UFG Report

Paradisi Giovanni

Samite Mounir
Master Degree in *Ingegneria e Scienze Informatiche*

University of Bologna
Course: **Distributed Systems**

Academic year: **2025/2026**

Date: **26 June 2026**

Repository: **https://github.com/giovihey/UFG**

Group: **Paradisi Giovanni / Samite Mounir**

AI disclosure: AI tools supported report drafting from our documentation, tables/diagrams, and parts of the C++ WebRTC work; all content was reviewed and edited by the authors.

Table of Contents

1 Introduction

UFG is a small distributed 2D fighting game built to explore a hard real-time problem: keeping two machines in agreement while both players expect immediate input response. The project combines a desktop game client, an authentication service, a signaling service, and a direct peer-to-peer gameplay channel.

The main contribution is the match architecture. During a fight, there is no authoritative game server. Each client runs the same deterministic simulation and exchanges only player inputs through a WebRTC data channel. Rollback netcode absorbs late inputs by rewinding and replaying local state, while committed-frame hashes detect cases where the two simulations diverge.

The backend services are intentionally modest. Authentication stores accounts and issues session tokens. Signaling pairs players and relays WebRTC setup messages, then leaves gameplay traffic to the peers. This keeps the distributed system focused on the central consistency and latency problem rather than on a large online platform.

The report first defines the product boundary and requirements, then explains the design, implementation, validation, deployment, and player-facing workflow. It ends with future work and self-evaluation, including the parts intentionally left outside the submitted scope.

2 Concept

UFG is a distributed real-time 2D fighting game. Two players, each on their own machine, connect peer-to-peer and fight in real time: every frame each client samples its local input, exchanges it with the opponent, and runs the same deterministic simulation. Two small backend services support the game without ever taking part in the fight: one for authentication, one for matchmaking and connection setup.

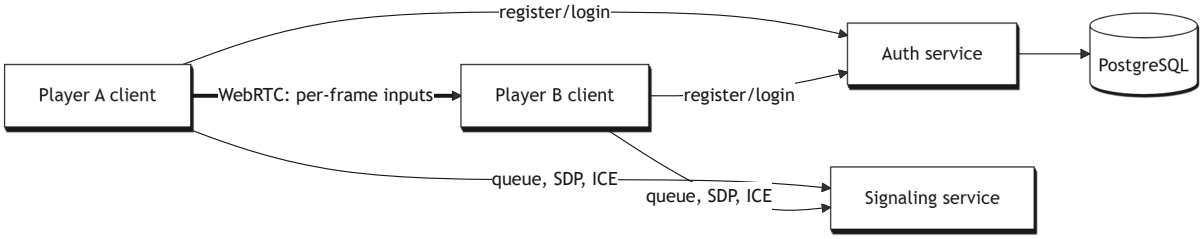

Figure 2.1: High-level view of the system.

2.1 Product Type

The main product is a **desktop application with a GUI**, written in Kotlin with Compose Desktop. It bundles the user-facing screens (login/register, menu, matchmaking practice, VS splash, fight, match-end) and the deterministic simulation that runs the match.

It is supported by two **web services**, both written in Go:

- an **authentication service**, backed by PostgreSQL, that manages accounts and issues JWT session tokens;
- a **signaling service** that pairs players over WebSocket and relays the WebRTC setup messages between exactly two matched peers.

There is no public CLI or reusable library: scripts and the Makefile are only developer conveniences.

2.2 Use Case Collection

2.2.1 Where are the users?

On desktop computers (macOS, Linux, or Windows), either on the same local network or anywhere across the internet, as long as both clients can reach the same configured auth and signaling services. WebRTC handles NAT traversal through a public STUN server, so players behind a home router connect directly without manual port forwarding.

2.2.2 When and how often do they interact?

Interaction has two rhythms:

- **Before a match** it is occasional: the user logs in, presses *Play*, waits in matchmaking, and may cancel. A local practice loop keeps the wait interactive.
- **During a match** it is continuous: the game targets 60 FPS, so each client samples input, simulates, and exchanges inputs with the opponent roughly every 16 ms.

2.2.3 How do they interact? Which devices?

Through the Compose Desktop GUI with mouse and keyboard. Menus use the mouse; gameplay uses the keyboard. At concept level the flow is account access, matchmaking with a local practice loop, a synchronized peer-to-peer fight, and a match result. The exact key bindings and screen-by-screen player instructions are kept in the user guide.

2.2.4 Does the system store user data? Which? Where?

Very little is persisted. Only the authentication service holds durable data.

Data	Where	Lifetime
Username, user id	PostgreSQL	Persistent

Password hash (bcrypt)	PostgreSQL	Persistent
JWT session token	Client memory	Until app closes
Matchmaking queue and rooms	Signaling memory	Until matched
Game world, inputs, snapshots	Client memory	One match

Table 2.1

No match history, rankings, or replays are stored.

2.2.5 Roles

There is one business role, the **player**: create an account, log in, queue, play, and return to the menu. No admin, moderator, or spectator role exists.

During connection setup two technical peer roles appear:

- **Host (offerer)**: the player already waiting; creates the WebRTC offer and picks the shared start time.
- **Joiner (answerer)**: the player who joins next; answers the offer.

Once the connection is open the two peers are symmetric: neither is an authoritative server, and combat is computed identically on both sides.

2.3 Scope Boundaries

Several omissions are intentional in this version because the project focuses on the distributed real-time match rather than on a complete commercial fighting game platform. There are no rankings, replays, spectators, mid-match reconnection, authoritative anti-cheat, or replicated backend services.

This concept fixes the product boundary and user context. The next chapter turns that context into testable requirements and the distributed-system properties that the design must preserve.

3 Requirements and Analysis

This chapter defines what UFG must do before discussing how it is built. The requirements focus on observable behaviour, then on the distributed-system qualities that make that behaviour possible.

There is one business actor: the **player**. A player registers or logs in, queues for a match, fights an opponent, and sees the result.

