WP4-A5. Production of interactive RockChain 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 document presents the outcomes of activity WP4.A5 – Production of the interactive RockChain Tool. While previous tasks in WP4 focused on the data layer (WP4.A1), refinement of the e-learning tool (WP4.A2) and the functional specifications (WP4.A3), WP4.A5 describes how the final interactive version of RockChain has been produced, packaged and prepared for distribution and use in real training contexts.

The objective of WP4.A5 is twofold. On the one hand, it provides a consolidated view of the technical architecture of the RockChain tool, showing how the mobile client, backend services and real-time orchestration work together to support multiplayer educational sessions. On the other hand, it documents the production and deployment workflow that leads to ready-to-use Android and iOS builds, a stable backend deployment and basic procedures for hosting, access and maintenance.

The result is a “how it was built” description that can be used by technical staff who need to maintain or extend the tool, and by project partners who require a clear overview of how the interactive RockChain Tool has been turned into a production-ready resource for VET and adult education.

2. SYSTEM ARCHITECTURE OF THE INTERACTIVE ROCKCHAIN TOOL

From a technical point of view, the interactive RockChain Tool is built as a three-layer system:

- A mobile client developed with Expo and React Native.
- A Firebase backend providing authentication, persistent storage and serverless functions.
- An authoritative Node.js + Socket.IO server that coordinates real-time rounds and ensures that all players see a consistent game state.

These components are complemented by an internationalisation layer, environment configuration utilities and basic observability mechanisms.

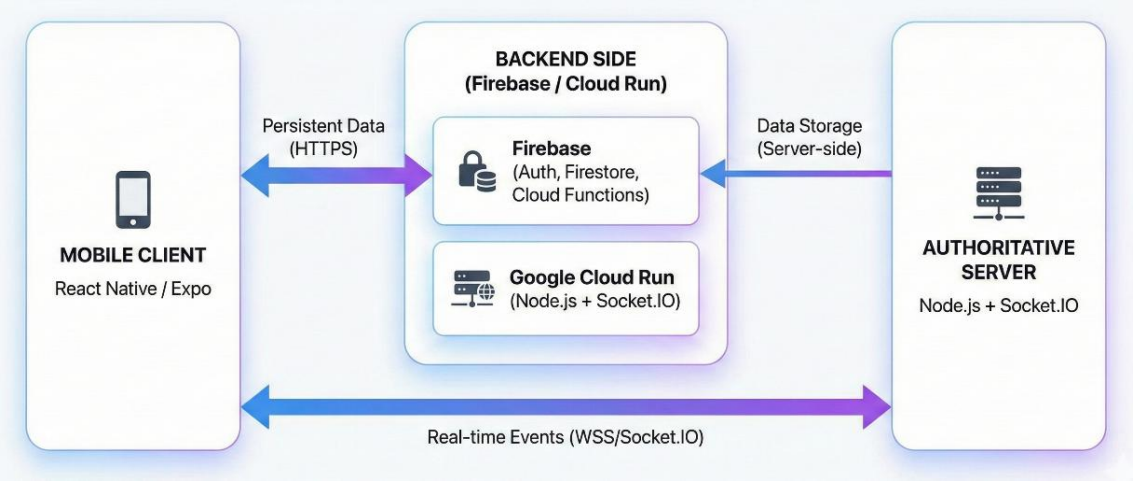

Figure 1: High level architecture.

2.1. Mobile client: frameworks, entry point and navigation

The RockChain mobile client is built as a React application running on React Native, packaged and orchestrated by Expo. In practice, this means:

- All screens and UI elements are written as React function components using JSX.
- React 19 provides the component model, hooks (*useState, useEffect, useContext*, custom hooks), and the Context API that the app uses to share game state across screens.
- React Native 0.79 renders these components as native views on Android and iOS, so the app feels and behaves like a native app while sharing a single JavaScript codebase.

- Expo acts as the wrapper and toolchain: it provides the development server, bundling, EAS build system and access to native services (network, storage, device info) without maintaining separate Xcode/Android Studio projects.

At runtime, everything begins in *App.js*, which is the entry point of the client. This file wires together the core cross-cutting services that every screen needs:

- It injects internationalisation by wrapping the entire component tree in *I18nextProvider i18n={i18n}.* This allows any screen to use translation keys instead of hard-coded strings, and to automatically pick the user's language.
- It wraps the UI in a *NavigationContainer* (from React Navigation), which defines a single navigation tree for the whole app (stacks, tabs and nested flows).
- It encloses the navigation inside two global providers, *GameProviderHybridRobust* and *SimpleMiningProvider,* and renders a *NetworkStatusBanner* at the top level:
  - *GameProviderHybridRobust* is a React Context + hook layer that exposes game-related state (current game and round, players, inventories, timers, rankings, navigation guards).
  - *SimpleMiningProvider* offers a dedicated mini-context for mining problems, making it easier to push and resolve proof-of-work challenges without coupling them tightly to the rest of the UI.
  - *NetworkStatusBanner* listens to connectivity changes and displays warnings when the device is offline or has poor connectivity.

Because these providers sit above the navigation container, any screen, no matter how deep in the stack, can access game state, mining state and network status through hooks, instead of reimplementing that logic locally.

On top of this technical foundation, the app follows a simple, linear navigation flow that mirrors the life cycle of a game. React Navigation is used with a main stack that moves users through these stages:

- *Login*: where a player registers or signs in using Firebase Auth.
- *Game:* a wrapper screen that sets up the context for the selected *gameId* and *userId*.
- *WaitingRoom:* where players gather and see who has joined the game before rounds start.
- *GameTabs:* the main in-game environment used while rounds are in progress.
- *End-of-round and results screens*: where summaries, rankings and final outcomes are shown.

When the user enters *GameTabs*, the layout switches to a tab-based structure. The bottom tab bar is part of the navigation tree (a standard React Navigation tab navigator), but it can be visually hidden so that the experience feels more like a cohesive game than

a typical app with visible tabs. Internally, listeners attached to the Socket.IO events and *GameContextHybridRobust* decide when to switch between modules — for example, automatically opening Mining when a new problem arrives, or navigating to the end-of-round summary when the timer reaches zero — so that learners do not have to handle complex transitions themselves.

During an active round, six main modules are available inside *GameTabs*:

- *Market*: the central module where players buy and sell stone blocks, waste and recycled products. It subscribes to product and price update events and renders dynamic lists based on the current market state.
- *Mining*: a focused view that appears when a proof-of-work challenge is triggered. It displays timed arithmetic problems, collects answers locally and sends them to the authoritative server for validation.
- *Recycle*: the module where parts of the player's inventory can be converted into RockCoins or new resources, operationalising the circular-economy logic in concrete actions and transactions.
- *Stats*: a dashboard implemented with React Native components plus react-native-chart-kit, showing performance indicators and rankings per round and across the whole game.
- *Profile*: a personal area that shows the player's avatar, current progress and language settings, and may expose other basic preferences in future extensions.
- *Stocks*: view summarises quantities and prices for key products, acting as a quick "market snapshot". Players can open it to understand briefly which resources are abundant or scarce before deciding whether to buy, sell or recycle.

