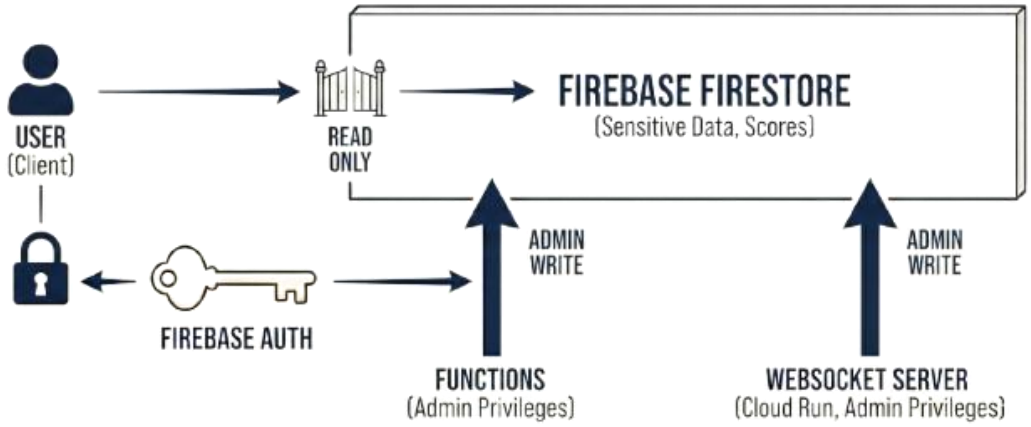

Figure 13: How security works in Rockchain.

5.2. GDPR and data protection

Since RockChain is used in pilot activities with real people, its data layer was designed in accordance with the principles of the **European Union's General Data Protection Regulation (GDPR)** and applicable national laws.

5.2.1. Data minimisation and purpose limitation

- The system only stores information that is strictly necessary for:
  - Running game sessions (user IDs, participation, in-game resources).
  - Evaluating and improving training activities (aggregate statistics, anonymized logs).
- No special categories of personal data (such as health, religion, or political opinions) are collected or stored.

5.2.2. Pseudonymisation of players

- At a technical level, users are primarily identified by their Firebase *userID*. Optionally, they can use a visible name or avatar during the learning experience.
- The project partners in each pilot are responsible for managing, separately and locally, any correspondence between a *userID* and the participant's real identity. Thus, the system operates using pseudonymous identifiers.

5.2.3. Legal basis and information to participants

- RockChain is used within structured training contexts (adult education, vocational training, continuing professional development).
- Each pilot site provides participants with:
    - A privacy notice explaining what data is collected in RockChain, for what purposes, and for how long.
    - Clear contact details for exercising their rights as data subjects (access, correction, deletion, etc.).
- Depending on the legal framework of the country, the legal basis may be consent or legitimate interest in offering and evaluating training. This is documented at the level of each pilot by the relevant partner.

5.2.4. Storage location and security measures

- All communication with the backend (Firestore, functions, WebSocket) is encrypted in transit using HTTPS or WSS.
- Data is also encrypted at rest, using standard Firebase and Google Cloud mechanisms.
- Access to the Firebase console and production environment is limited to a small group of consortium administrators, who use robust authentication and role-based permissions.

5.2.5. Retention, anonymisation and deletion

- Game data is retained only as long as it is necessary to:
    - Run the pilots.
    - Generate the agreed-upon project results (such as reports and general usage and learning statistics).
- Once that period has ended, partners may:
    - Further anonymize the data, removing any direct link between the *userId* and local identity lists.
    - Delete games, users, or entire datasets from the production Firestore instance.
- If a participant requests it, their information can be deleted or pseudonymized by modifying or deleting their *users*/{*userId*} document and related data, as explained in the pilot's privacy notice.

In summary, the RockChain data layer was designed to operate in a transparent, limited, and reversible manner, supporting the educational objectives of the project without losing control over personal data, which always remains in the hands of the local pilot partners.