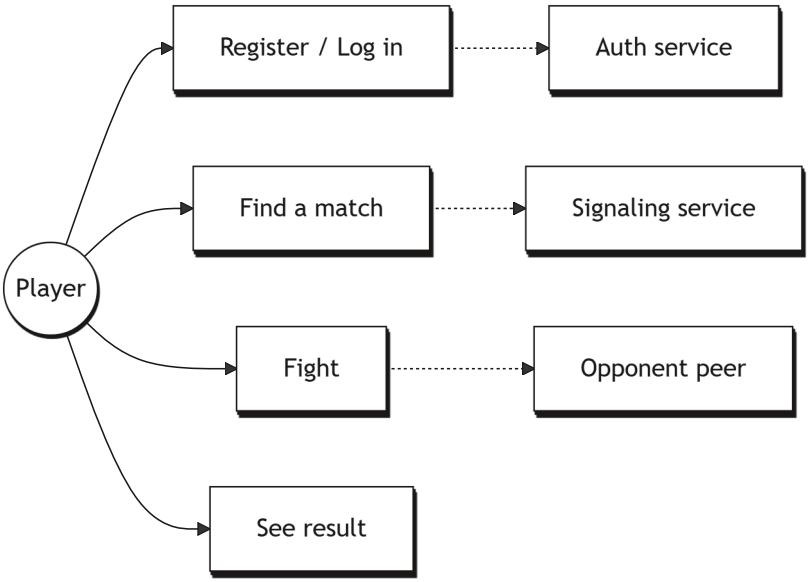

Figure 3.1: Use-case overview.

3.1 Functional Requirements

ID	Requirement	Acceptance criterion
FR1	A player can create an account and log in.	Registering a fresh username returns a token; logging in with the same credentials returns a token; wrong credentials are rejected.
FR2	Logged-in players are paired automatically.	With three queued clients, the first two are matched and the third waits.

FR3	Matched players establish a direct peer-to-peer connection.	Two clients exchange setup data through signaling and open a WebRTC data channel without manual host/join flags.
FR4	Players can move, jump, crouch, punch, and kick.	Inputs update movement and select attacks from the current character's move list.
FR5	Active attacks deal damage once per activation.	An active hitbox overlapping the opponent's hurtbox reduces HP and does not hit twice in the same swing.
FR6	Hits apply hitstun and knockback.	A hit player cannot act during configured hitstun and is pushed away from the attacker.
FR7	Matches are played as timed rounds.	A KO or timer expiry awards a round; reaching the configured round target ends the match.
FR8	Both machines stay consistent without an authoritative server.	Once both inputs for a frame are known, both peers converge to the same committed state.
FR9	The game renders the match clearly.	Fighters, health bars, timer, round pips, and active hitboxes are visible and scale with the window.
FR10	Matchmaking stays interactive.	Practice mode remains playable while queued, and Cancel returns to the menu.

Table 3.1

3.2 Non-Functional Requirements

ID	Requirement	Acceptance criterion
NFR1	Determinism.	Identical inputs produce identical `World` states.
NFR2	Fixed 60 Hz simulation.	Simulation steps are independent of render rate.

NFR3	Low-latency input transport.	Gameplay packets prefer freshness over reliable retransmission.
NFR4	Testability.	Core game logic can be tested without GUI, socket, database, or OS-specific state.
NFR5	Credential security.	Passwords are not stored in plaintext; sessions are signed and expiring.
NFR6	Confidential gameplay transport.	Peer-to-peer gameplay traffic is encrypted.
NFR7	Graceful backend shutdown.	Services stop accepting new work and drain in-flight work on termination.

Table 3.2

3.3 Implementation Constraints

Constraint	**Reason**
Kotlin / JVM client	Team familiarity and cross-platform desktop support
Compose Desktop	Native Kotlin UI with a simple render adapter
Go backend services	Small static binaries with straightforward concurrency
PostgreSQL	Durable account storage is relational and small
libdatachannel + JNI	WebRTC data channels from the JVM
Hexagonal client structure	Keeps deterministic domain logic isolated from I/O
Docker Compose	Reproducible local backend stack

Table 3.3

3.4 Distributed-System Focus

The important distributed concerns are narrow:

- **Consistency** is critical. Both peers must agree on committed match state.
- **Performance** is critical. Inputs must travel directly and quickly enough for a real-time fight.
- **Security** is partial. Accounts are authenticated and traffic is encrypted, but there is no authoritative anti-cheat.
- **Scalability** is limited to many independent two-player rooms on one signaling process.
- **Fault tolerance** is limited. Backend shutdown is graceful, but a mid-match peer disconnect ends the match.

3.5 Terms

Term	Meaning
Frame	One simulation tick at 60 Hz.
Input state	Bitmask of buttons held for a frame.
Frame data	Startup, active, and recovery timing for an attack.
Hitbox / hurtbox	Rectangles used to decide whether an active attack connects.
Hitstun / knockback	The temporary inability to act and pushback caused by a hit.
Rollback netcode	Predict remote input, simulate immediately, then rewind and replay when late input disagrees.
Committed frame	A frame old enough to have left the rollback window.
Desync	A committed-state mismatch between peers.

Table 3.4

With the required behaviour fixed, the next chapter explains the design that satisfies it.

4 Design

This chapter explains the architecture and consistency mechanisms behind UFG. The central design choice is simple: gameplay is peer-to-peer, while account management and matchmaking are handled by small supporting services.

4.1 Architecture

During a match, both clients are equal peers. Each client owns a local copy of the world, samples its player's input, exchanges inputs with the opponent, and runs the same deterministic simulation. There is no authoritative game server in the gameplay path, which avoids adding a server hop to every frame.

The backend is deliberately narrower. The auth service handles accounts and session tokens; the signaling service pairs two players and relays WebRTC setup messages. Once the data channel is open, gameplay inputs flow directly between the clients.

Internally, the client follows a hexagonal architecture. The domain is pure game logic; the application layer owns the loop and rollback service; infrastructure adapters provide rendering, keyboard input, authentication, signaling, and WebRTC.

