

Chameleon Table

A Distributed Multiplayer Card Game

Final Report for the Distributed Systems Course [2025/2026]

Andrea La Rosa
`andrea.larosa2@studio.unibo.it`

July 25, 2026

Chameleon Table is a distributed multiplayer card game inspired by Coloretto, developed as a final project for the Distributed Systems course. The application allows between 2 and 5 players to join a shared game room from different devices and play in turns, with one active game per authenticated user. The system follows a client-server architecture: a central FastAPI server manages game logic and state, while clients interact through a React single-page application running in the browser. Real-time updates are pushed to all connected clients via WebSocket, ensuring that every player sees the current state of the game without polling. Game state is persisted in a PostgreSQL database, so that active games survive server restarts and players can reconnect within a grace period. The full system is containerised with Docker and starts with a single command. Key distributed systems challenges addressed include crash recovery, real-time state synchronisation across multiple clients, and safe concurrent access to shared game state.

Disclaimer

During the preparation of this work, the author used Claude (Anthropic) as support for the design and implementation of a couple of the system's mechanisms — specifically the cache-aside recovery mechanism described in section 3.7 and the two-level WebSocket ping/pong resilience mechanism described in section 3.4 — as well as for assistance with the SCSS styling, and for translating into English several sections of this report that were originally drafted in Italian.

After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the final content of both the codebase and this report.

1 Concept

Chameleon Table is a web application inspired by the card game Coloretto, accessible from any desktop browser without installation. The backend exposes a REST API and a WebSocket endpoint; the frontend is a React single-page application.

Use cases

An unregistered user can sign up to the service by providing a username, email address, and password.

A registered user can log in to the service.

An authenticated user can create a new match, choosing the maximum number of players. The match is identified by a unique room code generated by the server.

An authenticated user can join a match that has not yet started by entering the room code, becoming a player.

An authenticated user can observe an ongoing match without participating, becoming an observer. The maximum number of observers per match is four.

The user who created the match can cancel it before it starts.

A player can leave an ongoing match. If too many players leave, the match is aborted.

An observer can leave an ongoing match.

Users interact with the system through a desktop browser, in sessions lasting the duration of a match. Turns follow each other in relatively quick succession — a player is expected to act within one minute of their turn.

The use cases and the actors involved are summarised in fig. 1.

Data

The system stores two categories of data. User data (username, email, hashed password) is persisted in PostgreSQL. Game state (deck, rows, players, scores) is kept in Redis during an active match and persisted to PostgreSQL for crash recovery.

Roles

There are three roles in the system:

- **Guest** — an unauthenticated user, who can only access the login and registration page.
- **Player** — an authenticated user who creates or joins a match and participates actively.
- **Observer** — an authenticated user who watches an ongoing match without participating.

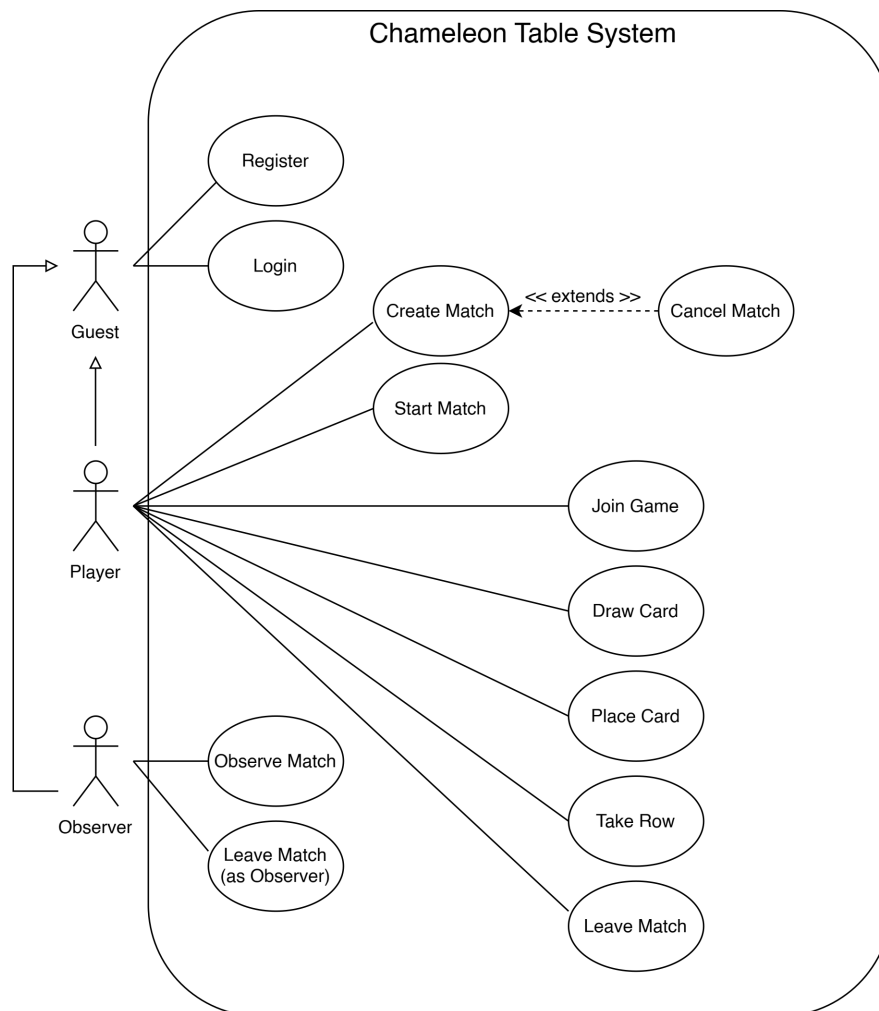


Figure 1: Use case diagram

Why is distribution needed?

Distribution is needed for several reasons. The primary one is *resource sharing*: multiple players need to interact with the same shared game state from different devices and locations simultaneously. *Fault tolerance* is also a concern: the system must survive server restarts without losing active matches, addressed through persistence on PostgreSQL and a reconnection grace period of two minutes. *Liveness* is guaranteed, with an explicit bound rather than instantaneously: an inactivity timer disconnects a player who does not act within one minute of their turn, after which a two-minute grace period runs before that player is permanently removed. A single unresponsive player can therefore freeze a match's progress for up to three minutes — and, in principle, longer still if several players become unresponsive one after another — but never indefinitely, which is the property this mechanism is actually responsible for. Finally, the architecture is designed to allow future *horizontal scaling* via Redis pub/sub, enabling multiple backend instances to share game state.

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

- F1** The system shall allow a user to register by providing a username, email address, and password.

Acceptance criterion: an unregistered user can create an account and subsequently log in.

- F2** The system shall allow a registered user to log in.

Acceptance criterion: a user with valid credentials receives an access token.

- F3** An authenticated user shall be able to create a new match, specifying a maximum number of players (between 2 and 5).

Acceptance criterion: a match is created with a unique room code, and the creator is listed as the first player.

- F4** An authenticated user shall be able to join a match that has not yet started by entering the room code.

Acceptance criterion: the player appears in the match player list, and all other players already in the match receive the updated state.

- F5** The player who created the match shall be able to start it once at least two players have joined, without necessarily waiting for the maximum chosen at creation.

Acceptance criterion: the match transitions to the PLAYING phase and all players in the match receive the updated state.

- F6** The player who created the match shall be able to cancel it before it starts.

Acceptance criterion: the match transitions to the ABORTED phase and all waiting participants are notified.

- F7** During their turn, a player shall be able to draw a card from the deck.

Acceptance criterion: the card is removed from the deck and assigned as the pending card.

- F8** After drawing a card, a player shall be able to place it on an available row.

Acceptance criterion: the card appears on the chosen row and the turn passes to the next player.

- F9** A player shall be able to take all the cards assigned to a row.

Acceptance criterion: the cards of the row are assigned to the player and the row is marked as taken.

- F10** A player shall be able to leave an ongoing match.

*Acceptance criterion: the player is marked as permanently left — distinct from the temporary **inactive** status used for the timeout/grace-period path (F14, F15) — so that a voluntary departure is counted correctly towards the threshold in F11.*

- F11** A match shall be automatically aborted if fewer than two players remain able to play — that is, have not permanently left the match. A player merely inactive within their grace period (F14) still counts towards this threshold, since the match continues, with their turn frozen, while that grace period is running.

Acceptance criterion: the match transitions to the ABORTED phase and all connected clients are notified once the count of non-left players drops below two.

- F12** An authenticated user shall be able to observe an ongoing match without participating in it.

Acceptance criterion: the observer receives real-time state updates without being able to perform any game action.

- F13** An observer shall be able to leave the match they are observing.

Acceptance criterion: the observer is removed from the observer list and no longer receives state updates.

- F14** The system shall allow a disconnected player to reconnect within a two-minute grace period.

Acceptance criterion: a player who reconnects within the grace period resumes playing normally.

- F15** The system shall detect connected but inactive players and automatically disconnect them after one minute of inactivity on their turn.

Acceptance criterion: a player who does not act within one minute is disconnected and the grace period timer starts; the turn remains frozen on that player for the duration of the grace period rather than passing to another player, preventing a disconnected player from being silently skipped.

F16 Active matches shall survive a server restart.

Acceptance criterion: after a server restart, active matches are restored and players can reconnect.

F17 An authenticated user shall be able to view the list of available matches, including those waiting for players and those currently in progress.

Acceptance criterion: the user can see waiting matches to join and ongoing matches to observe.

F18 At the end of a match, all players shall be able to view the final scores.

Acceptance criterion: once the match transitions to the FINISHED phase, the scores of all players are displayed.

F19 The number of observers per match shall not exceed four.

Acceptance criterion: a fifth user attempting to observe a match receives an error and is not added to the observer list.

F20 An authenticated user shall be able to participate in at most one match at a time.

Acceptance criterion: a user who is already a player in a WAITING or PLAYING match receives an error when attempting to create or join another one.

2.2 Non-functional Requirements

NF1 Game state updates for active matches shall be transmitted in real time to all connected clients (polling excluded).

Acceptance criterion: all connected clients receive the update within approximately 500ms of the action that generated it, on a local-area or comparable low-latency connection; this figure is an indicative target for manual acceptance testing, not a value enforced or measured by an automated test.

NF2 Connected clients shall never silently regress to an older game state once a newer one has been observed, except in the single documented case where the server's own authoritative state has itself gone backwards; in that case, deliberately resynchronising to the server's rolled-back state is preferred to a client remaining frozen on state the server no longer considers current. Clients shall in all cases eventually converge to the same state once message delivery resumes.

Acceptance criterion: for any two updates A and B with B's sequence number greater than A's under normal operation, no client that has already applied B subsequently applies A silently; a client that misses an update converges to the current state

within one heartbeat cycle (section 3.4) rather than remaining indefinitely stale. The sole exception is the stale-persisted-snapshot recovery described in section 3.7: there, a sequence number that decreases on an open connection is recognised as exactly that scenario and triggers an explicit resynchronisation, rather than being either silently applied or silently discarded. This is a monotonic-reads, eventually-consistent guarantee under normal operation, not simultaneous observation across clients — true simultaneity is not a meaningful guarantee in a distributed system, since broadcasts reach different clients with different latencies.

- NF3** The system shall guarantee match liveness; player inactivity shall not cause a match to stall.

Acceptance criterion: no match remains blocked indefinitely due to an inactive or disconnected player.

- NF4** The system shall ensure that only authenticated users can access game features.

Acceptance criterion: requests without a valid token are rejected by the system (error 401).

- NF5** The system shall ensure that each user can only perform actions allowed by their role.

Acceptance criterion: an observer cannot perform game actions; a player cannot act outside their turn.

2.3 Implementation Requirements

- I1** Data persistence shall be handled by PostgreSQL.

Motivation: the need for a robust relational database for crash recovery and predisposition to future failover architectures.

- I2** The in-memory game state shall be managed via Redis.

Motivation: the need for low latency in the most frequent operations and predisposition to future horizontal scaling via pub/sub.