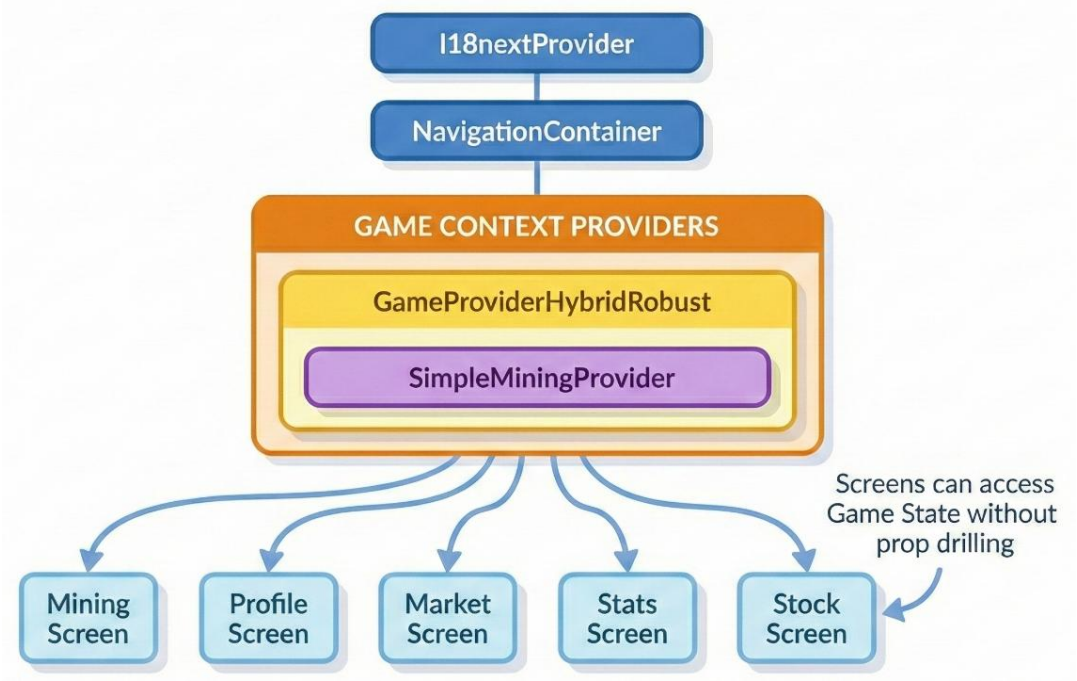

Figure 2: Component and provider tree.

All these screens are standard React components that receive their data primarily from *GameProviderHybridRobust* and the Socket.IO event stream, rather than from ad-hoc local state. This combination, React components and hooks, React Native rendering, Expo's build/runtime, React Navigation for flow control and context providers for shared state, turns the mobile client into a cohesive, declarative front end that is relatively easy to reason about and to extend, while still feeling like a native multiplayer game on learners' devices.

2.2. Client-side state management

On the client, RockChain relies heavily on the React Context API and custom hooks to manage shared state. Instead of letting each screen keep its own copy of players, timers or mining data, the app centralises most of the logic in a few well-defined providers. Screens then become mostly "views": they subscribe to that state and render it, but they do not decide the rules themselves. This is what keeps the app predictable even when many events arrive in real time.

At the core of this layer is *GameContextHybridRobust*. Technically, it is a React Context backed by a provider component and a set of hooks that expose the current game state to any screen. Conceptually, it is the main source of truth on the client. It keeps together:

- The list of players in the current game and their inventories (products, waste, RockCoins).
- The countdown timers and identifiers of the current round, so all screens know how much time is left and which round is active.
- The mining state and rankings, so that mining results and leaderboards are consistent across Market, Stats and other views.
- A synchronised view of the server clock, built from Socket.IO messages, which allows the client to derive remaining time without drifting too far from the authoritative server.
- Several "safe navigation" guards, which prevent race conditions when navigation changes at the same time as new events are processed (for example, avoiding that a screen reads stale data just as the round ends).

Beyond just holding data, *GameContextHybridRobust* also embeds a set of idempotence and resynchronisation mechanisms around the WebSocket connection. Each relevant event carries identifiers such as *roundId* and version, and the context keeps track of what has already been applied. If the connection drops and later recovers, the context can re-request or reconcile the authoritative state from the server instead of blindly trusting whatever was last rendered. This is what allows a player to lock their phone for a moment or briefly lose Wi-Fi without completely breaking their game.

Mining logic is further encapsulated in a dedicated context, often referred to as *SimpleMiningContext* and exposed via *SimpleMiningProvider*. The idea here is to keep a lightweight queue of mining problems and their UI handling separate from the rest of the game state. When the authoritative server emits new mining challenges, they are pushed into this queue; when the user answers or the timer expires, they are processed and removed. Because mining is handled in this isolated context, the rest of the UI can remain responsive even if several problems are created and resolved in quick succession, and mining-related UI (like modals or countdowns) does not have to be wired manually into every screen.

Around these two main contexts, the client uses a set of helper hooks to group specific business rules:

- *useLearningProgress* listens to game events and records educational achievements (e.g. blocks mined, waste reduced) in Firestore and can trigger visual feedback when certain milestones are reached.
- *useNetworkStatus* wraps the NetInfo API and feeds the *NetworkStatusBanner*, so any temporary loss of connectivity is immediately visible to the player.
- *useTutorial* controls onboarding flows for first-time users, deciding when to show explanations or hints and when to step back and let the player interact freely.

Together, these contexts and hooks implement a clear design principle: screens should be as simple as possible, while cross-cutting behaviour (timers, players, mining, learning progress, connectivity, tutorials) lives in shared, testable modules. This makes the app easier to reason about, easier to extend (new screens can reuse the same hooks) and more robust under real-time conditions, because there is a single place where game state is derived from backend events.

2.3. Firebase backend: authentication and persistent data

On the backend, RockChain relies on Firebase as a managed platform that integrates three key services: Authentication, Cloud Firestore, and Cloud Functions. Together, these services enable the secure management of each user's identity, the persistent storage of game data and learning progress, and the controlled execution of sensitive operations directly on the server. Although all the technical details about the data model, flows, and security rules are documented in deliverable WP4-A1, here is an overview of how this infrastructure is used within the interactive tool.

The heart of the system is Cloud Firestore, which acts as RockChain's "long-term memory." Instead of tables like in a traditional database, Firestore organizes information into collections and documents. In the production environment, the main collections are:

- *users*: contains one document per player, with their basic profile and performance in each game (rockCoins, waste, products), grouped by game identifier.
- *games*: one document per game session, where the access code, host, player list, current status, round, and certain key indicators used by both the client and server in real time are stored.
- *market*: product catalogs and dynamic prices per game, kept separate from the main document so as not to overload it with frequent updates.
- *blockchain*: a simplified history of the blocks mined in each game, recording who mined what and when, allowing blockchain-inspired mechanics to be reflected in the data.