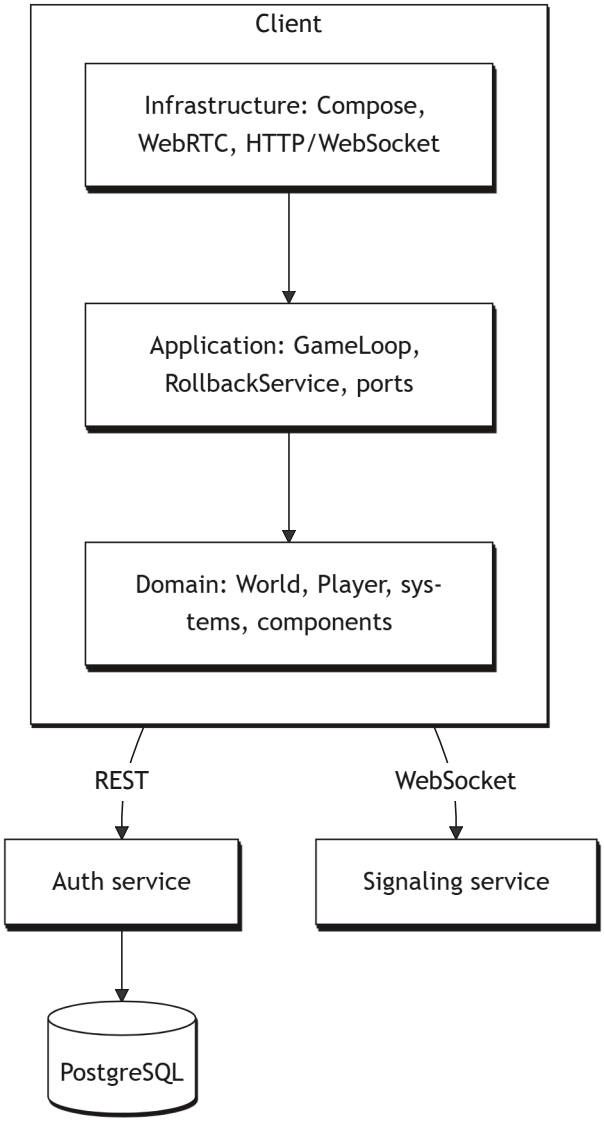

Figure 4.1: Client layering and supporting services.

The dependency rule is strict: inner layers never depend on outer layers. That is what lets the simulation be tested without a GUI, socket, database, or clock.

4.2 Components

The deployed system contains five component types:

Component	Role
Game client	Runs the GUI, simulation, rollback netcode, and WebRTC data channel

Auth service	Issues and verifies session tokens
PostgreSQL	Stores the durable account records
Signaling service	Holds the matchmaking queue and relays setup messages inside a two-player room
STUN server	Helps peers discover a direct path through NAT

Table 4.1

The backend services and database can be deployed together because they are not on the frame-by-frame gameplay path. Clients are configured with the auth URL, the signaling URL, and the STUN address.

4.3 Domain Model

`World` is the state of one simulation frame. It contains two `Player` values, the stage, the timer, and the match status. `Character` is a loaded template that provides health, hurtbox, movement, and move-list data. These values are Kotlin data classes, so every simulation step returns a new state instead of mutating the old one.

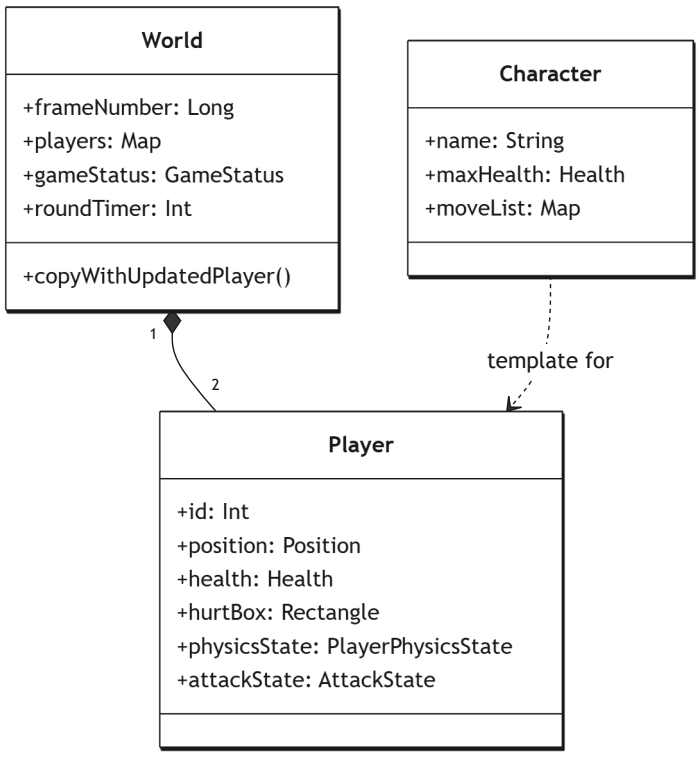

Figure 4.2: Domain model of a match.

4.4 Interaction

A session moves through three phases: login over REST, matchmaking and setup over WebSocket, then gameplay over a direct WebRTC data channel.

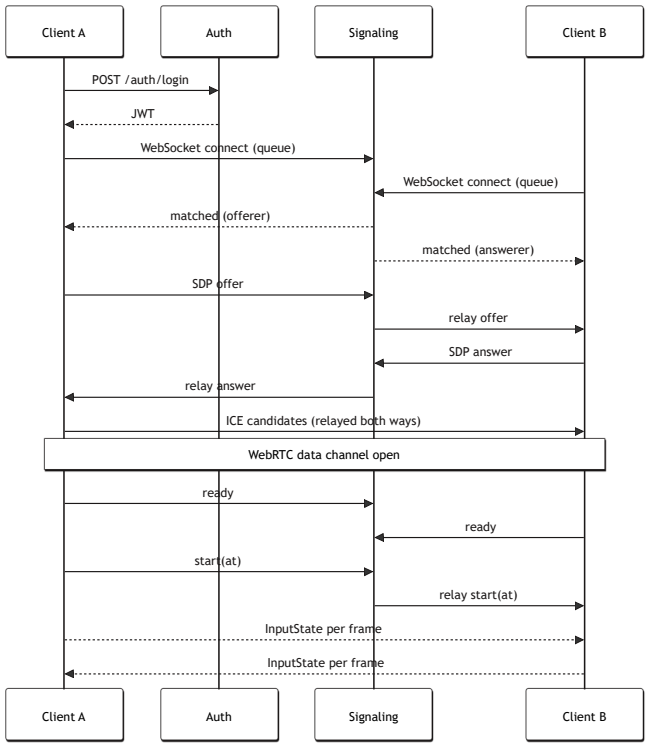

Figure 4.3: End-to-end interaction, from login to first input exchange.

The start handshake keeps the two simulations aligned. After both data channels are ready, the offerer broadcasts a shared start time (`now + 500 ms`). Both clients wait for that barrier before constructing their real game loops; the UI may keep the VS splash visible longer for presentation.