- I3** The entire system shall be containerised with Docker and startable with a single command.

Motivation: guarantees portability and reproducibility of the environment.

- I4** Real-time communication shall take place via WebSocket.

Motivation: necessary to guarantee NF1.

- I5** The system shall expose a REST API for room management and authentication.

Motivation: separates synchronous request-response actions (authentication, room management) from the push channel used for state updates, avoiding overloading the WebSocket connection with both responsibilities.

2.4 Relevant Distributed System Features

Relevant

Fault tolerance The system must survive server crashes, client disconnections, and player inactivity. Data loss of an active match is unacceptable. The system addresses this through crash recovery on PostgreSQL, a two-minute reconnection grace period, and an inactivity timer that automatically disconnects inactive players.

Resource sharing Multiple clients concurrently access the same shared game state. Concurrent access is protected by a per-room lock (`asyncio.Lock`) that serialises operations on the state, guaranteeing consistency.

Security The system manages user data and requires JWT authentication to access game features. Authorisation is role-based: during a match, an observer cannot perform game actions and a player can only act during their own turn.

Performance and concurrency The server manages multiple matches simultaneously. Update latency must be low to guarantee a good gaming experience; for this reason WebSocket is used instead of polling, and Redis serves as the fast-access layer for game state.

Partially relevant

Transparency The client is unaware of the internal details of the system (Redis, PostgreSQL, locks). It only sees the game state updated via WebSocket.

Scalability The system is single-instance, but the architecture is designed for future horizontal scaling via Redis pub/sub.

Evolvability The architecture is designed to be extended: not only for scaling via Redis pub/sub, but also for failover via PostgreSQL and future features via JWT.

Not addressed

Openness and interoperability Although standard protocols are used, interoperability with external systems has not been considered.

Economy No cost evaluation has been performed.

3 Design

This chapter explains the strategies used to meet the requirements identified in the analysis.

3.1 Architecture

The client-server architecture is a natural choice for a digitalised board game. Established platforms in the sector, such as Boardgame Arena and Chess.com, adopt the same approach for reasons that apply to Chameleon Table as well. The authoritative-server principle itself is discussed in the wider game networking literature primarily in the context of fast-paced, real-time games [Gambetta, 2013], where an authoritative server is motivated by cheating prevention despite the added latency it introduces; Gambetta explicitly notes that the simpler “dumb client” scheme — where the client sends no more than raw actions and renders whatever state the server returns, with no local prediction — already works well for *slow, turn-based games*, which describes Chameleon Table exactly: without the tight latency budget of an action game, there is no motivation here to trade any of the server’s authority for responsiveness. This distinction between action games and slower, turn-based ones is not merely anecdotal: an empirical study of latency tolerance across game genres [Claypool and Claypool, 2006] found that turn-based and strategy games tolerate network delay far better than fast-paced action genres, where delays of even a few tens of milliseconds measurably affect player performance — a further, evidence-based reason why Chameleon Table’s design accepts full server authority with no client-side prediction, unlike the real-time games for which that trade-off is usually debated.

The sole authority over the game state is the centralised server, in order to guarantee information hiding. The deck, the order of the cards, and information private to individual players are never transmitted to clients, preventing a dishonest client from accessing data outside its scope or altering the game state in its favour. The server is also the only entity authorised to determine who can act and when, guaranteeing the correctness of the turn order.

Crash recovery is simplified by centralisation. Since a match can last several minutes and many matches may run simultaneously, client-side crashes are frequent and inevitable; a peer-to-peer architecture would require strong coordination among participants. With a centralised server it is sufficient to maintain an up-to-date snapshot for each match: if the server crashes, the match is restored on restart; if a client crashes, the player can simply reconnect within the grace period.

Anomalous system states — disconnections, prolonged inactivity, abandonments — are handled centrally without burdening individual clients, which in a peer-to-peer model would be responsible for this logic.

The main drawback of this architecture is the single point of failure: a server crash interrupts all ongoing matches. This risk is mitigated through crash recovery on PostgreSQL, but a multi-instance architecture with automatic failover would offer stronger guarantees.

3.2 Infrastructure

The system infrastructure is composed of five components, all running as Docker containers orchestrated by Docker Compose, as shown in fig. 2.

Nginx is the sole entry point for all incoming traffic. It acts as the system's reverse proxy, routing requests to the appropriate internal service based on the URL path: `/` to the frontend, `/api/` to the backend REST API, and `/ws/` to the backend WebSocket endpoint. Rate limiting is also enforced to protect the backend from overload.

Frontend is a React single-page application composed of static files, not directly reachable from outside the Docker network.

Backend is the FastAPI application server; it manages game logic, state, authentication, and WebSocket connections. It is not directly reachable from outside the Docker network.

PostgreSQL is the relational database used for persistent storage of user data and game state snapshots. It is only reachable internally.

Redis serves as an in-memory data store used as the fast-access layer for active game state. It is only reachable internally.

All components reside on the same host and communicate over a shared internal Docker network. Components discover each other via Docker Compose's built-in DNS resolution: each service is reachable by its own name (`backend`, `db`, `redis`, etc.). Only the Nginx container exposes a port to the outside world (port 80 in development, ports 80 and 443 in production).

In the development environment, all containers run on a single Windows machine with Docker Desktop, and the browser runs on the same machine. In the production environment, all containers run on a single virtual machine on Google Cloud Platform; the browser runs on the users' devices, which are geographically distributed. A Dynu dynamic DNS domain name points to the public IP address of the VM, and Let's Encrypt provides the TLS certificate for HTTPS.

3.3 Modelling

Domain entities

The main domain entities are:

User A registered user, identified by username, email and password. They can take the role of Player or Observer during a match.

Match (GameState) A match, identified by a unique room code. It is the central entity of the system and contains all the state necessary for the match to proceed.

Player A user who actively participates in a match. They have a hand of cards, a set of jokers, and a status that can be active, inactive, or permanently left.

Observer A user who watches an ongoing match without participating. They have no cards and no turns.

Card A game card, characterised by a type (`color`, `joker`, `plus2`, `last_round`) and optionally by one of seven available colours.

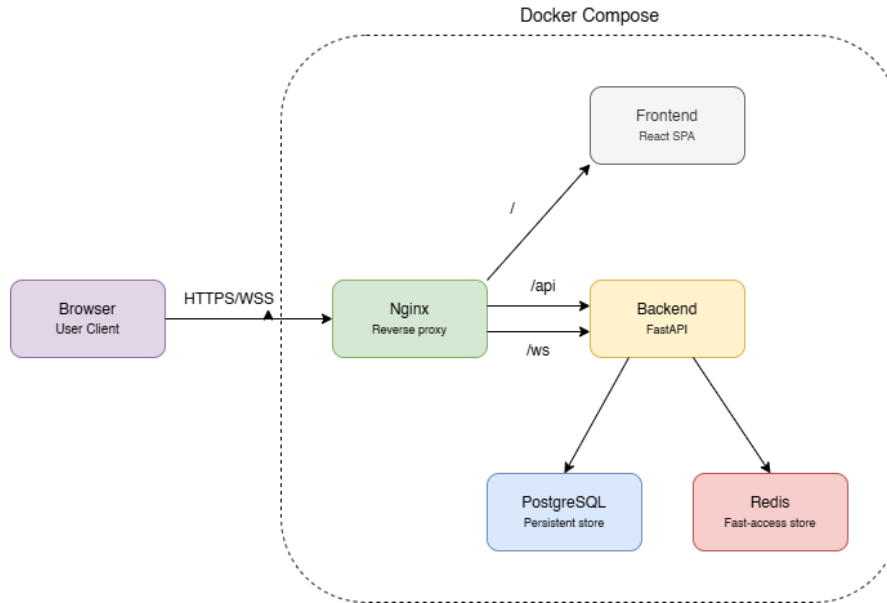


Figure 2: Infrastructure component diagram

Row One of the rows on the game board. It contains the cards placed during the turn and can be taken by a player.

Mapping to infrastructural components

The complete state of each match resides exclusively on the server. During an active match, the state is kept in Redis to guarantee low latency; a copy is persisted to PostgreSQL on every significant change for crash recovery. Clients maintain no authoritative state: they receive full state snapshots from the server via WebSocket and only display them. This guarantees consistency — there is a single source of truth. Information hiding is achieved by providing each client only with the information appropriate to its role.

User data (username, email, and hashed password) is persisted only in PostgreSQL and never passes through Redis.

Domain events

The main events that modify the system state are:

- Match: created, started, aborted, finished
- Player: joined, left, disconnected, reconnected
- Card: drawn, placed
- Row taken

- Round ended
- Observer: joined, left

Messages exchanged

Messages fall into two categories. *REST commands* are sent from the client to the server to request an action and receive a synchronous response with the outcome of the operation and, where the operation concerns an ongoing match, the updated state. *WebSocket events* are sent by the server to all clients connected to a room whenever the state changes. Each message contains the full snapshot of the match state — not just the delta — to simplify client-side handling and ensure that a reconnecting client immediately receives a consistent state. Each message includes a monotonically increasing sequence number. On a single, already-established WebSocket connection, TCP already guarantees in-order, non-duplicated delivery, so the sequence number is not there to compensate for the transport layer. Its actual purpose is threefold: to let a client detect that it has fallen behind after a race between a REST action’s synchronous response and the broadcast it triggers; to let a reconnecting client tell a genuinely new state apart from one it has already seen, since a fresh connection has no ordering guarantee relative to what came before it closed; and to drive the resynchronisation mechanisms described in section 3.4 — both the heartbeat-based recovery from a missed broadcast, and the recovery from a sequence number that unexpectedly *decreases* on an open connection, which signals that the server itself has recovered from a stale persisted snapshot (section 3.7) rather than that a message arrived out of order.

System state

The global state comprises the set of registered users and the set of active matches. For each match, the state includes:

- the remaining deck of cards
- the rows on the board
- the list of players with their respective cards and jokers
- the current turn
- the match phase (WAITING, PLAYING, FINISHED, ABORTED)
- the list of observers
- the card currently awaiting placement (`pending_card`)
- the current sequence number

The domain model is shown in fig. 3.

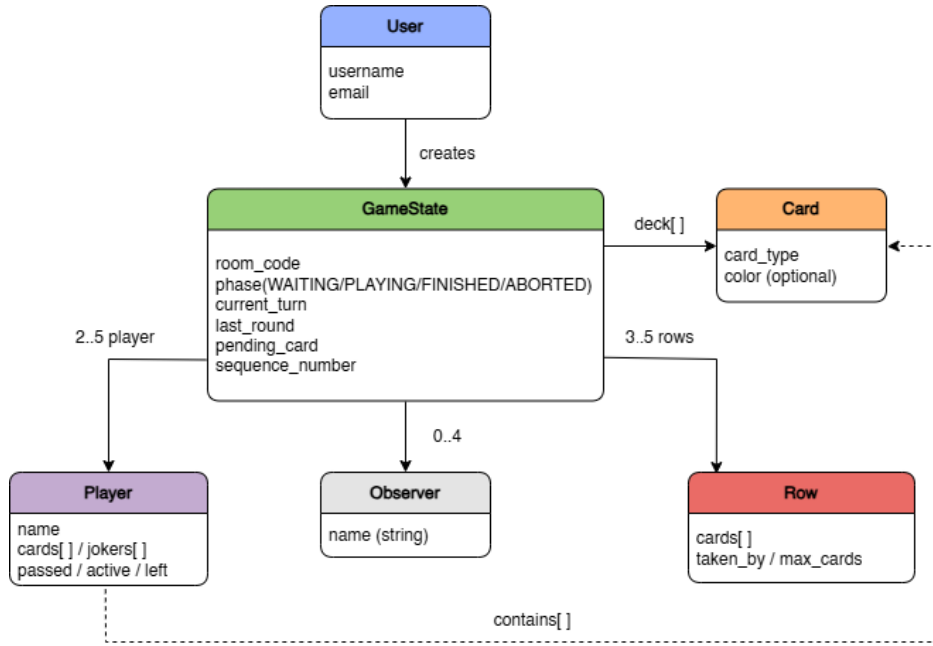


Figure 3: Domain model diagram

3.4 Interaction

Interaction patterns

The system uses two distinct interaction patterns, each suited to a different type of communication. Registration, login, room creation and joining, match start, leave, and all in-match player actions (drawing a card, placing it, taking a row) use the synchronous *request-response* pattern via REST. The client sends an HTTP request to the server through Nginx and receives a synchronous response containing the outcome of the operation and the updated state. These operations modify the system state and require JWT authentication in the request header.