For critical operations, clients do not write directly to these collections. Instead, they invoke server functions (Cloud Functions) that run with special privileges, apply validations and business rules, and only then modify the data in Firestore. These functions are detailed in the WP4-A1 report, but within the app they have three main roles:

- Game lifecycle management: functions such as *createGame*, *joinGame*, and *deleteGame* create and delete game documents, verify code uniqueness, and prevent obsolete sessions from the same host.
- Market and mining logic: Functions such as *assignProductsToGame*, *updateProductPrices*, *generateMiningProblem*, and *submitMiningSolution* control how products, prices, and mining challenges evolve, ensuring consistent rules across all games.
- Rewards and profiles: At the end of each round or game, functions such as *assignRoundRewards* and *updateUserProfile* consolidate the results and update user profiles and snapshot documents, maintaining consistency between what was seen on screen and what is permanently saved.

In addition, integration with Firebase includes the authentication system and a small layer of local storage on the device (*AsyncStorage*). Authentication ensures that each request to the backend is linked to a unique user ID, used consistently in Firestore and on the WebSocket server. Local storage allows the app to remember active sessions and save essential data between launches, avoiding interruptions in the event of brief connection drops or accidental app closures.

For details on the complete implementation, including security rules, data flows, and GDPR-related aspects, partners can refer to the WP4-A1 technical report.

2.4. Authoritative server and real-time services

Although Firebase handles storage and server logic securely and persistently, RockChain also needs a layer that responds almost instantly to player actions, manages timers, and decides in real time who wins mining challenges. To meet this need, the project uses a real-time server developed with Node.js and Socket.IO, packaged in a Docker image and deployed on Google Cloud Run. WP4-A1 details this architecture, but here is a summary of how it supports the interactive tool.

In simple terms, this server acts as the referee for each game in progress. For each game, it keeps in memory:

- Which players are connected and in which room.
- Their current inventories and their positions in the ranking.
- The active mining challenge (if any).
- The official status of the round, including the exact time it should end.

Based on this status, the server generates identifiers such as *roundId*, *version*, and *roundEndsAtMs*, which are attached to the events sent. This allows both the client and the backend to know which round they are referring to at all times and avoids confusion with duplicate or late messages.

The game's behavior is structured as a simple state machine with four main phases:

- PLAYING: the round is active; players can buy, recycle, or solve mining problems while the timer runs.
- ROUND_CALCULATING: actions are paused and the server calculates the results.
- ROUND_END: final values are now available; players can view their rewards and positions.
- WAITING_FOR_NEXT_ROUND: the game is paused until the host decides to start a new round or end the game.

As the game progresses through these phases, the server emits different events such as *start_round, ROUND_CALCULATING, round_ended, game_completed, inventory_data, player_rankings_data,* or *economy:state*. The mobile app listens to these events and updates screens, timers, and summaries in real time, ensuring that all players have a consistent view of what is happening.

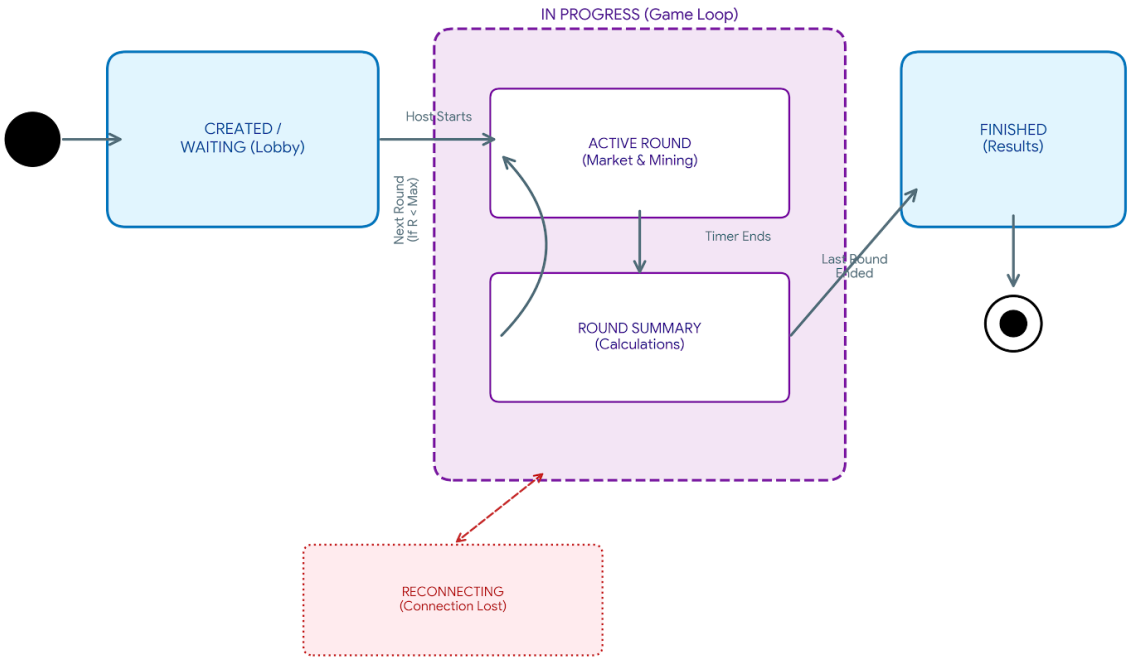

Figure 3: Rockchain state machine.

Since many actions can arrive at nearly the same time (especially at the end of a round), the server uses locking mechanisms and idempotent updates to avoid conflicts. Critical operations, such as assigning rewards or updating inventories, are executed in an orderly fashion, without overlapping, and are marked with round identifiers to ignore repetitions. When the server can access Firebase, it also saves the results of each round (inventories, rewards, economic status), ensuring that the in-memory snapshot and permanently stored data are aligned.

On the client side, all this complexity is encapsulated in a Socket.IO adapter. This adapter automatically selects the correct server (local or Cloud Run), opens and maintains the WebSocket connection, forwards authentication and game information after a reconnection, and translates backend events into the format expected by the React-developed app. Thanks to this middle layer, the server can evolve over time without affecting the user experience or the pedagogical functioning of the tool

2.5. Cross-cutting concerns: internationalisation, configuration and observability

Beyond the main client and backend components, RockChain includes several cross-cutting mechanisms that make the tool usable across countries, easier to deploy in different environments and easier to debug during pilots. Three of them are especially relevant: internationalisation, environment configuration and observability.

2.5.1. Internationalisation and accessibility

From the start, RockChain was designed as a multi-country tool. This meant that language support could not be an afterthought. Instead, the project integrated an internationalisation layer based on i18next.

The core setup lives in *src/i18n/index.js*, which:

- Loads language packs for English (EN), Spanish (ES), German (DE), Croatian (HR) and Romanian (RO).
- Detects the device language or uses the player's explicit choice to decide which pack to activate.
- Exposes an i18n instance that the rest of the app uses through I18nextProvider.