4.5 Simulation and Consistency

Every frame runs the same pure pipeline:

```
1   InputSystem -> AttackSystem -> PhysicsSystem -> HitDetectionSystem
```

Because both peers share only inputs, not world state, the simulation itself must be deterministic. Four properties support that:

1. Pure systems: no randomness, wall-clock reads, or external I/O in domain code.
2. Fixed timestep: physics advances with the same 1/60 s step on every machine.
3. Immutable state: each `World` is a snapshot that rollback can restore.
4. Rollback convergence: late authoritative inputs trigger rewind and replay, so committed frames eventually match on both peers.

Each gameplay packet also carries the sender's latest committed-frame hash. If a peer reports a different hash for the same committed frame, the client emits a desync event. The current implementation detects and logs the problem; it does not repair the match automatically.

4.6 Fault Tolerance and Availability

The design favours consistency over availability during a match. If one client runs too far ahead, it stalls briefly rather than letting the simulations drift outside the rollback window. Each packet repeats a small window of recent inputs, so isolated packet loss is usually covered by the next packet without retransmission.

The limits are intentional. There is no gameplay replication, no mid-match reconnection, and no automatic desync recovery. If the data channel closes, the game loop exits cleanly. The signaling service may notice the room cleanup, but it is not required for already-established gameplay.

The backend is also simple: one auth service, one signaling service, and one database. The signaling service can host many two-player rooms until bounded by the resources of that single process; there is no load balancer or replicated matchmaking state.

4.7 Security

Credentials are handled only by the auth service. Passwords are stored as bcrypt hashes, successful login returns a signed JWT, and repeated attempts are rate-limited. Gameplay traffic uses WebRTC's DTLS-protected data channel.

Authorization is minimal because the game has one player role. The practical access rule is room isolation: setup messages are relayed only between the two clients matched into the same room.

The next chapter maps these design choices to concrete protocols, data formats, storage, and libraries.

5 Implementation

This chapter maps the design to concrete technologies. Each protocol is chosen for the phase it supports: convenience before the match, low latency during it.

5.1 Protocols and Data

Phase	Protocol stack	Reason
Account management	HTTP/1.1 REST with JSON	Simple stateless request/response
Matchmaking and setup	WebSocket text messages	Persistent relay for `matched`, SDP, ICE, `ready`, `start`, and `peer_left`
Gameplay	WebRTC data channel: SCTP over DTLS over UDP	Encrypted peer-to-peer datagrams with NAT traversal

Table 5.1

The gameplay channel is unordered and has no retransmission. A stale input is less useful than the next fresh packet, so reliability is handled at the game layer by sending a redundant window of recent inputs.

Data representation follows the same split. Auth and signaling use JSON because those messages are off the hot path and easy to debug. Per-frame gameplay uses a fixed binary packet containing input mask, input frame, sender frame, committed frame, and committed hash. Both peers must agree on this layout.

5.2 Matchmaking Queue

The signaling server keeps matchmaking very small. It stores at most one waiting player. When the first player connects, they are put in this waiting slot. When a seco-

nd player connects, the server removes the waiting player, creates a room for the pair, and tells both clients that they have been matched.

The waiting player becomes the offerer and creates the WebRTC offer. The second player becomes the answerer. After that, the signaling server only relays setup messages between those two clients. Other matched pairs get their own rooms, so their setup messages do not mix.

If a player disconnects before being matched, the waiting slot is cleared. If a player disconnects after being matched, the room is dissolved and the other client is notified.

5.3 Deterministic Simulation

The game runs on a small engine we wrote ourselves rather than an off-the-shelf framework. The reason is determinism (NFR1): rollback only works if the same state and the same inputs always produce the same next state, and a general game engine hides too much non-deterministic behaviour (threading, floating-point, wall-clock reads) to make that guarantee.

The whole simulation is a single pure function, `GameLogic.step(world, inputs)`, that runs a fixed, ordered pipeline of systems and returns the next `World`:

1. `InputSystem` — turns each player's input into intent (movement, jump, attack);
2. `AttackSystem` — advances active attacks frame by frame;
3. `PhysicsSystem` — applies gravity, movement, and knockback at the fixed step;
4. `HitDetectionSystem` — resolves hitbox/hurtbox overlaps, once per activation;
5. game rules — ticks the round timer and resolves rounds and the match;
6. the frame counter is incremented.

The order is fixed and every system iterates players in a stable order (by id), so a replay produces bit-identical results regardless of the underlying map implementation.

Two thin application-layer pieces drive this core. `GameEngine` holds the current `World`: `step()` advances it through `GameLogic`, and `setWorld()` restores a past `World` — the single seam the rollback service uses to rewind. `TimeManager` decouples simulation rate from render rate with a fixed-timestep accumulator: it sums real elapsed time and releases discrete 60 Hz steps (`FRAME_DT = 1/60`), so physics never depends on the actual

frame rate. `GameLoop` ties it together each tick: poll local input, advance the simulation, render.

5.4 Rollback Netcode

Rollback netcode is used so the game does not freeze while waiting for the other player's input. In a fighting game, waiting even a few frames feels bad. The client therefore keeps playing immediately and fixes the result later if the remote input arrives late.

The implementation lives in `RollbackService`, which is called once per game frame by `GameLoop`. On every tick it does five simple things:

1. It records the local player's input for a near-future frame.
2. It sends that input, plus a few recent inputs, to the peer. Repeating recent inputs helps when one packet is lost.
3. If the peer's input for the current frame has not arrived, it guesses by reusing the last known remote input.
4. Before simulating a frame, it saves the current `World` as a snapshot.
5. It advances the game by running the normal deterministic game logic.

When the real remote input arrives, the service compares it with the guess. If the guess was correct, nothing special happens. If it was wrong, the service loads the saved snapshot from before that old frame and replays the frames up to the present using the correct input. This is the "rollback": go back to a known state, then simulate forward again.

This works because the domain code is deterministic. The same `World` plus the same inputs always produces the same next `World`. Each packet also carries the sender's current frame and a hash of an older committed frame. The frame number helps stop one client from running too far ahead; the hash helps detect if the two clients have silently diverged.

5.5 Authentication Service

The auth service exposes three endpoints behind a standard `net/http` router: `POST /auth/register`, `POST /auth/login`, and `GET /health`. Register checks that the usern-

ame is non-empty and the password at least six characters, rejects a duplicate username with `409`, hashes the password, stores the account, and returns a token (`201`); login verifies the password and returns a token (`200`), answering `401` on any bad credential. `/health` pings the database so an orchestrator can tell the service is actually ready, not merely running.