5.3. Backups, cleanup and basic maintenance

To be ready for production, RockChain's backend must not only be secure, but also easy to operate and maintain. The current configuration includes lightweight procedures for backups, cleaning up obsolete data, and daily maintenance.

5.3.1. Backups and recovery options

- Firestore already offers built-in redundancy and replication at the storage level.
- In addition, administrators can periodically export key collections (such as *games*, *users*, *blockchain*, *products*, *market*) to Cloud Storage, either through scheduled scripts or cloud functions.
- These backups serve both to:
    - Recover data in case of accidental deletions.
    - Preserve frozen snapshots of certain pilot phases for documentation or analysis purposes, even if they are later cleaned from the active database.

5.3.2. Cleanup of old games and pilot data

- To prevent data from growing uncontrollably and to support minimization:
    - Old games can be marked as archived and deleted, including their *blockchain*/{*gameId*} subcollections and *market*/{*gameId*} documents.
    - Related entries in *users*/{*userId*}.*games*[*gameId*] can also be deleted or anonymized.
- This cleanup can be done via:
    - Scheduled tasks (cloud functions or external scripts).
    - Manual actions coordinated by the technical partner.

5.3.3. Monitoring and basic operational checks

- Firebase and Cloud Run dashboards offer:
    - Usage metrics (reads, writes, bandwidth).
    - Error logs for functions and the WebSocket server.
    - Performance indicators that help identify queries that may need new indexes.
- During pilots, regular reviews are conducted to:
    - Confirm that limits or quotas are not exceeded.

- Detect incorrect configurations (such as poorly defined security rules or missing indexes).
- Adjust resources or indexing strategies if the number of concurrent users increases.

Thanks to these measures, RockChain's backend remains lightweight and manageable, allowing partners to operate the tool both during and after the project without the need for a dedicated DevOps team, while ensuring data integrity and availability.

6. NEXT STEPS AND CONCLUSIONS

The work carried out in WP4.A1 has resulted in a concrete and functional data layer for the RockChain learning tool. This is based on a hybrid architecture that combines Firestore, Firebase functions, and a dedicated WebSocket server running on Google Cloud Run, consumed from a mobile app built in React Native. This technology stack already provides a solid foundation for managing games, users, markets, and mining events in real time, with clear responsibilities for each component and a compact Firestore model that reflects the key elements of the game and the reward system.

From here, the next steps are not so much about making major structural changes, but rather about consolidating and reinforcing what has already been built. On the one hand, Firestore security rules for the main collections (games, users, blockchain) are being fine-tuned and tested under realistic scenarios to ensure that access is well controlled. We are also verifying that the separation between the development and production environments is properly implemented in all parts of the system: from mobile builds to functions and the WebSocket server. As part of this preparation, small-scale pilot sessions are being conducted with internal users and project partners to confirm that data flows work correctly under real game conditions.

In addition, lightweight monitoring and logging mechanisms are being incorporated for the most sensitive operations, such as round changes, reward calculations, and mining validation, which will help debug potential errors during the pilots. The indexes of the most active collections are also being reviewed and adjusted, based on actual query patterns, seeking a good balance between performance and write cost. Simple maintenance routines are also being defined to clean up old sessions and export key collections such as games, users, and blockchain, both for backup and analysis.

Looking ahead, we plan to expose well-documented entry points, either through functions or the WebSocket API, so that other activities within the same WP4 can connect to this data layer without having to duplicate logic. We also seek to leverage the information that is already being stored (such as mining history, market configurations, and rewards) to inform pedagogical decisions in scenario design or learning analysis in the following work packages

In summary, WP4.A1 has taken RockChain from an abstract design about “what to store” to a concrete data infrastructure that is production-ready and capable of supporting multiplayer games on mobile devices. The current architecture is intentionally simple and easy to understand, but robust enough to run real sessions, enforce basic security rules, and traceably record everything that happens in each game. By focusing the next steps on stabilization, monitoring, and small incremental extensions, rather than major redesigns, project partners can treat this data layer as the reliable backbone of the tool: something that can be deployed, maintained, and adapted as needed, both during pilots and in possible future evolution beyond the life of the Project.