All user-facing texts are stored as translation keys in *src/i18n/locales/*.json*. For every supported language, there is a corresponding JSON file where those keys are mapped to real strings. Components then request texts by key rather than hard-coding English or any other language.

To make the experience persistent, the selected language is saved in AsyncStorage. This means that:

- The first time the app runs, it can default to the device language.
- If the user changes language via the UI, that preference is saved locally.
- On subsequent launches, the app restores the same language without asking again.

Adding a new language is straightforward and does not require touching the core code:

1. Create a new JSON file under *src/i18n/locales/* with the same keys as the existing languages.
2. Register the new language code in *supportedLngs* in *src/i18n/index.js*.
3. Provide translations for all relevant keys.

This design supports one of RockChain's core goals: the tool can be transferred to additional countries and contexts by working on translation files rather than modifying the logic of the app.

2.5.2. Environment configuration and network detection

Because RockChain must run in several execution contexts (local development, staging and production) the project avoids hard-coding URLs or environment assumptions. Instead, it uses a small but explicit configuration layer.

Two modules are central here:

src/config/environment.js

- Computes values such as SOCKET_URL, NODE_ENV, timeouts and debug flags.
- Reads from build-time settings and environment variables to decide, for example, whether the app should talk to a local Socket.IO server, a staging instance or the production Cloud Run service.

src/utils/networkUtils.js

- Detects whether the app is running in a simulator/emulator or on a physical device.
- Based on that information, chooses the appropriate server address:
    - Localhost or a specific LAN IP during development.
    - The Cloud Run URL in staging or production builds.

On top of this, Expo environment variables (such as EXPO_PUBLIC_WEBSOCKET_URL) allow the project to override endpoints in specific build profiles. For example, a production EAS build can hard-wire the WebSocket endpoint to the official Cloud Run URL, while a preview build can point to a testing instance.

The net effect is that the same codebase can be rebuilt for different environments simply by choosing the right build profile and configuration, without editing source files. This directly supports reproducibility and makes it easier for partners to clone or update the deployment.

2.5.3. Observability and resilience

Finally, the architecture includes a minimum set of tools to understand what is happening in real time and to survive temporary network problems. This is particularly important in a classroom context, where Wi-Fi quality may vary.

On the client side:

- *GameContextHybridRobust* logs information about clock drift, duplicate events and resynchronisation attempts to the console. During development and pilot testing, these logs help identify situations where the client and the authoritative server temporarily diverge and how often resyncs are required.
- The combination of automatic socket reconnection and React Context ensures that, when the connection is re-established, the client can request the current authoritative state again instead of relying on stale data.

On the server side:

- A *logMetric* utility emits structured JSON events for key moments such as connection, *round_start*, *round_end*, resync and error. When the server runs on Cloud Run, these events can be captured by Google Cloud Logging or similar tooling, providing a timeline of what happened in each game session.
- The server enforces idempotent event handling using identifiers like *roundId*, version and, where relevant, *operationId*. This prevents duplicated rewards or inconsistent state when messages are retried or arrive late.
- Per-round locks (*roundLocks*, *gameStates*) ensure that critical sections — such as end-of-round calculations — are not executed concurrently for the same game and round.

Taken together, these mechanisms mean that the architecture is not only functional for pilots but also traceable and robust. Developers and technically inclined partners can inspect logs to understand how sessions evolved, while players experience a game that continues to work even when short-lived connectivity issues occur.

3. KEY RUNTIME FLOWS IN THE INTERACTIVE ROCKCHAIN TOOL

Although the previous section explains how RockChain's architecture is built, here we focus on what a person experiences when using the tool: how they log in, how they join or create a game, how the rounds progress, and what kind of responses the system offers. In short, we look at the step-by-step process from the user's point of view.

3.1. Authentication and onboarding

It all starts on the login screen (*LoginScreen*), where users can do two things:

- Create a new account by entering the minimum required information (such as an email address, password, and a name that will be visible in the game).
- Log in with an existing account, reusing their saved credentials.

This screen is directly connected to the Firebase authentication system. When someone enters their details and clicks “Enter,” the app does the following:

1. It sends the credentials to Firebase for verification.
2. If everything is correct, Firebase returns an authenticated user ID and an access token.
3. If there is an error (e.g., incorrect password or email already registered), the app displays a clear message and asks to correct the data.

What happens after successfully logging in?

Three things happen automatically and almost simultaneously:

The profile is created (or retrieved) in the database.

- If it is a new account, the app or a function on the server creates a document in the database with basic information: visible name, email, creation date, default language, etc.
- If it already existed, the profile is reused, along with the history of previous games, if any.

A secure token is obtained for game actions.

- This token allows the app to communicate with the server and access protected features, such as creating or joining games.
- It is also used to connect to the server in real time, so that the system knows who is behind each action.

Access the main game area.

- The app stops displaying the login screen and goes directly to the game area, where the person can:
- Start a new game.
- Join an existing game by entering a code.
- View basic information about RockChain.
- Or view their previous achievements.

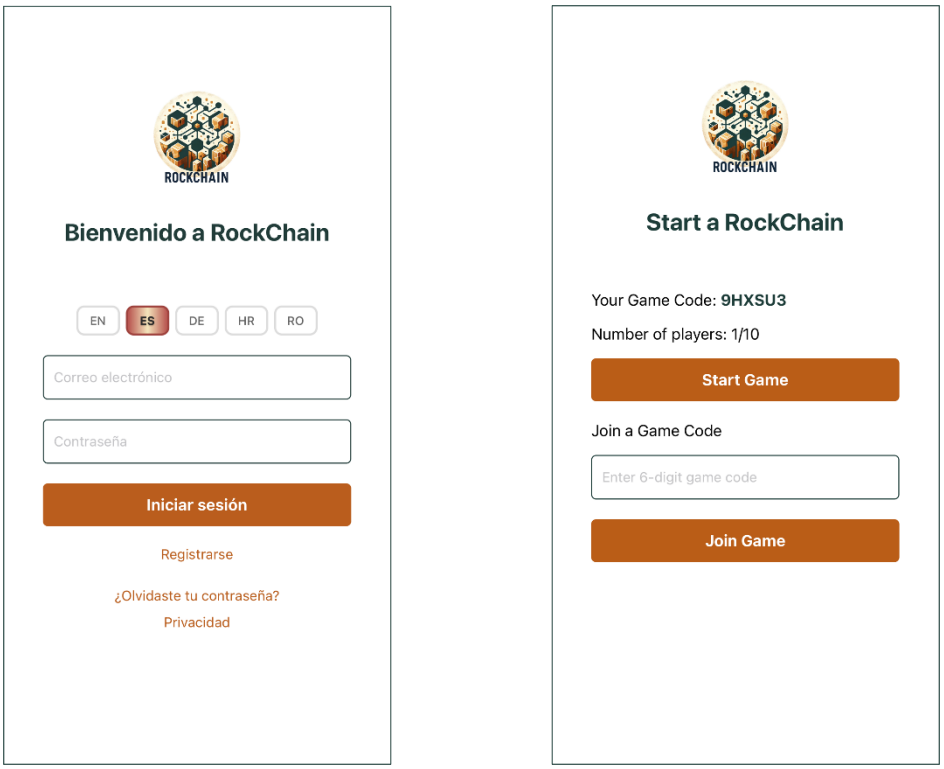

Figure 4: LOGIN and GAME screens.