WebSocket is used almost exclusively as a server-to-client push channel: the client never issues a *game* command over it (the only message a client ever sends on this channel is the lightweight heartbeat ping described later in this section, which carries no game semantics). Real-time game operations follow the asynchronous *push* pattern described below. Each client connected to a room maintains a persistent WebSocket connection with the server. Whenever the state changes — following a REST action — the server broadcasts the new state to all clients connected to the room. The client does not request updates; it receives them automatically. This guarantees that all participants stay up to date without polling, converging on the same state as broadcasts arrive — not that every client observes the exact same state at the exact same instant, which is not a meaningful guarantee to make in a distributed system (see NF2 in section 2 and the discussion in section 3.8).

Concurrency

Multiple clients may send requests to the same room simultaneously. To guarantee state consistency, each room is protected by an exclusive lock (`asyncio.Lock`). The backend acquires the corresponding lock before every read or write on a match's state. This serialises operations on the same room, preventing race conditions. Locks on different rooms do not block each other, guaranteeing parallelism across distinct matches.

Messages and consistency

Every WebSocket message includes a monotonically increasing sequence number, for the reasons already discussed earlier in this section: not to reorder messages within a single TCP connection, which is unnecessary, but to let a client detect that it has fallen behind (a missed broadcast, a race between a REST response and the broadcast it triggers, or a reconnection) and resynchronise. The client tracks the last applied sequence number; a message with the same number is a duplicate and is discarded, while a message with a *lower* number is treated not as something to silently discard but as a signal that the server's own state has gone backwards — which can only happen through the stale-snapshot recovery path described in section 3.7 — and triggers a resynchronising REST call rather than being dropped.

All messages contain the full snapshot of the match state. Compared to delta-based updates, this approach simplifies client-side handling and eliminates the need to maintain an incremental state: a reconnecting client immediately receives a consistent state without having to reconstruct the history of previous events.

WebSocket — connection and reconnection

On connection, the backend verifies the JWT token passed as a URL parameter. If the token is valid and the room exists, the client is added to the connected list and immediately receives a full snapshot of the current match state.

In the event of an involuntary disconnection, the backend sets the player as inactive and starts a two-minute grace period timer. During this period, the other clients receive a broadcast notifying them of the player's inactivity. If the player reconnects within the grace period, the timer is cancelled, the player is reactivated, and all clients receive a new broadcast. If the timer expires without reconnection, the player is considered to have left permanently. Under specific conditions, the match is aborted.

Zombie connection detection

The reconnection mechanism described above relies on the client's `onclose` event firing when the connection drops. WebSocket itself, as specified in RFC 6455 [Fette and Melnikov, 2011], is built on top of a single long-lived TCP connection, and its close handshake assumes a well-behaved network that actually delivers a closing frame or lets the underlying TCP connection reset cleanly. On some networks — most notably mobile carrier NAT/hotspot configurations — a connection can become a *zombie*: the underlying

network silently drops packets in one or both directions without either endpoint ever receiving a proper TCP close signal, so the browser’s WebSocket still reports `readyState = OPEN` even though the connection is, in practice, dead or has silently lost a specific message. This is precisely the gap that motivates an application-level heartbeat on top of the protocol: RFC 6455 itself defines ping/pong control frames for exactly this purpose, since TCP alone cannot be relied on to surface every real-world connectivity failure in a timely manner.

A naive ping/pong heartbeat that merely asks “does this connection respond right now?” does not cover this scenario: a single dropped broadcast can leave the client permanently out of sync while a ping sent moments later would still receive a normal pong, since the connection itself has not actually failed. The mechanism therefore operates on two levels:

Level 1 — dead connection The client sends a lightweight `{"type": "ping"}` message every 15 seconds. If no pong is received within 6 seconds, the client force-closes the WebSocket itself, which triggers the existing reconnection logic described above. This interval is deliberately more aggressive than the 25–30 second interval typically recommended to keep NAT mappings alive, since the one-minute per-turn inactivity timeout (F15) requires detecting a dead connection well before a player risks being marked inactive for a network problem that is not their fault.

Level 2 — stale but alive Every pong response carries the room’s current sequence number, read directly from the server’s state at the moment the pong is sent. If this number differs at all from the one the client has already applied — whether the server is ahead, meaning the client missed at least one broadcast even though its connection never actually closed, or behind, in the stale-snapshot case discussed in section 3.7 — the client’s local number can no longer be trusted in either direction. Rather than touching the WebSocket, it issues a single `GET /rooms/{room_code}/state` REST call — the same endpoint already used elsewhere — to resynchronise, then resumes normal operation over the existing connection.

The two levels are complementary in the same sense as the two fault-tolerance mechanisms in section 3.7: level 1 answers “is this connection even alive?”, while level 2 answers “the connection is alive, but did I miss something?”. Neither level requires any change to the server’s broadcast logic; the pong handler only reads the current sequence number from Redis, without acquiring the per-room lock, since it is a pure read with no correctness requirement stronger than eventual consistency over the next ping cycle.

Nginx

Nginx is not a passive proxy. For REST requests, it enforces rate limiting to protect the backend from overload. For WebSocket connections, it handles the `HTTP → WebSocket` protocol upgrade via the `Upgrade` and `Connection` headers, and keeps the connection open without timeout for the entire duration of the match. The client always

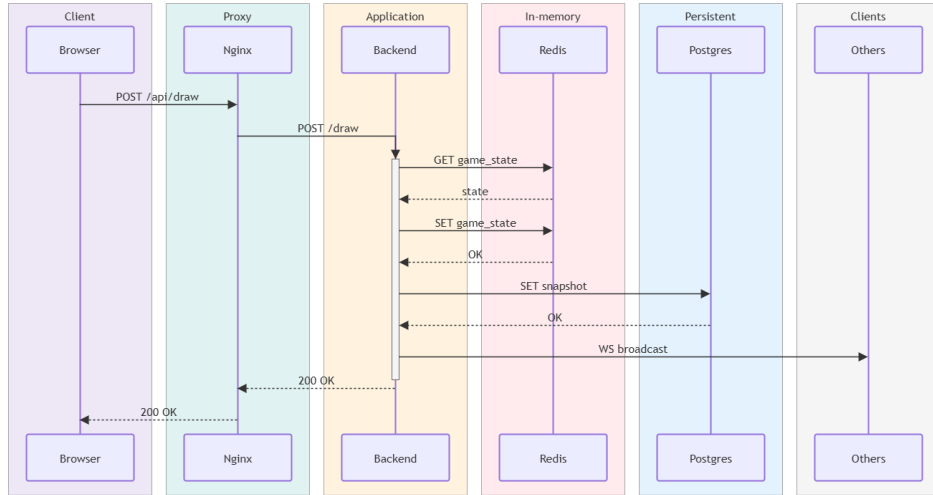


Figure 4: Sequence diagram: draw card flow

communicates exclusively with Nginx and is unaware of the existence of the backend, Redis, or PostgreSQL.

The sequence diagrams in fig. 4 and fig. 5 illustrate respectively the flow of a game move and the WebSocket connection/reconnection flow.

3.5 Behaviour

Stateful components

Redis holds the in-memory state of all active matches. It is the component that the backend reads and updates on every game operation. The state is volatile — in the event of a Redis crash the data would be lost — but this is prevented by PostgreSQL.

PostgreSQL is stateful. It holds the persistent storage of user data and match snapshots. It survives server crashes and enables crash recovery on restart.

The backend is the central and logically stateful component. It is the sole entity responsible for updating the system state. It handles REST requests from clients, updates the state in Redis, persists to PostgreSQL, and notifies all clients via WebSocket broadcast. It also manages asynchronous timers for player disconnection and inactivity.

Stateless components

Nginx is completely stateless. It only routes requests to the appropriate service based on the URL path and handles the WebSocket upgrade. It maintains no information about the state of matches or users.

The frontend is stateless from an authoritative point of view. It locally maintains a copy of the match state received via WebSocket, but this copy is purely for display purposes. It makes no game decisions and does not modify the system state autonomously.

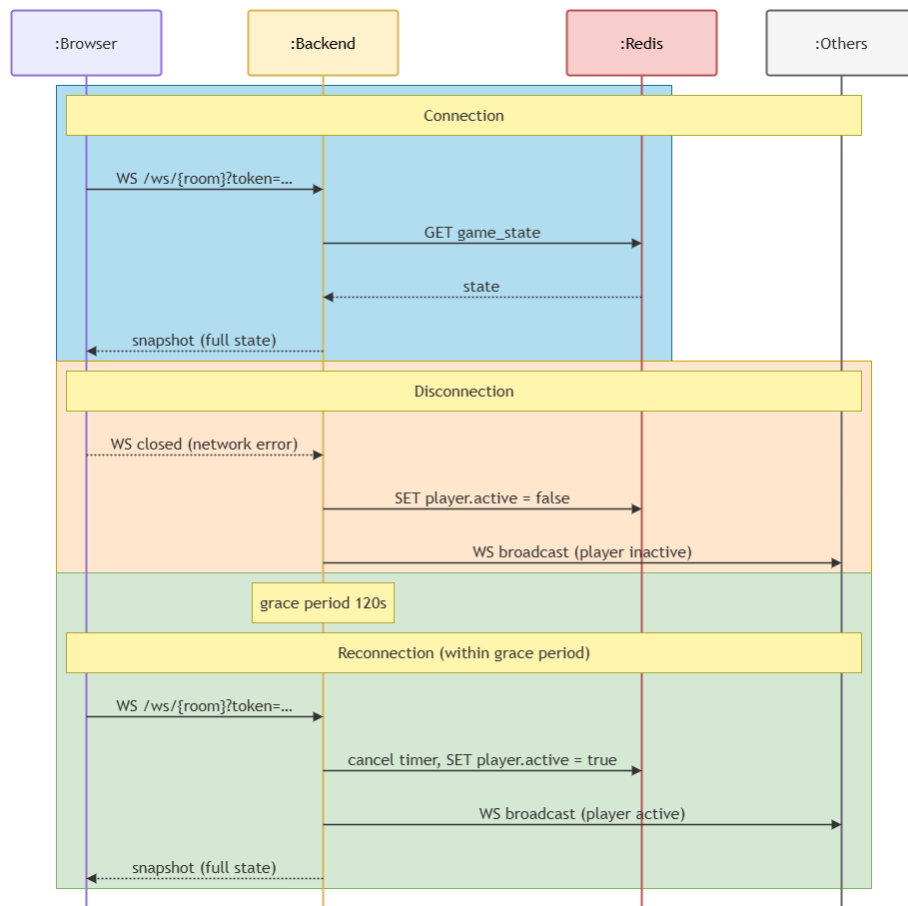


Figure 5: Sequence diagram: WebSocket connection and reconnection

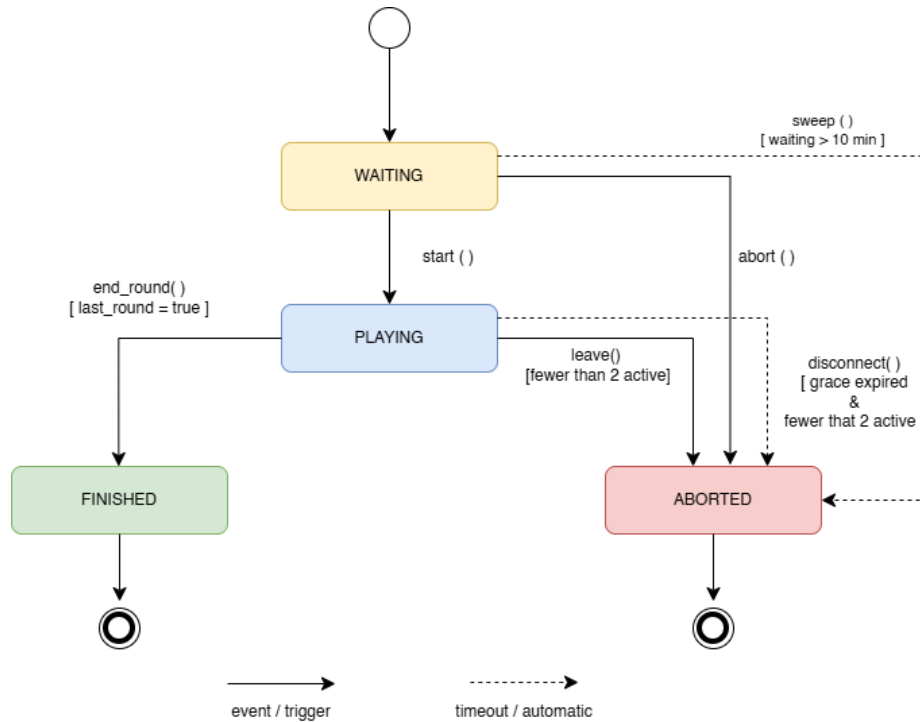


Figure 6: Match lifecycle state diagram

Match lifecycle

The match itself is behaviourally interesting; it is modelled as a finite state machine with four states:

WAITING The match has been created and players are joining. In this state players can be added, and the creator can start or cancel the match.