Storage. Accounts are the only durable state, kept in PostgreSQL through `pgx` with parameterized queries over a bounded connection pool. Each row holds an id, a username, and a bcrypt hash — never a plaintext password.

Passwords and tokens. Passwords are hashed with bcrypt before storage, so a leaked database does not expose them. On success the service returns a JWT signed with HS256 that carries the user id and a 24-hour expiry; the secret is read from the environment, and the service refuses to start without it.

Rate limiting. A per-IP token-bucket limiter (2 requests/s, burst 5) fronts both endpoints and answers `429` when exceeded, blunting brute-force attempts.

5.6 Technology Stack

5.6.1 Game client

Concern	Choice
Language / toolchain	Kotlin 2.1.10 on JDK 17
UI	Compose Desktop 1.9.3
Serialization / signaling	kotlinx-serialization-json 1.8.0, `org.json` 20251224, Java-WebSocket 1.5.6
Native bridge	JNI wrapper over libdatachannel
Logging	slf4j 2.0.13, Logback 1.5.13, kotlin-logging 7.0.0
Tests / quality	Kotest 5.8.0, MockK 1.13.13, ktlint 14.0.1, detekt 1.23.8, Dokka 2.1.0, Jacoco

Table 5.2

5.6.2 Backend services

Service	Stack
Auth	Go 1.25.0, `net/http`, `golang-jwt/v5` 5.2.1, `pgx/v5` 5.5.5, `x/crypto` 0.22.0, `x/time/rate` 0.15.0
Signaling	Go 1.22.0, `gorilla/websocket` 1.5.3, in-memory queue and room map behind a mutex

Table 5.3

Both services are static binaries with graceful shutdown. The auth service also exposes a `/health` endpoint that checks database reachability.

5.7 Build and CI

The game client builds with Gradle and compiler warnings as errors. CI runs the Kotlin tests and `./gradlew build`, builds the C++ channel, and builds the Go signaling service. The current CI gaps are backend unit tests, auth-service build coverage, and CTest registration for the native channel harnesses.

The backend stack is run through Docker Compose: PostgreSQL, auth service, and signaling service.

6 Validation

Validation follows the code structure. Deterministic game logic is tested automatically; adapters are tested at their boundary; behaviour that depends on real rendering, live services, WebRTC, or NATs is kept as manual acceptance testing.

Automated evidence for this report revision: on **2026-06-26**, from repository base revision `5cad47a` with report edits in the working tree, `cd game && ./gradlew test` ran **86 tests** with **0 failures** and **0 skipped tests**. The generated XML reports are under `game/build/test-results/test/`.

6.1 Automated Tests

The Kotlin suite contains 84 project-specific checks plus 2 placeholder examples in `MainTest`. The useful checks fall into four groups.

Group	Specs	What they validate
Domain combat	`InputSystemSpec`, `PlayerStateSpec`, `PhysicsSystemSpec`, `AttackSystemSpec`, `CrouchSpec`, `FacingSpec`, `HitDetectionSystemSpec`, `RectangleSpec`	Movement, jump/crouch, attack phases, hitbox overlap, damage, hitstun, knockback, and bounds
Determinism	`DeterminismSpec`, `WorldHashSpec`	Repeated runs over the same inputs produce identical states and comparable committed-frame hashes
Rollback	`RollbackServiceSpec`	Zero-delay correctness, convergence under delayed input, bounded frame

		advantage, stalls, and desync events
Adapters/data	`NetworkAdapterSpec`, `JsonCharacterRepositorySpec`	Packet queueing, peer-frame tracking, committed-hash dedupe, and character JSON loading

Table 6.1

The C++ channel directory also contains `protocol_test.cpp` and `local_test.cpp`, which create two libdatachannel peers in one process and open a data channel. These harnesses are useful, but they do not by themselves prove the Kotlin JNI wrapper, the signaling server, or the gameplay packet format.

6.2 Build Gates

The repository CI runs the Kotlin tests and `./gradlew build` for the game client, builds the C++ channel, and builds the Go signaling service. The Gradle build enables JDK 17, warnings-as-errors, ktlint, detekt, Kotest, MockK, Jacoco, and Dokka.

The current gaps are explicit: there are no Go unit-test files, the auth service is not built in CI, and the C++ harness executables are not registered as CTest tests yet.

6.3 Manual Acceptance

Manual checks are required for the parts that need a real environment. Before submission, each run should record the date, commit hash, OSes, network setup, commands, pass/fail result, and a short screenshot or log excerpt.

Area	**Manual scenario**	**Why manual**
Auth	Start Docker Compose, register a user, log in, reject bad credentials and duplicate registration	Depends on live PostgreSQL and the Go service

Matchmaking	Launch three clients; first two match while the third waits	Requires multiple live clients and signaling room state
NAT traversal	Run two clients on different networks and confirm a WebRTC data channel opens	Depends on real NAT behaviour and public STUN availability
Combat feel	Land hits in the GUI and inspect hitstun, knockback, and active hitboxes	The logic is tested, but readability is visual
Rounds	Win by KO and by timer expiry; confirm round reset and match end	Round rules do not yet have a dedicated spec
Rendering	Resize the window and inspect stage scaling, HUD, timer, and hitboxes	Compose rendering is visual
Queue cancel	Queue without an opponent, keep practice responsive, then cancel	Depends on GUI events and coroutine cancellation

Table 6.2

The automation gaps are also listed in Future Work: Go service tests, a round/timer rule spec, CTest registration for the native harnesses, and a JNI packet-format integration test.

7 Deployment

UFG has two deployable parts. The backend stack is containerized with Docker Compose. The desktop client is built locally because it links a platform-specific WebRTC library through JNI.

7.1 Prerequisites

Tool	Needed for
Docker + Compose	PostgreSQL, auth service, signaling service
JDK 17+	Game client
CMake 3.16+	Native WebRTC bridge
libdatachannel + OpenSSL	Native bridge dependencies
Go 1.22+	Optional service runs outside Docker

Table 7.1

Gradle is provided by the repository wrapper.