RockChain does not differentiate between “normal accounts” and “host accounts.” Anyone who is logged in can create a game and becomes a host if they share their code and other players join their session. This makes the process the same for everyone: each person accesses their game space, and then the groups naturally decide which one to use as a shared game.

3.2. Game creation and joining by players

The lifecycle of a game on RockChain is designed to be as simple and seamless as possible. Once a person logs in, the app treats them like everyone else: they are automatically assigned their own game, which they host, and from there they can decide whether to use it as a session for their group or join someone else's using a code.

There is no need to decide between "creating" or "hosting": if you share your code and start the game, you are the host. It's that simple.

3.2.1. Automatic host game on *GameScreen*

Right after successfully logging in, the app takes the person to the game's main screen. At that moment, the application automatically creates a personal game in the background: there is no need to press any "create" button.

In the backend, this process takes care of:

- Checking if that person already had previous active games and, if so, closing them to avoid duplicates.
- Ensuring that the global product catalogue and basic market data are available.
- Creating a new game document with:
  - A unique code to share (such as *ABC123*)
  - The user ID as the host
  - Initial status "*waiting*"
  - The round counter set to zero
  - And any default settings that may be needed

When creation is complete, the app has everything it needs to get started. On screen, the person will see something like.

- "*Your game code: ABC123*" at the top.
- "*Players connected: 1*".
- A "*Start game*" button.
- A field to enter a code if she wants to join another session.

If you decide to use your game as a group session, simply:

- Share your code with your teammates (aloud, on screen, via chat, etc.).
- Wait for everyone to connect (the player counter shows this).
- Press "Start game."
- No further action is required: that personal game becomes the official group game.

3.2.2. Joining someone else's game

The opposite can also happen when you reach this screen, you may already have your own game created, but you want to join another game that a friend has already started.

That's what the "Join a game" area is for. All you have to do is:

- Enter the session code shared by the host.
- Press the "*Join game*" button.

The app will:

- Search for that game using the code.
- Check that it is still open to new players.
- Add the user to the list of participants.
- Create or update their personal game data (*coins, inventory, waste, etc*.).
- Change the context of their game: from that moment on, they become part of the host's session.

From that moment on, the person sees the host's code and the correct number of connected players on the screen. When the group is complete, the host can press "*Start game*", which will take everyone to the waiting room and start the game.

3.3. Waiting room and synchronisation

Once a person decides to use their own game as a host or joins someone else's, all participants are taken from the game's main screen to the waiting room. This screen has a very specific role: to gather the group at a stable starting point and ensure that, when the countdown begins, all devices are perfectly synchronized.

The lobby does not manage its own state separately. Instead, it listens to two key sources of information:

- *GameContextHybridRobust*, which collects:
  - Updates to the game document (*games*/{*gameId*}), such as new players arriving or status changes
  - The current list of players, and the overall game status (waiting, in-game, finished, etc.)
- Real-time server events (*Socket.IO*), which indicate:
  - The start of the countdown,
  - Game phase changes (e.g., going from "*waiting*" to "*playing*")

With this information, the lobby displays only the essentials:

- The list of players connected to that game,
- The current game status (messages such as "*Waiting for players*," "*Ready to start*," "*Starting in 3...2...1...*"),
- Optional indicators that players are ready, if that feature has been enabled

What happens when the host press "*Start game*"?

When the host considers that the group is ready and presses "*Start game*":

1. Your device calls the *startGame* function in the backend, sending the game ID.
2. The server checks that:

- The game is in a valid state (e.g., it is not already in progress and there are enough players)
- Coordinates with the real-time server to prepare for the start of the first turn:
    - A unique round identifier (*roundId*) is generated
    - The exact time when the round will end (*roundEndsAtMs*) is calculated
    - The game is marked as ready to move on to the game phase after the countdown

The server then issues a series of events via *Socket.IO*:

- A countdown start event ("*the round will start in a few seconds*")
- Optional events marking the progress of the countdown ("*3... 2... 1...*")
- An event indicating that the round has officially started

Each waiting room receives these events, and *GameContextHybridRobust* uses the timestamp (*roundEndsAtMs*) to calculate the time remaining on each device. Even if there are slight differences in Wi-Fi connection, all players leave the waiting room and enter the round with the same time limit

The waiting room acts as a synchronization chamber. It ensures that:

- Everyone is connected and visible in the game.
- The host can decide the exact moment to start.
- The whole group starts the round at the same time, thanks to a combination of cloud features and real-time communication.

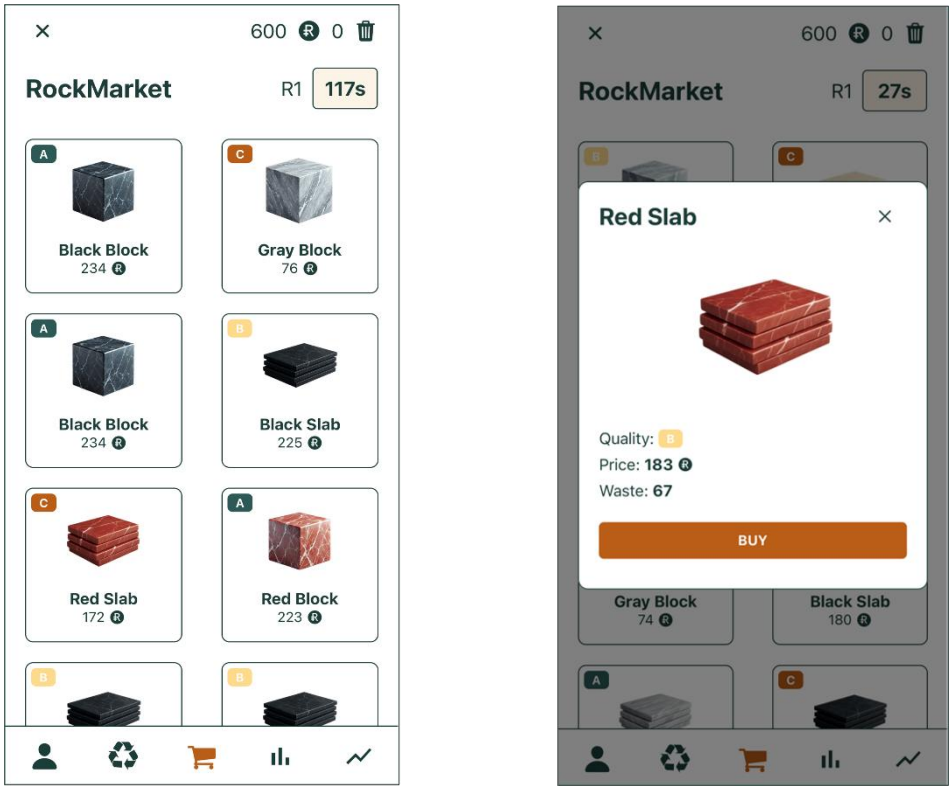

Figure 5: MARKET screen.