PLAYING The match is in progress. Players take turns. The state transitions to **FINISHED** after the last round is completed following the drawing of the `last_round` card. It transitions to **ABORTED** if too many players leave or disconnect permanently.

FINISHED The match has ended normally. The final scores are available.

ABORTED The match has been interrupted, either before it started due to cancellation by the creator, or during the match due to too many players leaving.

The state diagram in fig. 6 illustrates the transitions between states with their guards and triggers.

3.6 Data and Consistency Issues

Data

The system manages two categories of data with different characteristics and requirements.

User data — username, email, and hashed password — is persisted exclusively in PostgreSQL in relational form. It is written once at registration and read at every login. It never passes through Redis.

Match state is the most critical and frequently updated data. It is kept in Redis during the match to guarantee low latency, and persisted to PostgreSQL on every significant change (match start, card drawn, row taken, round ended, player left) for crash recovery. PostgreSQL stores the snapshots in serialised form.

Consistency

The main risk of inconsistency is concurrent access to a room's state from multiple simultaneous REST requests. This is resolved through an exclusive per-room lock (`asyncio.Lock`). Every operation that reads or modifies the state acquires the lock before proceeding and releases it upon completion. This guarantees that operations on the same room are serialised.

The WebSocket broadcast always occurs within the lock, after the state has been updated in Redis and, per section 3.7, after the PostgreSQL snapshot has been attempted. This guarantees that no two clients ever receive broadcasts for the same room in an order different from the order in which the corresponding actions were actually applied, and that the PostgreSQL snapshot for a room is never persisted out of order relative to another concurrent action on that same room, since both the Redis write, the PostgreSQL write, and the broadcast for a given action all complete while the same lock is held before the next action on that room can begin.

This ordering guarantee has a concrete cost that is worth stating rather than glossing over: the broadcast to each connected client is awaited sequentially, one at a time, while the lock is held. A single slow or unresponsive client (for instance, on a degraded mobile connection) can therefore delay the broadcast — and, transitively, the next action on that room — for every other participant, a textbook instance of head-of-line blocking. With at most five players and four observers per room this is a bounded, currently acceptable cost, but it would need revisiting (for instance, by gathering the per-client sends concurrently, or by moving the network I/O itself outside the lock and relying on the sequence-number mechanism above to keep clients correctly ordered even if delivery is not) before the per-room player count, or the tolerance for slow clients, could grow substantially.

Holding the lock slightly longer than the minimum needed for the state update is a deliberate trade-off: it guarantees that every connected client observes a state that exactly matches the committed room state, at the cost of a marginally longer lock-holding time per action.

Shared data between components

The backend is the only component that reads from and writes to Redis. PostgreSQL is used only for persistence and recovery, not for real-time operations. There is no direct state sharing between multiple backend instances, as the system is single-instance.

The architecture is designed for future horizontal scaling: by introducing Redis pub/sub, multiple backend instances could share match state, replacing the in-process lock with a distributed Redis lock.

3.7 Fault-Tolerance

Data replication and persistence

Match state is managed on two levels to balance performance and safety.

In-memory (Redis) Used during gameplay to minimise latency.

Persistent (PostgreSQL) On every relevant event or state change, the state is saved to the database as a snapshot.

The synchronisation between the two systems is not transactional: each write to PostgreSQL is still awaited synchronously by the request that triggers it (so the response only returns once it has actually completed, or failed), but there is no atomic, all-or-nothing guarantee spanning both stores — there is no rollback on Redis if the PostgreSQL write fails, or vice versa. This is therefore not intended to guarantee high availability in real time. PostgreSQL acts as a safety anchor for crash recovery: in the event of a server crash, the system restarts from here.

The order of the two writes is deliberate and matters: on every mutating endpoint, Redis is written first (`set_game`), and the PostgreSQL snapshot is written second, wrapped in a `try/except` that only logs a warning on failure rather than raising (section 3.7, “Database resilience” below). One consequence worth stating explicitly: the HTTP 200 response returned to the client reflects that the action was applied in Redis, not that it was durably persisted to PostgreSQL — if the PostgreSQL write fails silently, the client has no way to know that this particular action would not survive a subsequent Redis-only crash. This is an accepted trade-off in favour of gameplay continuity over always communicating durability status to the client, consistent with the system’s general preference (section 3.8) for not interrupting an active match over a persistence-layer hiccup.

A second, more subtle consequence follows from that same ordering: if Redis’s write succeeds but the following PostgreSQL write fails, and Redis is then the one that crashes shortly afterwards, the cache-aside recovery described next would restore a PostgreSQL snapshot whose sequence number is *older* than what connected clients had already seen and applied. Naively, a client that only ever discards messages with a sequence number less than or equal to its own would then discard every subsequent broadcast as a stale duplicate, appearing frozen even though the match itself has recovered. The client-side handling therefore treats a sequence number that *decreases* on an already-open connection

differently from an ordinary duplicate: since sequence numbers cannot legitimately move backwards during normal operation, a decrease is treated as a signal that the server’s own state has been rolled back, and triggers a resynchronising REST call (the same mechanism used for the heartbeat-driven recovery in section 3.4) rather than being silently dropped.

Server crash recovery

On startup, the backend queries PostgreSQL to retrieve all matches that were in the `PLAYING` state at the time of the crash. To restore the session without penalising any player:

1. All players in the match are temporarily set to inactive.
2. A two-minute grace period timer is started.
3. If clients reconnect within the timeout, they resume the match from where they left off; otherwise, they are considered permanently disconnected.

This mechanism is *eager* and *global*: it runs once, at backend startup, and restores every match that was active at the time of the crash in a single pass.

Cache-aside recovery from a Redis-only failure

A terminological note before describing this mechanism: it is called *cache-aside* because it follows that pattern’s shape (repair the cache from the database on a miss), but the roles are inverted relative to the pattern’s textbook form. In a traditional cache-aside setup, the database is the authoritative source of truth and the cache is a disposable, always-reconstructible read optimisation. Here it is the opposite, as discussed later in section 3.8: Redis is authoritative during an active match, and the PostgreSQL snapshot can lag behind it by however much time has passed since the last successful persist. The mechanism below is best read as “PostgreSQL is invoked with cache-aside *semantics* to repair Redis”, not as a claim that PostgreSQL is the system’s primary source of truth.

The mechanism above assumes that the backend process itself has restarted. A different, and in some ways more likely, failure mode on a resource-constrained single-instance deployment is that *only* Redis crashes and loses its data while the backend process keeps running — for instance because Redis is killed by the operating system’s out-of-memory manager while the Python process survives. In this case `lifespan()` never runs again, so the eager recovery above never triggers, yet the affected matches would otherwise appear to have vanished.

To cover this complementary scenario, `redis_store.get_game` and `redis_store.game_exists` implement a *lazy, per-room* cache-aside recovery: whenever Redis reports that a key for a given room is missing, the backend falls back to the last snapshot persisted in PostgreSQL, transparently repopulates Redis with it, and lets the original request proceed as if the cache had never missed. The repair happens exactly once per affected room, triggered by whichever client action or state query happens to touch that room first after the data

loss; rooms that receive no traffic in the meantime simply remain repaired the next time someone does interact with them.

The two mechanisms are complementary rather than redundant: the eager, global recovery in `lifespan()` answers the question “the backend process itself restarted, what do I do with everything that was running?”, while the lazy, per-room cache-aside recovery answers “Redis alone lost some data, but I am still running — what do I do the next time someone asks for that specific room?”. Designing for both was a deliberate acknowledgement that, in a distributed system, components are expected to fail *independently* of one another, not necessarily in lockstep. The two mechanisms cover mutually exclusive failure modes: exactly one of them, never both, is the one that actually runs for a given incident.

Recovery from a silent Redis restart

The lazy mechanism just described repairs Redis when a room’s key is *missing*. There is, however, a second Redis-only failure scenario that this mechanism does not cover: Redis, owing to RDB persistence on a Docker volume, can restart on its own — for instance following an OOM-kill and an automatic container restart — without the backend process ever noticing. In this case Redis does not lose the key: it reloads it from the on-disk RDB snapshot, which can be minutes old relative to the last state actually played. Since `get_game` only consults Postgres when Redis reports the room’s key as missing, a snapshot that is present but stale triggers no repair at all: it silently masks Postgres, which may hold a more recent state.

To cover this scenario, `redis_recovery_watcher` was introduced: a background task, started from `lifespan()`, that PINGs Redis every five seconds by default. On the down-to-up transition — that is, the moment a PING succeeds after the previous one had failed — the watcher calls `reload_playing_games_from_postgres()`, which unconditionally overwrites, for every match in the `PLAYING` phase, whatever state Redis currently holds with the last snapshot persisted in Postgres. Matches in other phases (`WAITING`, `FINISHED`, `ABORTED`) are left untouched, since they have no evolving game state to protect. This is deliberately the inverse of the logic in `get_game/game_exists`: there, Postgres is consulted only in the *absence* of the Redis key; here, the overwrite happens unconditionally on the recovery edge, precisely because the key *can* be present yet unreliable.

This overwrite can produce exactly a sequence-number regression on the client side — the same pattern already documented for the recovery `lifespan()` performs after a backend restart (cf. NF2 and the discussion in section 3.4), here simply triggered by a different event (Redis going down, rather than the backend). It is handled by the same, single client-side resynchronisation mechanism already in place: no additional ad-hoc logic was needed.

The mechanism was verified manually by stopping and restarting the Redis container via Docker Compose and observing the logs, as well as with a counter-tested automated test confirming that `reload_playing_games_from_postgres` correctly overwrites a stale state.

Involuntary client disconnection

If a client loses its connection (detected via WebSocket closure), a two-minute grace period timer is started to allow reconnection. In the meantime, the match continues normally for the other participants — with one exception: if it is the disconnected player's turn, the turn is deliberately *frozen* on them rather than passed to the next player. Both `current_turn` and the turn-advancement logic treat a player as still holding the turn as long as they have not been marked permanently `left`, regardless of their `active` status; only once the grace period expires without reconnection does the turn genuinely move on. This was a deliberate design choice over the alternative of skipping the disconnected player's turn immediately: the latter would let other players act while the disconnected player's cards and pending decisions are still in limbo, which does not match the intended semantics of a grace period.