7.2 Backend Stack

Create a local `.env` file before starting the backend:

```
POSTGRES_PASSWORD=<choose-a-strong-password>
JWT_SECRET=<long-random-string>
```

Start all backend services with:

Expected result: `docker compose ps` shows Postgres, auth, and signaling running; `GET http://localhost:8081/health` returns `200` once Postgres is ready. Stop the stack with:

```
make down
```

Published ports:

Service	Host port
Signaling	`8080`
Auth service	`8081`
PostgreSQL	`5432` in the development override

Table 7.2

The Compose stack uses a named Postgres volume, a private bridge network, environment-based secrets, health checks, and `restart: unless-stopped`.

7.3 Game Client

Build the native WebRTC bridge:

```
make channel
```

On Windows, pass dependency locations if they are not installed at the default paths:

```
make channel LIBDATACHANNEL_PREFIX=path/to/libdatachannel OPENSSL_PREFIX=path/
```

Build and run the client:

7.4 Running a Local Match

Use three terminals:

```
make signaling   # backend stack
make game        # player one
make game        # player two
```

The first queued client becomes offerer, the second becomes answerer. They exchange SDP/ICE through signaling, open the direct WebRTC data channel, and start the match.

The current client build uses localhost backend URLs: `http://localhost:8081` and `ws://localhost:8080/ws`. Running against a backend on another machine therefore requires rebuilding or configuring the client with shared auth and signaling URLs.

The next chapter switches from operator instructions to the player's view of the application.

8 User Guide

This guide describes how a player uses UFG, from launching the client to winning a match. It assumes the backend stack and the client are already running (see the deployment chapter).

8.1 Getting Started

The client opens on the **authentication** screen (Figure 8.1). From there the flow is linear:

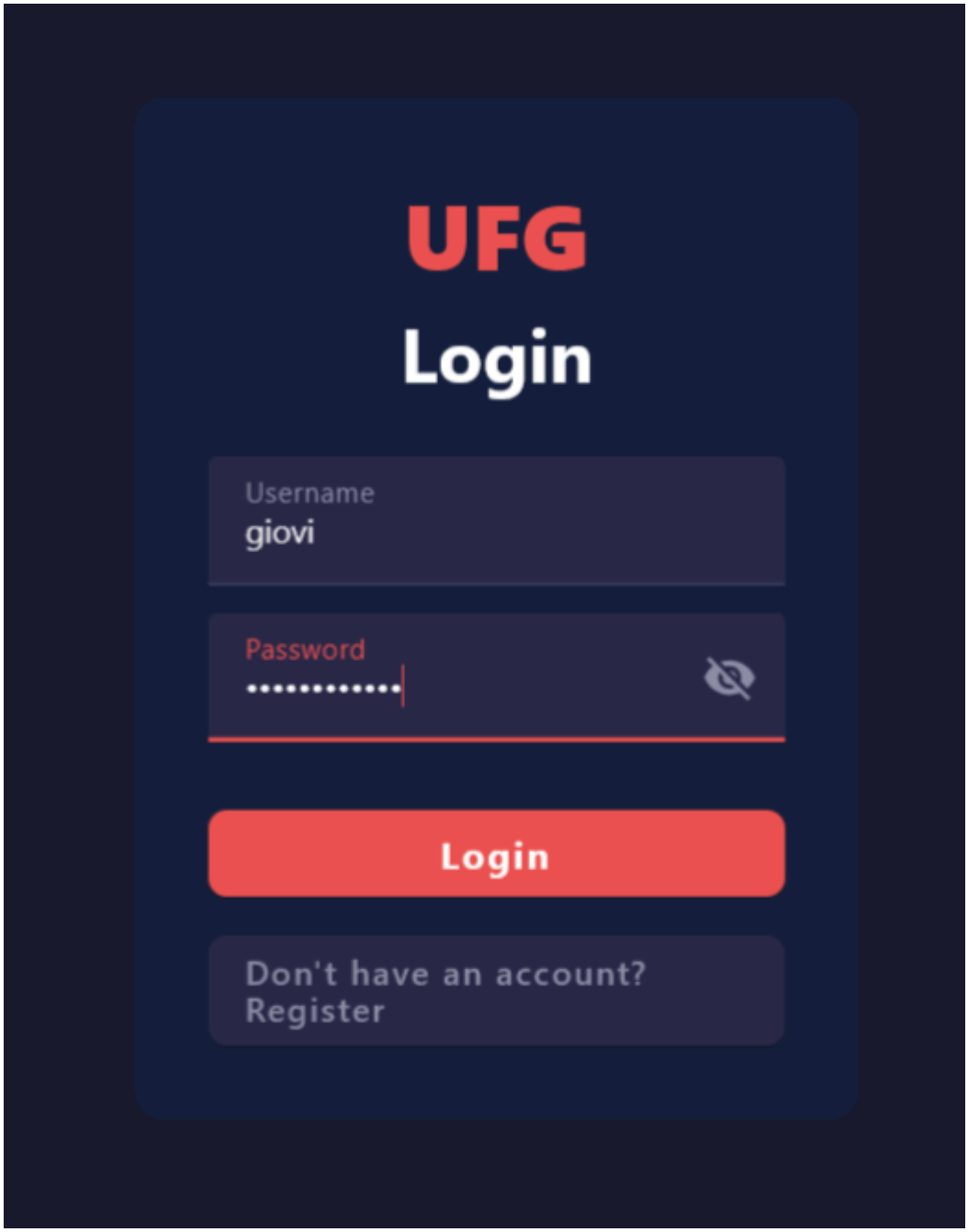

Figure 8.1: Login screen: register a new account or log in with an existing one.

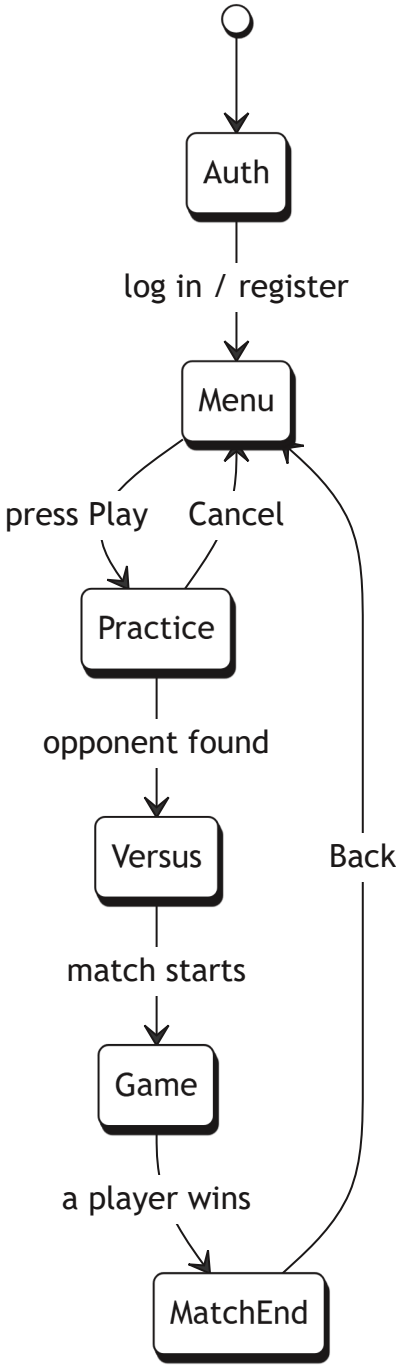

Figure 8.2: Screens the player moves through.