3.4. Gameplay during the round: navigating between screens

When a round begins, the app automatically moves all players from the lobby to the main game interface, called ***GameTabs***. From that point on, each person can move freely between several key screens for the duration of the round.

Game time is controlled by a central value called *roundEndsAtMs*, which defines exactly when the round should end. This value is provided by the server in real time, and the game context (*GameContextHybridRobust*) converts it into a local countdown so that all devices have the same time limit, regardless of minor connection delays.

During a typical 120-second round, players can interact with the following modules:

- *MarketScreen*: allows players to buy and sell products (*blocks, slabs, recycled materials*). The product list is automatically updated if the server changes prices or supply.
- *RecycleScreen*: here you can transform part of your inventory into RockCoins or new materials. Each action is validated in the backend, which checks if the rules are followed before applying it.

- *MiningScreen*: offers "proof-of-work" challenges. The app displays math problems with a time limit, the player submits their answer, and the server decides who was correct and who was the fastest.
- *StatsScreen*: displays performance indicators and rankings with graphs updated according to progress.
- *ProfileScreen*: provides a summary of the user's profile, current resources, and RockCoins. It also allows you to change the language. The information comes directly from Firestore.

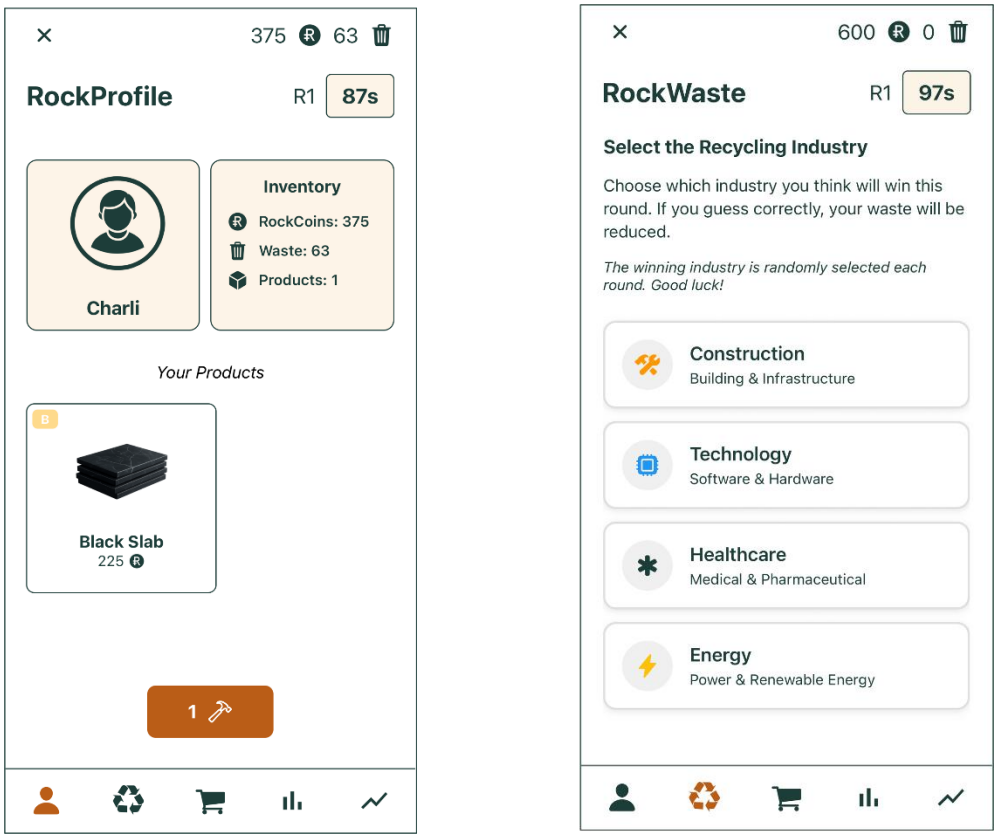

Figure 6: PROFILE and RECYCLE screens.

- *StockScreen*: provides an overview of the market: what products are available, how prices vary, and what resources are scarce or abundant. It is useful for planning before buying or recycling.

One basic principle remains consistent across all these screens: the app never makes final decisions on its own. Every action performed by the player is interpreted as an intention sent to the server. For example:

- *"I want to buy X product at the current price"*
- *"I want to recycle this waste in this industry"*
- *"This is my response to the mining challenge"*

The real-time server receives each action, checks if it is valid according to the current state of the game (inventory, prices, timer, etc.), and:

- Applies it if everything is in order, updating the data in memory (and in Firestore if necessary).
- Rejects it if it does not comply with the rules or arrives too late.

The server then sends the updates to all players: new inventories, rankings, price changes, or mining results.

Thanks to this structure, each round is:

- Interactive: players can move between different screens and make decisions freely.
- Fair and consistent: the same rules apply to everyone, and the game state is kept in sync for the whole group.
- Time-controlled: the *roundEndsAtMs* value ensures that all players finish the round at the same time, regardless of minor differences in their devices or networks.