If the player reconnects within the grace period, the timer is stopped and the player becomes active again. If the timer expires, the player is removed from the match. If their absence makes it impossible to continue, the match is aborted.

A subtlety surfaced during testing: if the disconnected player was the last one still owing a move in the current round — i.e. every other active player had already taken a row — expiring their grace period and marking them `left` could leave the round with no player left to act, without the end-of-round check ever firing, since that check previously only ran as a side effect of a `take_row` request. The fix was to re-run the same end-of-round check directly at the point where the grace period expires, rather than relying exclusively on a subsequent player action to trigger it.

Inactivity handling (anti-stall)

A player may stay connected yet stop acting on their turn, which the disconnection mechanism above cannot detect: the WebSocket never closes, so nothing fires. An inactivity timer therefore runs for the whole duration of each turn: if the current player does not act within one minute, they are marked inactive and the two-minute grace period begins, exactly as if they had disconnected.

The complementary case — a player who is marked inactive and then acts again via REST — is handled by reactivating them on their next accepted action, since a REST request is itself proof of liveness. Simply flipping `active` back to `true`, however, is not enough. A `handle_disconnection` task already scheduled from the original timeout, however, does require explicit handling — a subtlety that only surfaced while writing the regression test for this fix. Relying on the woken-up task simply finding the player `active` again and no-oping is not sufficient: if the player is marked inactive a second time before the *first* task's deadline, that orphaned task wakes up at its original deadline, observes `active = False` — set by the newer cycle, not its own — and expels the player before the current grace period has actually expired. The reactivation therefore also cancels the pending expulsion task outright; the `active` check inside `handle_disconnection` remains in place as a second line of defence rather than the only one.

Database resilience (PostgreSQL down)

All queries to PostgreSQL are wrapped in `try/except` blocks. If the database goes offline, the system does not crash: matches continue to run using Redis only. In this degraded scenario, crash recovery is not guaranteed, and the anomaly is immediately logged at warning level on the server.

Redis resilience in background timers

A related gap was identified in the background timers themselves (`handle_inactivity`, `handle_disconnection`): unlike REST endpoints, these have no caller to report a Redis connection failure to, so an unhandled exception at the exact moment a timer wakes up — not during the wait itself, only at that specific instant — would silently kill the task rather than degrade gracefully. If this happened to `handle_disconnection` specifically, the affected player would never be marked `left`, potentially freezing the match indefinitely, since nothing else re-triggers that logic on its own. For this reason, `room_exists_with_retry` — the function used by both timers to verify, upon waking up, that the room still exists before proceeding — retries indefinitely on failure, rather than giving up after a fixed number of attempts: if Redis remains unreachable for longer than expected at that exact moment, the timer simply keeps waiting for it to become available again, instead of silently abandoning its task. This was identified late in development and was not exercised by a dedicated test, given the difficulty of reliably reproducing the exact timing coincidence required.

Abandoned waiting rooms

The mechanisms described so far all protect matches once they are **PLAYING**: the per-turn inactivity timer (section 2, F15) guarantees that an active match can never stall indefinitely, since it forces a turn forward — and, if enough players stop responding, the match towards **ABORTED** — regardless of how few players remain, as long as at least one of them was expected to act.

A room in the **WAITING** phase has no such protection, since it has no current turn to time out: if the host creates a room and simply navigates away without starting or cancelling it, nothing in the per-turn machinery ever notices. A background sweep, started once from `lifespan()` and running for the lifetime of the backend process, checks every room every sixty seconds and aborts any still **WAITING** more than ten minutes after its creation, broadcasting the updated phase to any client still connected to it. Each room's creation timestamp is recorded once, automatically, when its `GameState` is first constructed. This is a deliberately coarse, low-frequency check: unlike the per-turn timers, there is no live gameplay waiting on its outcome, so correctness only requires that an abandoned room eventually be cleaned up, not that it happen promptly.

3.8 Availability

In-memory state store

Redis is not a cache in the traditional read-through sense, where PostgreSQL would be the source of truth and Redis would only absorb reads. During an active match, Redis *is* the primary read/write store: every game action reads and writes directly in memory, while PostgreSQL is kept up to date with a persistence snapshot for crash recovery (section 3.7), not queried on the hot path. This avoids continuous queries to the relational database during a match; the main advantage is minimal latency and reduced load on PostgreSQL.

Load balancing and scalability

The system is currently single-instance, so there is no load balancing. Nginx acts as a reverse proxy but only forwards requests to a single backend server without distributing the load across multiple machines. As discussed in the Data and Consistency section, the architecture has been designed to support future horizontal scaling via Redis pub/sub.

Client-server connectivity loss

If the network between a client and the server goes down, the client loses its WebSocket connection. From the backend’s perspective, this situation is indistinguishable from a normal disconnection, so the standard two-minute grace period timer is started.

If connectivity is restored within the timeout, the client reconnects and resumes without problems. If the outage lasts more than two minutes, the player is disconnected permanently and, depending on the conditions, the match is aborted.

In general, in the event of a client-server connectivity loss the system makes a deliberate application-level trade-off: it is preferable to abort a match than to continue it with a player whose state is indeterminate. This is not an instance of the CAP theorem in the strict sense, as originally formulated by Gilbert and Lynch [Gilbert and Lynch, 2002]: CAP describes the trade-off between replicas of a distributed data store under a network partition between those replicas, whereas here there is a single authoritative server and no replica to be partitioned from; the client is a display of server state, not a peer holding a copy of it. The connectivity loss described here is better read as a decision about how to treat an *unreachable client*, not about reconciling divergent copies of data.

A closer fit for the trade-offs actually made throughout this system is the PACELC framework [Abadi, 2012], which extends CAP by noting that even absent any partition (the “else” in its name), a system still has to trade off latency against consistency. This is a more accurate description of the Redis/PostgreSQL relationship in section 3.7: there is no network partition between them in the CAP sense, yet the system still deliberately favours low latency (Redis alone on the hot path) over having every read immediately reflect a durably committed PostgreSQL write.

This preference for aborting over continuing is itself conditional on *which* component is unreachable, not an absolute rule applied uniformly everywhere in the system. When the

unreachable party is a *client*, the system leans towards abandoning the match rather than letting it continue in an indeterminate state, as described above. When the unreachable party is instead the *persistence layer* (PostgreSQL), the system takes the opposite stance, as described in section 3.7: the match keeps running on Redis alone, favouring continuity of gameplay over the durability guarantee that PostgreSQL would otherwise provide. The two choices are not contradictory once framed this way, but they do represent different priorities for different failure modes, and it is worth being explicit that the system does not have a single, uniform answer to “consistency or availability” — the right answer here depends on what, specifically, has failed.

3.9 Security

Authentication and session management

Access to all game features is entirely conditional on authentication, which is handled via JWT (JSON Web Token). At registration, the user’s password is hashed using bcrypt before being saved to PostgreSQL; passwords are never stored in plain text under any circumstances.

When a user logs in, the server validates the credentials and returns a signed JWT token. Clients must then include this token in the **Authorization** header for standard REST calls, or as a URL parameter when establishing a WebSocket connection.

Authorisation and role logic

The backend implements role-based access control, verifying the user’s identity on every request. The server ensures that each action is consistent with the player’s role and the match state. Specifically:

- Only active players can perform game moves (drawing cards, placing them, or taking rows).
- Lobby management (starting or cancelling the match) is reserved exclusively for the match creator.
- Observers can only watch and cannot perform any game action.
- A player cannot act outside their own turn under any circumstances.

All these checks are enforced server-side and cannot be bypassed by modifying the client.

Cryptographic schemes

For the protection of sensitive data, well-established standards are used. Passwords are hashed with bcrypt, an adaptive hashing algorithm that provides strong resistance against brute-force attacks. JWT tokens are signed with HMAC-SHA256 using a secret key stored in environment variables. In the production environment, all traffic is encrypted over HTTPS and WSS to prevent man-in-the-middle attacks.

4 Implementation

Network protocols and communication

Two different approaches are combined depending on the requirements. HTTP/1.1 is used for all standard REST calls: account management (registration, login), room creation, and game actions. For everything that requires real-time updates, clients open and maintain a persistent WebSocket connection, which is used to instantly receive state changes from the backend.

Data format

Both REST and WebSocket communication use JSON as the interchange format. Payloads travel with JSON bodies and responses return the match state serialised in the same way. On the backend, Pydantic handles validation and serialisation; on the frontend, the standard JavaScript methods (`JSON.parse` and `JSON.stringify`) are used.

Database interaction

Interaction with PostgreSQL goes through SQLAlchemy in asynchronous mode, using the `asyncpg` driver. SQLAlchemy's ORM is used to write queries in standard SQL without burdening the codebase. The connection to Redis is also asynchronous, managed via the `redis-py` library. Match state is serialised to a JSON string before being saved in Redis; on reading, `dacite` is used to reconstruct the Python dataclasses.

Authentication and authorisation

The JWT mechanism is handled with the `python-jose` library. Tokens include a configurable expiry and are signed with HMAC-SHA256. Passwords are hashed with bcrypt via `passlib`. The `/login` endpoint validates the user and issues the token; protection of subsequent routes is delegated to a FastAPI dependency (`get_current_user`) injected directly into the protected endpoints.

For permissions, no external RBAC framework has been introduced to avoid overcomplicating the project. Everything is checked explicitly server-side: each endpoint manually verifies that the user meets the requirements for that specific action (for example, checking whether it is their turn, whether the match is in the `PLAYING` state, or whether the user is an active player).

4.1 Technological details

FastAPI Chosen as the main backend framework for its strong async performance and native support for WebSocket and Pydantic.

React + Zustand For the frontend user interface, using Zustand to manage global state in a lightweight way.

PostgreSQL & Redis The former for long-term persistence, the latter as an in-memory data store for active matches.

SQLAlchemy & dacite Respectively the async ORM and the tool for converting dictionaries into dataclasses.

Nginx & Docker Compose Nginx acts as a reverse proxy also handling WebSocket connections and rate limiting; Docker Compose orchestrates the containers both locally and in production.

python-jose, passlib/bcrypt & Pydantic For security, password hashing, and data validation.

pytest & pytest-asyncio The framework used to run asynchronous tests.

5 Validation

5.1 Automatic Testing

The backend was developed following a Test-Driven Development (TDD) approach. All tests are implemented using **pytest** and **pytest-asyncio**, and can be run with a single command from the project root:

```
pytest tests/ -v
```

The suite comprises 86 tests in total, structured into three macro-categories: unit tests, integration tests, and a dedicated set of fault-tolerance and timing tests added in the final stage of development.

Category	Tests
Unit	48
Integration	25
Fault-tolerance and timing	13
Total	86

Unit Tests

Unit tests verify the correctness of the game logic in complete isolation, without involving the network, the database, or Redis. For each test, the **GameState** object is instantiated using helper functions defined in **tests/conftest.py**, which then invoke the pure functions in **backend/game.py**.

test_deck.py Verifies deck initialisation and the initial game state. *Requirements covered:* *F3, F5.*

`test_turns.py` Verifies the correct management and alternation of turns, including that a player who has left the match permanently (rather than merely being marked inactive during a grace period) is the one skipped when determining whose turn it is next. *Requirements covered: F7, F8, F15.*

`test_round.py` Verifies round management and the related state transitions. *Requirements covered: F9, F18.*

`test_score.py` Verifies the score calculation algorithm. *Requirements covered: F18.*

`test_observers.py` Verifies the management and tracking of observers in a match. *Requirements covered: F12, F13, F19.*

`test_two_players.py` Verifies the application of the special rules for two-player matches. *Requirements covered: F3.*

Integration Tests

Integration tests verify the interaction between the different software components through the full HTTP stack, including the authentication module, routing, game logic, and state persistence on Redis and PostgreSQL.

Available in `tests/api/test_integration.py`, these tests use `httpx.AsyncClient` with `ASGITransport` to make direct calls to the FastAPI application, simulating client-server interaction without the overhead of a real network connection.