1. **Log in or register.** Use **Register** to create an account with a new username, or switch back to **Login** for an existing account. Bad credentials are rejected.
2. **Press Play.** The client joins the matchmaking queue.
3. **Wait in Practice.** While the queue looks for an opponent, a local practice stage keeps the controls live so the wait is never a frozen screen. Pressing **Cancel** leaves the queue and returns to the menu.

4. **Fight.** When an opponent is found, a brief *VS* splash shows both names and the match begins on opposite sides of the stage.

8.2 Controls

The menus use the mouse; the fight uses the keyboard.

Key	Action
`A` / `D`	Move left / right
`W` / `S`	Up / crouch
`Space`	Jump (no double jump)
`P`	Punch
`K`	Kick

Table 8.1

8.3 During a Match

- **Move** with `A`/`D` to approach or retreat, and **jump** with `Space`; you cannot jump again until you land.
- **Attack** with `P` or `K`. Punches are generally faster and lighter; kicks are slower but hit harder. Holding `S` while attacking selects the crouching variant when the current fighter defines one.
- **Time the active frames.** Every move has startup, active, and recovery phases loaded from the fighter's character data. During startup and recovery the attack cannot hit; during active frames the hitbox can connect.
- **Landing a hit** deals the move's configured damage and puts the opponent in **hitstun**, pushing them back with **knockback**. A single attack can only connect once per swing.

8.4 Winning

A round ends when either a player's HP reaches **0** or the **99-second timer** runs out, in which case the player with more HP takes the round. The first player to win **two rounds** wins the match, after which the match-end screen offers a return to the menu.

8.5 Reading the Screen

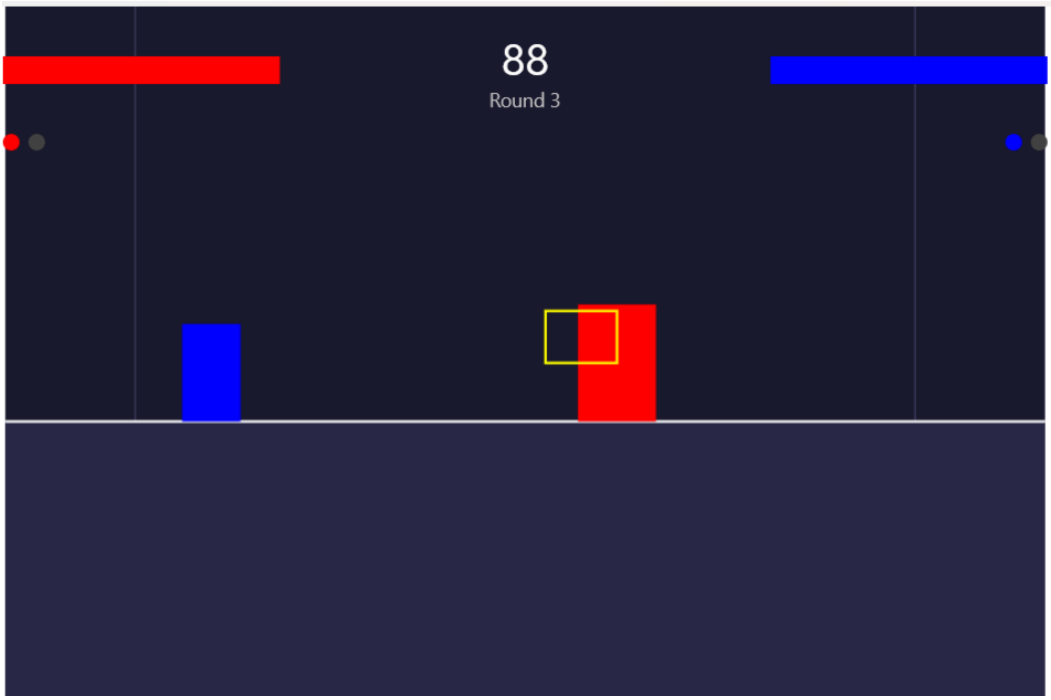

Figure 8.3: In-match screen: fighters, health bars, round pips, the timer, and an active hitbox outline.

The match screen (Figure 8.3) shows everything the player needs at a glance.

Element	Where	Meaning
Fighters	On the stage	Player 1 is the **blue** fighter, Player 2 the **red** fighter
Health bars	Top corners	Player 1 top-left, Player 2 top-right; the fill shrinks with damage

Round pips	Beside each health bar	One pip per round already won (first to two)
Timer	Top centre	Counts down from 99
Yellow outline	On a fighter	An attack's hitbox, shown only during its active frames

Table 8.2

The whole 800×600 stage scales uniformly to the window, so resizing keeps the play area proportional and simply adds letterboxing if the aspect ratio differs.

This completes the run-time view of the current system. The following chapter steps back from the software to each author's assessment of their own contribution.

9 Self-Evaluation

9.1 Paradisi Giovanni

My role covered a large part of the game itself, both its logic and the screens the player moves through, together with the authentication service. Mounir and I designed the entity-component-system together at the start, and from there I took ownership of the simulation systems and built most of what the player actually interacts with. In practice my main contributions were the physics, attack, and hit-detection systems, the engine and game-loop wiring that runs them at a fixed timestep, the round and match logic, the client screen flow (menu, matchmaking wait, practice, and match-end screens) with its login form, and the authentication service in Go, along with the tests for the systems I built.

Tying those systems into something playable meant building our own small game engine and loop instead of reaching for an existing framework. This was a deliberate choice: the determinism the netcode depends on is far easier to guarantee when we control every step of the simulation ourselves.