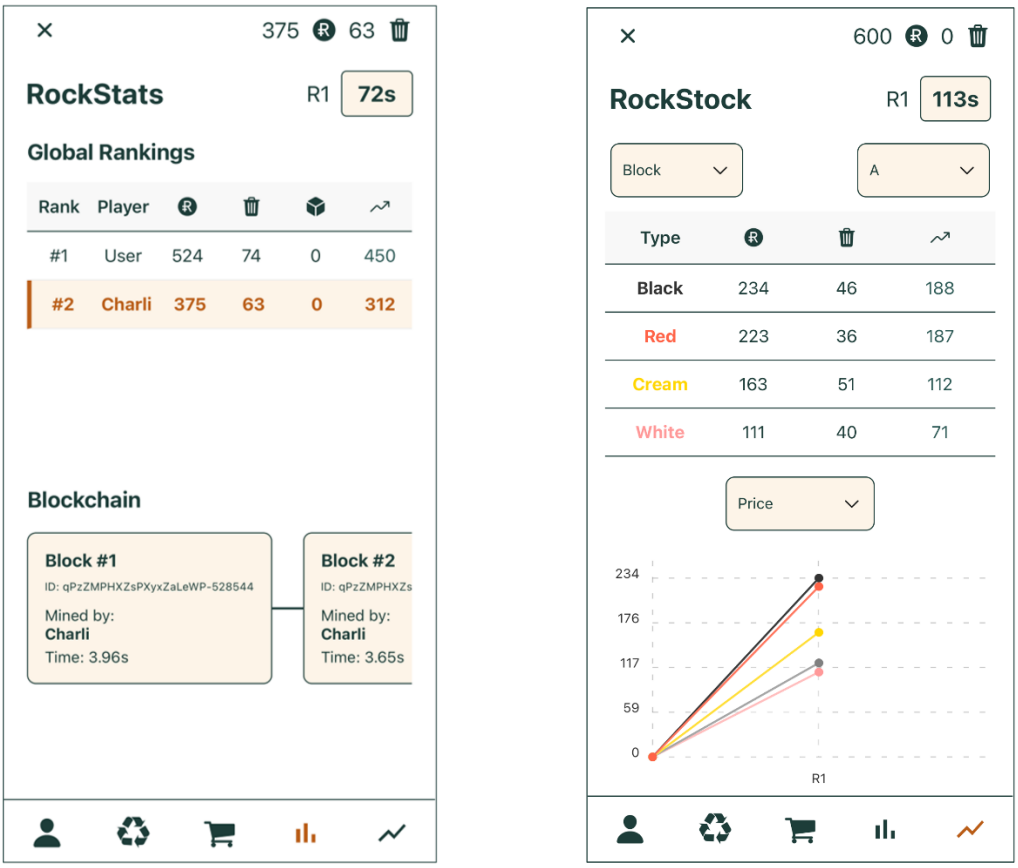

Figure 7: STATS and STOCK screens

3.5. End-of-round calculations and final results

When the round time expires (*roundEndsAtMs*), the system stops accepting new actions from players. From that moment on, control passes completely to the backend, which is responsible for freezing the game state, calculating the results, and displaying the information in a clear and consistent manner.

On the real-time server (Socket.IO), the round closing process follows these steps:

1. **Freeze actions:**
   The server changes the game state to *ROUND_CALCULATING* and stops accepting new moves. Any messages that arrive late are ignored or marked as invalid.
2. **Calculate rewards securely:**
   Using round-level locks and checks to prevent repetition, the server:
   - Processes each action only once
   - Avoids duplicating rewards or counting the same event twice
   - Ensures that simultaneous actions do not generate errors or inconsistent data
3. **Choosing the winning industry:**
   The configured logic is applied to determine which industry won the round, a key piece of information for the feedback players will receive.
4. **Saving the results in Firestore**
   The server updates:
   - The game document in *games*/{*gameId*} (status, round number, scores)
   - The individual statistics in *users*/{*userId*}.*games*[*gameId*]

   If available, Firebase Admin handles these operations securely and efficiently.
5. **Notify devices**
   Once the calculations have been made and the data saved, the server sends a series of events such as:
   - *ROUND_CALCULATING*
   - *ROUND_ENDED*
   - *REWARDS_ASSIGNED*

   These messages allow clients to close the active phase of the game, show that the round is over, and then display the detailed results.

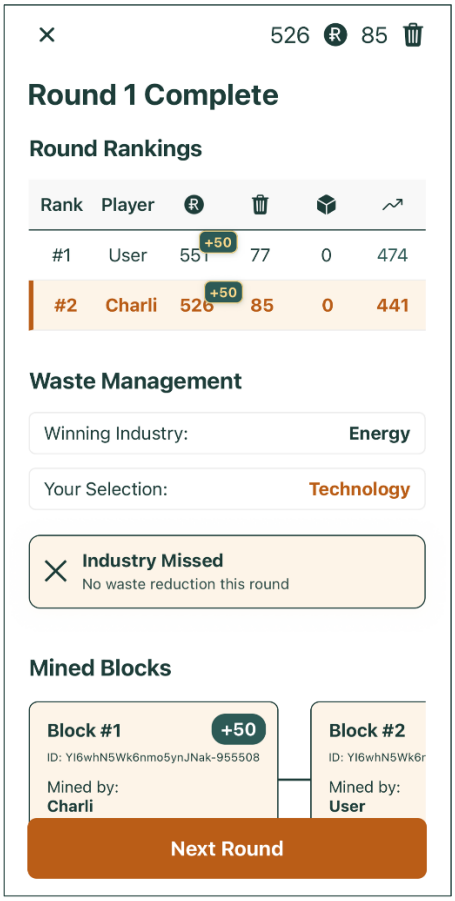

Figure 8: End of round screen.

In the app, the game context (*GameContextHybridRobust*) listens for these events and updates the interface. It switches from the active game view to summary screens, such as:

- *EndOfRoundScreen* shows the winning industry, who solved the mining challenge (if there was one), how each player's inventory and RockCoins changed.
- *GameResultsScreen* (or a similar final screen) shows the overall ranking of players, accumulated statistics from the rounds played, and other useful indicators for discussion or reflection.

This closes the round for the entire group. Since all important data is stored in Firestore, players can:

- return to the main screen and start a new game as hosts with the same or another group, or
- exit the game and continue with other course activities.

Completed games and their data remain available for later use in WP5 (evaluations, analysis, or reviews in future sessions), regardless of what players do next.

Together, this flow completes the cycle that began at login: it goes from authentication to game setup to active experience and finally to a conclusion with clear results. Thanks to the combination of a real-time server, a solid structure in Firestore, and well-designed summary screens, the technical architecture becomes a practical, repeatable, and useful experience in different educational contexts.

4. HOSTING, ACCESS AND DISTRIBUTION

From an operational point of view, the interactive RockChain Tool is delivered as a hosted cloud service plus a mobile app. Training centres do not need to install or maintain local servers; they only need internet access and suitable in at least one of the main platforms.

This section offers a concise, technical view of how the tool is hosted and accessed. For step-by-step, trainer-oriented instructions on preparing and running a session readers can refer to the WP4-A4 “Guideline notes”, so that users can use the tool with ease.

4.1. Backend hosting

The backend is hosted on ***Google Cloud*** under the RockChain project and is composed of the elements described in Section 2:

- *Cloud Firestore*: stores games, users, market data and learning-related records.
- *Firebase Cloud Functions*: implement the authenticated operations that create and manage games, generate mining problems, assign rewards and update user profiles.
- *Node.js + Socket.IO server on Cloud Run*: coordinates real-time rounds and acts as the authoritative timekeeper and referee.

All core data is stored in a European region, aligning the deployment with EU data protection requirements and simplifying compliance for partners operating in different countries. Because these services are fully managed, there is no need for local database servers or custom infrastructure at the education centres: maintenance and scaling are handled at cloud level.

4.2. Mobile app distribution

The RockChain client is distributed as a **mobile application** at least for one of the two main platforms:

- An Android build (APK/AAB), suitable for installation via store listing or direct distribution to managed devices.
- An iOS build (IPA via TestFlight or App Store listing), depending on the distribution strategy agreed with each partner.

You can also find the links and/or QR codes published in the RockChain web area, here: https://rockchain.eu/tool/

In practice, trainers preparing a course can visit the RockChain web space, scan the QR code or follow the link for their platform, and install the current production version of the app on their devices or on institution-owned tablets.