The test environment is isolated by setting the `TESTING=1` environment variable, which forces the backend to connect to a dedicated test database. PostgreSQL and Redis run inside Docker containers, exposing the necessary ports to the local environment.

The integration suite comprises 25 tests, organised by logical domain.

Authentication

`test_register` Verifies that a new user can register and that the response contains the correct username (F1).

`test_register_duplicate_username` Verifies that attempting to register with an already existing username returns HTTP 400 (F1).

`test_login_valid` Verifies that a registered user receives a valid JWT token on login (F2).

`test_login_invalid` Verifies that incorrect credentials return HTTP 401 (F2, NF4).

Room Management

`test_create_room` An authenticated user can create a room; the response includes the room code and the initial `waiting` phase (F3).

test_create_room_unauthorized An unauthenticated room creation request is rejected with HTTP 401 (NF4).

test_join_room A second player can join the room and appears correctly in the participant list (F4).

test_join_room_not_found Attempting to access a non-existent room returns HTTP 404 (F4).

test_join_room_full Attempting to join a room that has already reached the maximum number of players returns HTTP 400 (F4).

test_start_room The host can start the match, transitioning the room state to **playing** (F5).

test_start_room_not_enough_players Attempting to start with fewer players than the minimum returns HTTP 400 (F5).

test_abort_room The host can cancel a waiting match, transitioning the phase to **aborted** (F6).

Gameplay

test_draw_card The current player can draw a card; the response must contain the card marked as **pending_card** (F7).

test_draw_card_not_your_turn Attempting to draw outside one's own turn is blocked with HTTP 400 (F7, NF5).

test_draw_twice_without_place Attempting to draw a second card without having placed the previous one returns HTTP 400 (F7).

test_place_card After drawing, the player can correctly place the card on a row (F8).

test_place_without_draw Attempting to place a card without having drawn first returns HTTP 400 (F8).

test_take_row A player can validly collect a row after a card has been placed on it (F9).

Leave and Abort

test_leave_room A player can leave an ongoing match (F10).

test_leave_room_aborts_game If the number of players who have not permanently left drops below the minimum threshold, the match is automatically aborted by the system (F11).

Observers

`test_observe_room` An authenticated user can observe an ongoing match and is added to the observer list (F12).

`test_observe_room_full` Attempting to add a fifth observer beyond the configured limit is rejected with HTTP 400 (F19).

`test_observe_room_already_player` A user who is already participating as a player cannot also register as an observer (F12).

Sequence Numbers

`test_sequence_number_increases_on_draw` Verifies that the game state sequence number increases strictly after a card draw (NF2).

`test_sequence_number_increases_on_leave` Verifies that the sequence number increases following a player leaving the match (NF2).

Fault-Tolerance and Timing Tests

In the final stage of development, four additional test modules were added under `tests/` to cover scenarios that are, by nature, sensitive to real wall-clock timing and to concurrent access — areas that are easy to get subtly wrong and that unit tests alone do not exercise well.

Timing and race conditions (`test_timing_races.py`) Seven tests exercise the inactivity timer and the room lock under real elapsed time, using a fixture that temporarily shrinks the inactivity timeout to one second so that the tests run in seconds rather than minutes. `test_inactivity_full_cycle_does_not_self_cancel` is a regression test for a self-cancellation bug found during development: the coroutine that handles an inactivity timeout used to cancel its own asyncio task as a side effect of advancing the turn, so the disconnection logic that should have followed was silently never reached. Further tests verify that only the current-turn player may act even under concurrent requests (`test_concurrent_draw_only_current_turn_player_succeeds`), that acting just before the deadline correctly resets the timer against real elapsed time (`test_action_before_deadline_resets_timer`, which uses its own wider timing margins rather than the shared fast-timeout fixture, after the original tighter margins were found to be occasionally flaky under real scheduling overhead), that a double-click style race on the same action is serialised by the per-room lock rather than corrupting state (`test_concurrent_double_draw_same_player`), that the turn-start timestamp used for the client-side countdown is stable across repeated reads and advances only when the turn genuinely changes (`test_turn_started_at_stable_across_reads`), and that inactivity timeouts cascade correctly through several players without ever getting stuck (`test_cascading_timeouts_three_players`).

Finally, `test_rest_action_during_grace_period_reactivates_player` is a regression test for the reactivation gap described in section 3.7 (“Inactivity handling”): it verifies end to end that a player marked inactive is reactivated by their next accepted REST action, and that the expulsion task scheduled by the original timeout is cancelled rather than left to fire at its original deadline during a later inactivity cycle — the orphaned-task race that writing this very test surfaced. *Requirements covered: F14, F15, NF2, NF3.*

Fault tolerance (`test_fault_tolerance.py`)

`test_redis_repairs_room_from_postgres_after_data_loss` validates the cache-aside recovery mechanism described in section 3.7: a room’s entry is deleted directly from Redis to simulate a data loss, and the test verifies that the very next request for that room is transparently repaired from the PostgreSQL snapshot, that Redis is repopulated as a side effect, and that a real game action against the repaired state succeeds. *Requirements covered: F16.*

One further test, added alongside the recovery watcher described in section 3.7, covers the mechanism introduced to handle a Redis-only failure that leaves the room’s key present but stale. `test_reload_playing_games_from_postgres_overwrites_stale_redis` verifies that `reload_playing_games_from_postgres` correctly overwrites a stale in-memory snapshot with the last state persisted in PostgreSQL for a match in the PLAYING phase, counter-testing the core behaviour of `redis_recovery_watcher` on the down-to-up transition. The complementary fix to `room_exists_with_retry` described in section 3.7 (“Redis resilience in background timers”), by contrast, was verified manually rather than by a dedicated automated test, for the same reason already noted for the original retry-with-backoff mechanism it replaces: reliably reproducing the exact timing coincidence required is difficult. *Requirements covered: F16.*

Game flows (`test_game_flows.py`) Three tests cover flows that were previously only exercised implicitly: a player leaving mid-match while the others continue normally (`test_leave_mid_game_others_continue`), a match being correctly aborted once enough players have left (`test_abort_when_enough_players_leave`), and an observer being able to read the match state but rejected with HTTP 403 on every game action (`test_observer_can_view_but_not_act`). *Requirements covered: F10, F11, F12.*

Full playthrough (`test_full_playthrough.py`)

`test_full_three_player_game_reaches_finished_with_scores` plays an entire three-player match end to end through the real REST API, picking rows greedily with no strategy purely to reach a terminal phase. This single test happens to also exercise the “all rows full, must take a row instead of drawing” rule, since that condition arises naturally at some point during a full game. *Requirements covered: F7, F8, F9, F18.*

Test environment isolation

Early in development, the test suite shared both Redis and PostgreSQL with the running development environment: `TESTING=1` only switched SQLAlchemy to `NullPool` and had no effect on which database or Redis instance was actually used. In practice this meant that running `pytest` against a running development stack would create real, visible rooms in Redis, and — more seriously — would drop the `games` table and delete every row from `games` and `users` in PostgreSQL at the end of every test session, requiring a manual backend restart and re-registration of all test accounts.

This was resolved by making `TESTING=1` redirect both stores to a dedicated location: Redis tests now use logical database 1 instead of the default 0, and PostgreSQL tests connect to a separate `chameleon_test` database instead of the development one. Since this second database does not exist by default on a fresh clone, a dedicated `ensure_test_database_exists` routine connects to Postgres' own maintenance database and creates `chameleon_test` on first run if it is missing, so the test suite is fully self-contained and requires no manual setup step beyond `docker compose up` followed by `pytest`.

Environment Issues: Windows Event Loop

Running asynchronous integration tests with Redis on Windows required ad-hoc configuration. By default, Windows uses `ProactorEventLoop`, which tends to close sockets aggressively, conflicting with the asynchronous disconnection mechanism of the `redis-py` client at the end of tests and raising `RuntimeError: Event loop is closed` exceptions.

To ensure suite stability, the test architecture was configured at three levels:

1. **Policy override:** inclusion of `WindowsSelectorEventLoopPolicy` in the root `conftest.py` to mitigate socket crashes.
2. **Session-scoped event loop:** definition of a custom `event_loop` fixture with `scope=session` to prevent the loop from being destroyed and recreated between tests.
3. **Scope alignment:** configuration of `asyncio_default_test_loop_scope = "session"` in `pyproject.toml`, ensuring lifecycle alignment between the event loop, `async` fixtures, and integration tests.

5.2 Acceptance Testing

The following acceptance tests were conducted through controlled manual testing sessions. Due to the inherent limitations of the automated test setup — primarily the use of `ASGITransport`, which simulates HTTP calls in-memory but does not support stateful connections such as WebSockets, nor allows control of the Docker container lifecycle from within test code — the scenarios described below were not included in the `pytest` suite but were validated manually, both in a local staging environment and, for the final round,

directly against the production deployment at <https://chameleon-table.ddnsfree.com>.

Real-time WebSocket updates. Validated by opening multiple browser tabs in parallel and verifying that every game action performed on one client was reflected instantly and simultaneously on all other connected participants and observers.

Crash recovery. Tested by starting an active match, abruptly stopping the backend Docker container with `docker stop`, and then restarting it. The match state was successfully restored from the data persisted in PostgreSQL, allowing players to reconnect and resume the session from where it was interrupted.

Silent Redis restart. Tested by starting an active match and then stopping and restarting only the Redis container via Docker Compose, leaving the backend process itself running throughout. The logs confirmed that `redis_recovery_watcher` detected the down-to-up transition on its next PING cycle and invoked `reload_playing_games_from_postgres`, which overwrote the freshly reloaded (and potentially stale) RDB snapshot in Redis with the last state persisted in PostgreSQL, without requiring a backend restart or player intervention.

Disconnection and grace period. Validated by simulating the sudden closure of a player's browser tab. The system correctly started the two-minute grace period timer, promptly notified the other connected clients, and restored the session without penalty when the player reconnected within the time window.

Inactivity timer. Tested by intentionally leaving a player idle during their turn. After one minute of inactivity, the system automatically removed the player from the active turn and started the corresponding grace period.

Nginx rate limiting. Validated by simulating a burst of high-frequency requests against the exposed endpoints. Beyond the traffic threshold configured in the reverse proxy, Nginx correctly rejected excess requests with HTTP 429 (Too Many Requests).

Mixed-content WebSocket issue on the production deployment. Manual testing against the live HTTPS deployment surfaced an issue that no local test, automated or manual, could have caught: the frontend's fallback WebSocket URL was hardcoded to the insecure `ws://` scheme, which browsers silently block when the page itself is served over HTTPS (a *mixed content* restriction). In the local development environment, where the page is served over plain HTTP, the same code path happened to produce a matching, and therefore working, `ws://` URL, which is why the defect had gone unnoticed until the first live acceptance test. The fix was to derive the WebSocket scheme from the page's own protocol at runtime (`wss://` when the page is HTTPS, `ws://` otherwise) instead of hardcoding it. This is reported here as a concrete illustration of why acceptance testing against the actual production environment, and not only against a local stand-in for it, is a necessary complement to automated testing.

Zombie connection on a mobile network. Tested by placing one client on a mobile hotspot connection and performing a game action from a second client while the hotspot client remained connected but momentarily impaired. The action's broadcast was silently lost for the affected client without its WebSocket ever closing, exactly matching the scenario described in section 3.4. The two-level ping/pong mechanism correctly detected the stale sequence number on the next heartbeat cycle and resynchronised the client via

a single REST call, without ever closing or reopening the WebSocket connection.

Real-time update on room join. Opening two browser windows side by side surfaced a defect where a second player joining a waiting room was not reflected in real time on the host's screen, even though the underlying WebSocket connection was healthy. The cause was that `join_room` (and, separately, `abort_game`) updated the room state and broadcast it without first advancing the sequence number, unlike every other state-mutating endpoint; the host's client therefore discarded the broadcast as a stale duplicate. Both endpoints were corrected to advance the sequence number before broadcasting, consistent with the rest of the API.