4.3. Access workflow for trainers and learners

For end users, access to RockChain follows a simple, repeatable sequence:

1. **Install the app**
    - Learners and trainers install the RockChain app on their Android or iOS devices, either on personal phones/tablets or on institution-owned devices where the app may be pre-installed.
2. **Create or use an account**
    - On first use, they register or log in via Firebase Auth from *LoginScreen*, as described in Section 3.1.
    - Accounts can be reused across sessions and courses, so users do not need to register again each time.
3. **Join or host game sessions**
    - Once logged in, each player lands in *GameScreen*, where a personal game code is prepared for them in the background.
    - Small groups decide whose code to use: one player acts as host by sharing their code and starting the game, and the others join using the “Join game” input.
    - Trainers can either let learners host their own games (one per group) or act as hosts themselves if they want to run a demo or control timing more tightly.

From a trainer’s perspective, this means that running a RockChain session consists mainly of:

- Ensuring that all participants have the app installed and can log in.
- Organising learners in small groups (each with a host who shares a game code).
- Monitoring how groups progress through rounds and using the results screens for debriefing.

4.4. Requirements for training centres

Because RockChain relies on cloud hosting and mobile distribution, the infrastructure requirements for training centres are minimal:

- Network: a classroom Wi-Fi network with internet access, capable of supporting simultaneous connections from the number of devices used in a session.
- Devices: Android or iOS smartphones/tablets that meet the minimum OS requirements for the deployed build (as specified in RockChain documentation).
- No local server installation: centres do not need to deploy additional servers; all game coordination, storage and authentication are handled in the cloud.

This combination, Google Cloud backend, mobile apps for Android and iOS, and access through the RockChain web area, allows the interactive RockChain Tool to be used in different countries and institutions without complex local setup, while keeping a single, centralised deployment that can be maintained and updated by the project partners.

5. MONITORING, RESILIENCE AND MAINTENANCE

A final aspect of this document is ensuring that the interactive RockChain Tool can be monitored, debugged and maintained over time, especially during pilots in real classrooms. The goal is not to build a heavy observability stack, but to provide enough visibility and robustness so that partners can understand what is happening and keep the system stable.

5.1. Observability and logging

Both the client and the server include lightweight observability mechanisms that help during development, pilot deployment and incident analysis.

On the client side, *GameContextHybridRobust* logs key technical signals to the console, such as:

- Clock drift between the local device and the authoritative server.
- Resynchronisation attempts (for example, after reconnection).
- Detection of duplicate or out-of-order events.

These logs are mainly used during development and pilot testing, when developers or technical partners run the app with debugging tools attached. They make it easier to reproduce and diagnose issues related to timing, navigation and state consistency.

On the server side, a *logMetric* utility writes structured JSON events to Cloud Logging. Typical events include:

- New connections and disconnections.
- Round starts and round ends.
- Resync operations.
- Errors or unexpected conditions.

With these logs, technical staff can:

- See how many games have been played in each period.
- Identify error patterns or performance bottlenecks (for example, repeated resyncs for certain games).
- Support debugging when partners report incidents, by correlating user reports with concrete events in the logs.

The overall approach is intentionally lightweight: there is no complex monitoring dashboard, but there is enough structured information to support basic maintenance, optimisation and troubleshooting.

5.2. Resilience and fault handling

Because RockChain is a real-time multiplayer tool used over classroom Wi-Fi, it must cope with intermittent connectivity and temporary failures without breaking the experience. Several resilience strategies are therefore built into the architecture:

- Automatic socket reconnection: The client's Socket.IO adapter attempts to reconnect automatically when the network is temporarily lost. After a successful reconnection, it re-sends the authentication token and *join_game* information so that the server can reattach the socket to the correct game and user.
- Idempotent operations on the server: Server-side operations rely on identifiers such as *roundId*, version numbers and, where needed, operation IDs. This makes it possible to recognise repeated or late messages and ignore them instead of applying them twice, avoiding duplicated rewards or inconsistent state when messages are retried.
- Round-level locks for critical phases: During sensitive phases like end-of-round calculations, the server uses round-level locks so that only one calculation flow runs at a time for each game and round. This prevents race conditions when many actions arrive near the end of the countdown.
- Graceful handling of desynchronisation: If a client has been offline, backgrounded or suffers a brief disconnection, the combination of *GameContextHybridRobust* and the authoritative server allows the device to resynchronise with the current state when it comes back. Instead of relying on whatever was rendered before, the client can refresh its view of the active round, inventories and rankings.

In practice, these mechanisms mean that short-term network issues or transient errors should not invalidate a game session. Trainers can usually continue with the activity without needing deep technical intervention, and learners who briefly lose connection can rejoin and keep playing in a consistent state.

5.3. Maintenance and future evolution

From a maintenance perspective, WP4.A5 sets out a few simple but important principles to guide future evolution of the tool:

- Keep real-time logic centralised on the authoritative server: All time-critical decisions (who mined first, when the round ends, how rewards are applied) remain on the server. The client consumes *authoritativeState* instead of trying to implement its own alternative rules. This reduces the risk of divergence between different client versions.

- Treat Cloud Functions as the single-entry point for backend operations: Any new operation that changes persistent data should be implemented as a Cloud Function, registered in functions/index.js and always called through authenticated wrappers on the client. This keeps security checks and business logic on the server and makes it easier to reason about data flows.
- Document schema changes and migrations: When Firestore document schemas evolve (for example, adding new fields to *games/{gameId}* or *users/{userId}.games[gameId]*), those changes should be documented explicitly. If existing data needs to be adjusted, migration scripts or utilities can be placed under *functions/src* so that partners have a clear path to update production data.

By following these practices, technical partners can add new features—such as extra screens, additional indicators, extended logging or new game modes—without compromising the stability of the existing game. RockChain remains maintainable and extensible beyond the initial project lifetime, while preserving the core behaviour that has been validated during pilots in WP5.

6. CONCLUSIONS

WP4.A5 has delivered a production-ready version of the interactive RockChain Tool, consolidating the design, implementation and deployment work carried out in previous tasks of WP4. The resulting system brings together a modern, multi-language mobile client, a Firebase-based backend and an authoritative real-time server on Cloud Run, all deployed on scalable cloud infrastructure and packaged into Android and iOS builds that can be installed on standard devices used in VET and adult education.

By explicitly documenting the system architecture, the runtime flows, the production and deployment workflow, and the hosting, access and maintenance procedures, this activity ensures that RockChain is not a one-off prototype but a reproducible and maintainable asset. Technical staff have a clear reference on how the tool is structured and how to update it, while trainers have a stable, installable application that can be integrated into real courses with minimal local setup.

In this way, the interactive RockChain Tool becomes the practical core of the project's e-learning offer: it operationalises the pedagogical and curricular work from previous WPs into a concrete, playable experience that can be used in classroom settings across partner countries. The tool is now ready to support pilot activities in WP5 and provides a solid basis for potential future exploitation and extension beyond the lifetime of the project.