Grace period turn freezing. Tested by leaving one player deliberately idle past the one-minute inactivity threshold in a three-or-more-player match, while the other players had already taken their rows for the round. The turn correctly remained frozen on the inactive player for the two-minute grace period rather than passing to another player, and, once the grace period expired without reconnection, the round correctly concluded automatically instead of leaving the match stuck with no player able to act.

6 Deployment

6.1 Prerequisites

The only software required on the host machine is Docker and Docker Compose (bundled together in Docker Desktop on Windows and macOS, or installed as the `docker-compose-plugin` package on Linux). No other dependency needs to be installed manually: Python, Node.js, PostgreSQL, and Redis all run inside their respective containers, with fixed versions pinned in the Dockerfiles (Python 3.12-slim for the backend, Node 20-alpine for the frontend build stage, PostgreSQL 15, Redis 7-alpine).

6.2 Local (development) setup

The repository is a monorepo containing the `backend/`, `frontend/`, `nginx/`, `scripts/`, and `tests/` directories at its root, alongside a base `docker-compose.yml` and a `docker-compose.override.yml` that Docker Compose merges in automatically for local development (it exposes the PostgreSQL and Redis ports to the host, which is convenient for inspecting the database or running the test suite from outside the containers, but is not desirable in production).

From the project root:

```
docker compose up --build -d
```

This builds and starts all five containers (`nginx`, `frontend`, `backend`, `db`, `redis`). On a cold build, PostgreSQL and Redis may take a few seconds to become `healthy` before the backend, which depends on both, actually starts accepting requests; a short pause before running the next command avoids a spurious failure of the seed script below.

```
Start-Sleep -Seconds 15      # PowerShell (Windows); use 'sleep 15' on Linux/macOS
python scripts/seed_db_nodeps.py
```

The seed script is optional: it populates the database with a handful of test accounts (`player1` through `player5`, all sharing the password `password123`) to make manual multi-player testing quicker. It has no external dependencies — it uses only Python’s standard library. The application is then reachable at `http://localhost`. Note that the single-command requirement (I3) refers to the system itself, not to this optional seeding step: the application is fully functional and reachable after `docker compose up` alone, with zero registered users.

To fully reset the local environment, including all persisted data:

```
docker compose down -v
docker compose up --build -d
```

The `-v` flag also removes the named volumes (`db_data`, `redis_data`), so this should only be used when a clean slate is actually desired.

6.3 Environment variables

Both environments read configuration from an `.env` file at the project root (an `.env.example` template is provided in the repository). The relevant variables are:

`DATABASE_URL` the PostgreSQL connection string used by SQLAlchemy.

`REDIS_URL` the Redis connection string.

`SECRET_KEY` the HMAC-SHA256 signing key used for JWT tokens.

`REGISTRATION_ENABLED` a boolean flag that, when set to `false`, disables the `/register` endpoint. This is used in production to prevent unsolicited public sign-ups once the intended set of accounts has been created, while keeping the endpoint itself available for administrative use when temporarily re-enabled.

6.4 Production deployment

The production environment runs on a single Google Cloud Platform `e2-micro` virtual machine (Ubuntu 24.04 LTS), chosen for its eligibility under GCP’s free tier. Docker Compose is used exactly as in development, with an additional override file that is merged on top of the base configuration:

```
docker compose -f docker-compose.yml -f docker-compose.prod.yml up --build -d
```

`docker-compose.prod.yml` differs from the development configuration in three respects: it exposes ports 80 and 443 on the `nginx` container instead of just 80; it mounts the Let’s Encrypt certificate directory (`/etc/letsencrypt`) read-only into the container

so that Nginx can terminate TLS; and it does *not* expose the PostgreSQL and Redis ports to the host, since there is no legitimate reason for those services to be reachable from outside the Docker network in production.

Every service in the base `docker-compose.yml` is configured with `restart: unless-stopped`. This was added after observing that, without it, a container that crashed or that was interrupted by a VM reboot (which GCP's free-tier virtual machines are occasionally subject to for host maintenance) would simply stay down until someone noticed and restarted it manually. With the policy in place, Docker itself restarts the affected container automatically, both on an unexpected crash and after the whole VM reboots; a container that is deliberately stopped by an operator with `docker compose stop`, however, correctly stays stopped, since the policy only reacts to unintended terminations.

Since the VM's external IP address is ephemeral and changes on every reboot, a dynamic DNS provider (Dynu) maps the domain `chameleon-table.ddnsfree.com` to the VM's current IP address; the mapping must currently be updated by hand after a reboot, and this manual step is the single most fragile part of the whole deployment. Let's Encrypt, via Certbot, issues the TLS certificate for that same domain name, which Nginx then uses to serve the application over HTTPS.

Deploying an update to the running production instance is a matter of pulling the latest code and rebuilding:

```
git pull
docker compose -f docker-compose.yml -f docker-compose.prod.yml \
    up --build -d
```

Crucially, this command sequence does *not* include the `-v` flag, so the PostgreSQL and Redis volumes, and therefore all real user accounts and match data, are preserved across deployments.

6.5 Quick Testing Walkthrough

This is a condensed, six-step walkthrough for anyone who wants to try the system quickly, without re-reading the sections above in full.

1. Start the system

From the project root:

```
docker compose up --build -d
```

Wait about 15 seconds for PostgreSQL and Redis to become healthy before proceeding.

2. Create test accounts

```
python scripts/seed_db_nodeps.py
```

This creates five accounts (`player1` through `player5`, all with password `password123`). The script has no external dependencies — it uses only Python's standard library, so no `pip install` is required beforehand.

3. Play a match with multiple players

Open two or more browser tabs at `http://localhost` (using separate incognito/private windows is recommended, so that each tab keeps its own login session instead of sharing browser storage). Log in as a different test account in each tab, have one player create a room, share the room code with the others, join, and start the match once at least two players are in.

4. Try reconnection

Mid-match, close one player's tab entirely, then reopen `http://localhost` and log back in with the same account within the two-minute grace period that starts as soon as the connection drops. The player should resume the match normally; the other participants see the player marked inactive in the meantime and reactivated on reconnection. (Note: the separate one-minute inactivity timeout described in section 2 only applies while a player stays connected but does not act on their turn; closing the tab disconnects immediately and starts the two-minute grace period directly, without going through that first stage.)

5. Try crash recovery

While a match is in progress:

```
docker compose stop backend
docker compose start backend
```

After the backend restarts, refresh the game tabs: the match state should be restored from PostgreSQL, and players can continue from where they left off.

6. Run the automated test suite

```
pytest tests/ -v
```

This runs all 86 tests described in section 5.1, covering game logic, the REST API, and the timing/fault-tolerance scenarios discussed throughout this report.

7 User Guide

This section walks through the typical use of Chameleon Table from the perspective of a player, following the same order in which the corresponding screens are encountered.

7.1 Registration and login

A new user registers by providing a username, an email address, and a password from the *Auth* page (accessible at `/login`). Once registered, the same page is used to log in; on success, the browser stores a JWT access token and the user is redirected to the Lobby, shown in fig. 7.

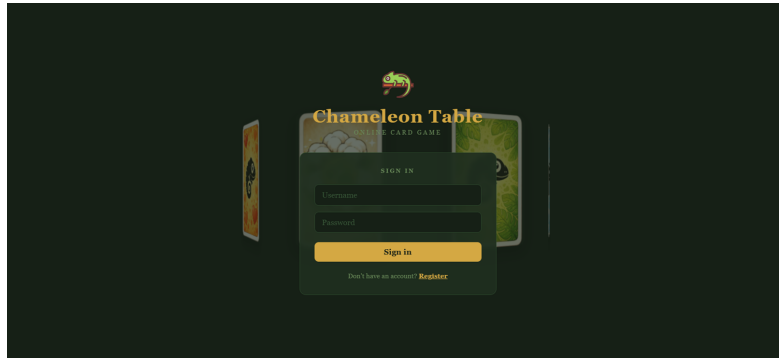


Figure 7: Auth page — sign in form

7.2 Lobby

The Lobby (/) is the landing page for an authenticated user. It shows:

- a form to create a new room, where the host chooses the maximum number of players (from 2 to 5);
- the list of *Available rooms* — matches still in the **WAITING** phase that can be joined — refreshed automatically every three seconds via REST polling;
- the list of *Live games* — matches currently in the **PLAYING** phase, which can be watched as an observer;
- a link to the *Rules* page, and a *Logout* button.

This is the one place in the system where polling is used: it only refreshes the lobby’s room listing, a low-frequency, non-time-critical view. It does not conflict with NF1, which specifically concerns real-time propagation of *game state* changes once a match is underway — those are always pushed via WebSocket, never polled.

Rooms the user is already part of are visually highlighted and labelled “*you’re in*”, with a *Rejoin* button in place of *Join*, so that a user who navigates away and comes back can easily find their way into an already-started or already-joined match, as shown in fig. 8.

7.3 Waiting room

Once a room is created or joined, the user lands on the Waiting Room (`/waiting/{roomCode}`), which lists the players who have joined so far, out of the maximum chosen at creation. The player who created the room (the host) additionally sees a *Start game* button, enabled only once at least two players have joined, and a *Cancel room* button to abort the match before it starts. Non-host players simply wait, with the page updating in real time via WebSocket as other players join, as shown in fig. 9.

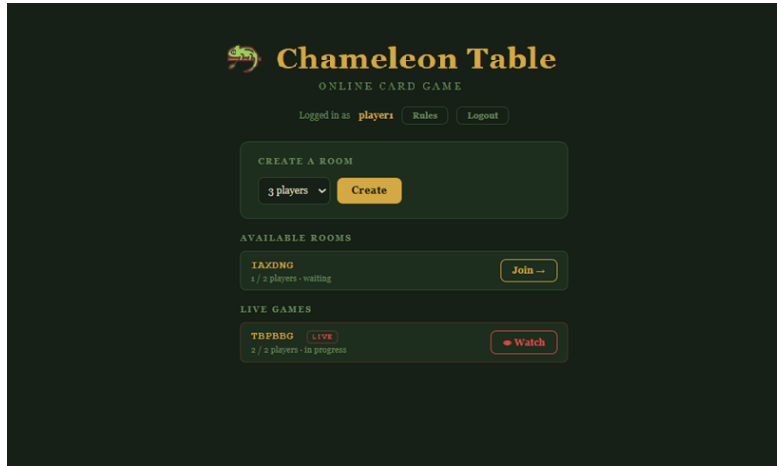


Figure 8: Lobby — create room form, available rooms, and live games

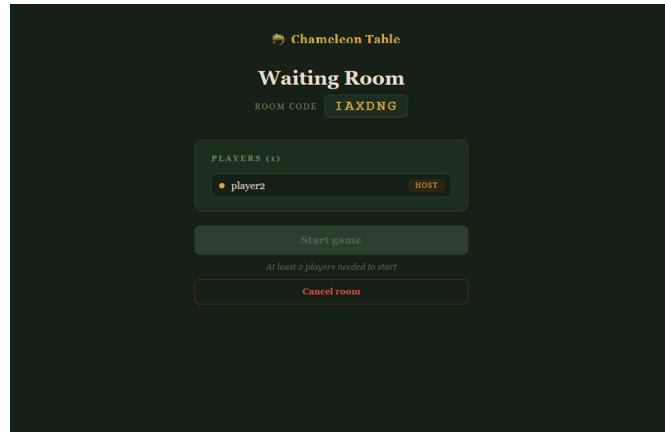


Figure 9: Waiting room — player list and host controls

7.4 Playing a match

Once the host starts the match, all players are moved to the Game screen (`/game/{roomCode}`), which shows the shared rows in the centre, the current player’s hand and jokers at the bottom, the opponents’ hands around the edges, the remaining deck count, and a circular countdown timer next to the current player’s name.

On their turn, a player can either *draw* a card from the deck — which is then shown as a pending card until it is placed on one of the available rows — or, once at least one row has a card on it, *take* that row entirely. The interface only allows the action that is currently legal: the draw button is disabled once every row is full, at which point the player must take a row instead.

The countdown timer reflects the real, server-side inactivity deadline: it is computed from a timestamp the server includes in every state update, rather than from a fixed

value that would drift out of sync after a page reload.

If the current player becomes disconnected or unresponsive past the one-minute inactivity threshold, their turn freezes — rather than passing to another player — for the duration of a two-minute grace period. During this window, the other participants see a distinct amber/red countdown in place of the normal turn timer, the header text changes to indicate that the match is waiting for that player to reconnect, and a pulsing “reconnecting...” badge appears on their panel, as shown in fig. 10 from an opponent’s point of view.

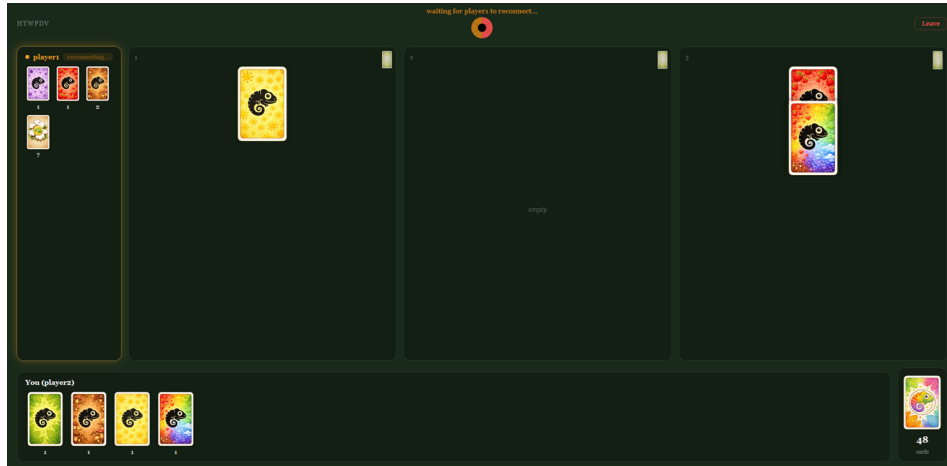


Figure 10: Game screen from an opponent’s perspective, showing the grace-period countdown and “reconnecting” badge

A player can leave the match at any time via the *Leave* button, after confirming in a modal dialog. fig. 11 shows the screen during an active match.

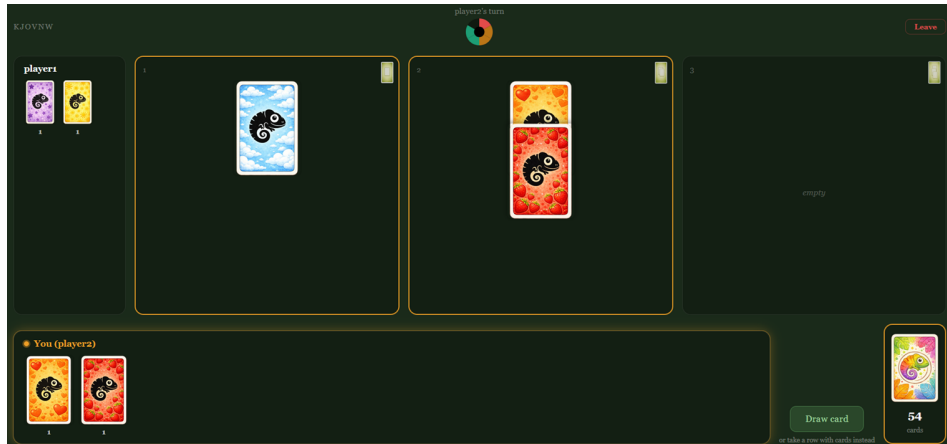


Figure 11: Game screen — rows with capacity icons, hand, turn timer, and draw/take actions

7.5 Observing a match

A user who chooses *Watch* on a live game from the Lobby joins as an observer (`/observe/{roomCode}`). Observers see the same shared game state as players, updated in real time, but have no way to draw, place, or take cards; any such attempt is rejected by the server. An observer can leave at any time without affecting the ongoing match.

7.6 End of match and scores

When the last round completes, the match transitions to the **FINISHED** phase and every player is shown the final scores, computed from each player's three best colours plus any +2 bonus cards, as described by the rules, as shown in fig. 12. If a match is aborted instead — either cancelled before starting or interrupted by too many players leaving — players are returned to the Lobby without a score screen.

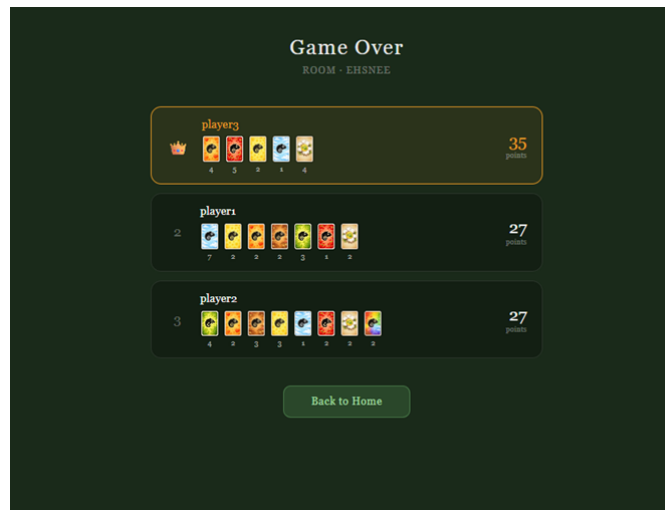


Figure 12: Results screen — final scores per player

7.7 Rules page

The *Rules* page, reachable from the Lobby, restates the mechanics of the game (drawing and placing cards, taking rows, scoring) adapted to Chameleon Table's supported range of 2 to 5 players, and clarifies two points that are specific to the online version and not covered by the original board game: a match can continue with as few as two active players, and if a player leaves or disconnects mid-round, the number of rows on the table does not change until the following round, as shown in fig. 13.

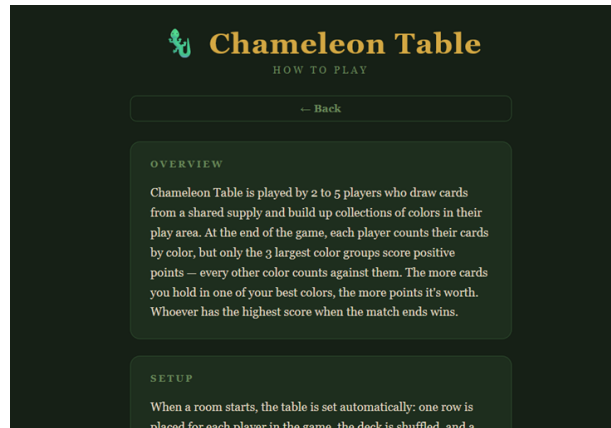


Figure 13: Rules page

8 Self-evaluation

Strengths

The project covers the core requirements of the Distributed Systems course, in particular crash recovery, real-time synchronisation via WebSocket, and fault tolerance managed through the two-level Redis/PostgreSQL approach, complemented by a lazy cache-aside repair path for the case where only Redis, and not the backend process, fails. Network-level resilience is addressed beyond simple disconnection detection: a two-level WebSocket heartbeat distinguishes a genuinely dead connection from one that is still alive but has silently missed a broadcast, a distinction that proved necessary after observing exactly this failure mode on a real mobile network during manual testing. The codebase has been kept reasonably clean by separating the game logic from the network protocol handling, making it more readable. For ease of submission and testing, the entire application is containerised with Docker Compose and starts with a single command; a small deployment on a Google Cloud Platform virtual machine with HTTPS has also been set up and used for the final round of acceptance testing. The test suite, expanded in the final stage of development with dedicated timing and fault-tolerance tests, now comprises 86 tests covering the core game logic, the main REST endpoints, and several timing-sensitive edge cases that would otherwise be difficult to reproduce reliably by hand. The test environment is also fully isolated from development: both Redis and PostgreSQL route to dedicated instances when running under `pytest`, with the test database created automatically on first run so that a fresh clone of the repository requires no manual setup step.

Weaknesses

The main limitation is that the system is currently single-instance and does not scale horizontally, although the Redis-based architecture would ideally allow this in the future.

The frontend interface is very minimal, designed just to test matches on desktop and not optimised for mobile at all. Error handling on the client side also needs refinement, as it currently only covers the main cases. More generally, several of the bugs discussed in this report — the missing sequence-number advance on room join, the round that could become stuck after the last player’s grace period expired, and the earlier test-isolation gap — were found through manual testing or incidentally while implementing an unrelated feature, rather than by the automated test suite itself; this suggests the suite, while substantial, still has room to grow in the coverage of less common state transitions.

Role in the project

The project was developed individually, covering both the backend with FastAPI and the frontend in React, including the Docker container configuration, JWT authentication, and the WebSocket logic for game turns.

9 Future works

Horizontal scaling and high availability

The biggest limitation of the project is that it runs on a single instance. To allow the backend to scale horizontally and handle many more concurrent matches, the plan would be to introduce Redis pub/sub. Pub/sub is a well-established pattern for exactly this kind of fan-out [Eugster et al., 2003]: publishers (here, the backend instance that just handled a mutating request) emit events without needing to know which subscribers (the other backend instances, and transitively their connected clients) exist or how many there are, decoupling producers from consumers in both space and time. In this way, multiple server instances could exchange match state information, replacing the current in-process locks with distributed locks, also on Redis. Furthermore, to eliminate the single point of failure of the single server, a standby instance could be configured for failover, ready to take over automatically if the primary node goes down.

It is worth being explicit about what pub/sub alone would and would not solve, since it is easy to overstate. Pub/sub addresses exactly one problem: propagating a state-changing broadcast from the instance that handled the originating request to every other instance, so their connected clients can be reached too. It does not, by itself, address at least three further problems that a genuinely multi-instance deployment would need to solve: which instance holds the lock for a given room at a given moment (a distributed lock along the lines of Redlock would need its own scrutiny regarding correctness under instance failure and clock drift, a subject with known open debate in the wider community [Kleppmann, 2016]); how a client’s WebSocket connection, once accepted by one instance, is kept reachable when a subsequent REST action for the same room happens to land on a different instance; and whether individual game actions remain safe to retry (idempotent) if a request is resent after an ambiguous failure. Each of these would need its own design before horizontal scaling could be considered solved rather than merely planned for.

Real browser-level end-to-end testing

The test suite now covers pure game logic through unit tests and the full HTTP/WebSocket-adjacent stack through integration and fault-tolerance tests driven via `httpx.AsyncClient`. What it still does not cover is the frontend itself: no test currently drives a real browser, opens an actual WebSocket connection the way the React client does, or exercises the Docker container lifecycle from within a test. Introducing a browser-level end-to-end framework such as Playwright would close this last gap and would, in particular, have been able to catch the mixed-content WebSocket issue described in section 5.2, as well as the zombie-connection and stuck-round scenarios discussed above, automatically rather than only through manual acceptance testing.

Frontend refactoring and UX

The graphical interface does its job but is quite bare. Among the first improvements to make would be optimising the layout for mobile devices, since it is currently designed for desktop only. It would also be worth adding some animations to make turns less static, introducing sound notifications, and generally improving the user interface.

Additional features: observers and OAuth

There are two interesting features that could be implemented at a later stage.

Advanced observer mode: today observers see exactly the same screen as players. It would be nice to separate the views, offering observers a dashboard with real-time statistics and the complete history of the match moves.

Social authentication (OAuth): instead of managing the entire registration and login flow in-house with custom JWTs, integration with external providers (such as Google or GitHub) would make access much faster and more secure for the user.

References

- Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.
- Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.
- Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
- Ian Fette and Alexey Melnikov. The websocket protocol. RFC 6455, IETF, 2011.
- Gabriel Gambetta. Fast-paced multiplayer (part i): Client-server game architecture. <https://www.gabrielgambetta.com/client-server-game-architecture.html>, 2013. Accessed 2026.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

Martin Kleppmann. How to do distributed locking. <https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html>, 2016. Accessed 2026.