I think the strongest part of my work is that the game logic is deterministic and tested where it matters. Each system is a pure `World -> World` function, so I could exercise the physics, attack, and hit-detection systems directly with Kotest, without a window or a network. That coverage was not only a quality gate: it gave us the confidence that the simulation was deterministic enough for Mounir's rollback netcode to rewind and replay it correctly. Hit detection in particular had to iterate players in a stable order and register a hit only once per activation, and the specs are what kept that behaviour honest.

The product is not production grade. I chose to put testing effort into the domain systems and left the Go auth service and the round logic to manual verification rather than CI. That was a deliberate trade-off given where we chose to spend our effort, but still a gap I would close. More broadly the combat is shallow compared to a real fighting game.

The hardest technical part for me was getting determinism exactly right. Small choices that look harmless, such as iteration order over a map, where a value is

clamped, or when a frame counter advances, turn into correctness bugs the moment the same frame is replayed during a rollback, and the only reliable way to catch them was to keep the systems pure and pin their behaviour with tests. Keeping the Compose screens in step with a domain that was still changing underneath them was a smaller but constant second effort.

The way we split the work suited me well. Because I owned the game client and the auth service while Mounir owned the infrastructure and the netcode, we could usually work in parallel without stepping on each other's code.

With more time the first things I would add are automated tests for the auth service and a round-logic spec, and then deeper combat, since the deterministic foundation is already there to support it. Overall my contribution spanned most of the playable game: the deterministic simulation, the screens the player moves through, and the round logic, plus the authentication service that gets players into a match.

9.2 Samite Mounir

My main role was to make the game playable between two peers and to help keep the project structure under control. Since I had more experience with CI, Docker, and software process, I worked on the parts that connected the game code to a reproducible development workflow. In practice, my main contributions were the CI setup, the Docker configuration, the WebRTC bridge, the rollback netcode, and the matchmaking flow. Each group member was responsible for testing and documenting the parts they worked on.

I think the strongest part of the product is its structure. The game logic is kept deterministic and separated from infrastructure, which made it possible to add advanced networking ideas such as rollback netcode without rewriting the whole game. The architecture also makes the project easier to extend, because the networking, rendering, and domain logic have clear boundaries.

The product is still not production grade. The UI is functional but not complete, because visual polish was not the main goal of the project. The system has also not been deployed online, so we could not perform strong real-world benchmarking. For this reason, the validation is strongest around local automated tests and weaker around live network performance.

The hardest technical part for me was understanding the peer-to-peer connection flow and making the game experience feel responsive. Rollback netcode was also challenging, because it only works if the simulation is deterministic and if old states can be restored correctly. Open-source libraries, documentation, and AI tools helped during this work, but the final behaviour still had to be integrated and checked inside our own codebase.

At the beginning Giovanni and I worked together closely to understand the game structure and decide how the main systems should fit together. Later, when the distributed parts became clearer, we divided the tasks more explicitly. We used a pull-request based workflow on GitHub, which worked well and avoided major merge conflicts. One practical difficulty was making complex configuration work on Windows; the Windows CI was the slowest and hardest pipeline to make pass, probably because it's a sloppy OS.

If I had more time, I would start the distributed part earlier and keep a more regular development pace. We had some long pauses during the project, and each pause cost time because we had to remember previous decisions before continuing. At the same time, spending time on the game structure was necessary, because rollback netcode depends on having a deterministic game in the first place.

Overall, my contribution was mainly architectural and infrastructural. I helped set up a clean project structure, divided work in a way that kept the team productive, and implemented the peer-to-peer parts that made the game playable between two machines.

10 Future Work

The current implementation demonstrates the core approach: deterministic local simulation, rollback, and committed-frame hash checks across two peers. The next steps are about making the game deeper, nicer to use, and easier to validate.

10.1 Gameplay Features

Feature	What it adds	Implementation sketch
Character selection	Players choose fighters instead of using the hardcoded rushdown/heavy matchup.	Add a select screen and exchange both picks before the match starts.
Blocking	A defensive option that reduces damage or converts hitstun into shorter blockstun.	Use opponent-relative facing and the existing `BLOCKSTUN` placeholder.
Expanded move list	Heavy, aerial, and additional directional attacks.	Add more `Attack` entries to each JSON character; the current attack pipeline already consumes frame data.

Table 10.1

Because character data affects simulation, peers should also compare a roster hash during setup and refuse to start if their local character definitions differ.

10.2 Presentation

The game is currently readable but visually minimal. Presentation can improve without changing deterministic game logic because rendering is derived from the finished `World`.

- Replace colored rectangles with per-state sprites.
- Add sound effects for hits, jumps, round transitions, and menu actions.
- Add impact feedback such as flash, shake, hitstun coloring, and stage art.
- Smooth visible rollback corrections by interpolating rendered positions for a few render frames while keeping simulation state unchanged.

10.3 Offline Training Bot

Practice mode already runs locally while matchmaking, but the opponent is idle. A training bot would make offline practice useful. It can be implemented as a second input source that reads the current `World` and returns an `InputState` each frame, leaving the online rollback path untouched.

10.4 Accounts and Sessions

The JWT issued at login is currently obtained, decoded client-side for the user's identity, and then unused: no request carries it and no service verifies it after issuance, so every session starts with a fresh username and password. The token is designed to enable two follow-ups. The first is **persistent login**: store the token and, on launch, skip the login screen straight to the menu while it is still valid — the navigation hook for this is already stubbed in `ScreenManager`. The second is **token-authorized matchmaking**: send the token to the signaling server so it admits a player only after verifying it, instead of trusting whoever connects. The server-side verifier (HMAC-only, rejecting `alg:none`) already exists for exactly this; making either feature trustworthy means verifying the token's signature rather than only decoding it.

10.5 Validation and Tooling

Item	Why it matters	Implementation sketch
Go service unit tests	Auth and signaling are currently covered mainly by build/manual checks.	Add handler tests for auth and WebSocket tests for queueing, relay, cancel, and disconnect cleanup.

Round/timer rule spec	Timer, KO, round reset, and match end lack dedicated automated coverage.	Test `GameLogic.step` fixtures for timeout, KO, round counter, reset, and best-of completion.
JNI packet test	Kotlin tests cover adapter queues, but not one real native packet through the wrapper.	Send one gameplay packet through `WebRtcBridge.sendInput` and assert all fields survive unchanged.
CTest registration	The native harnesses are built but not registered as CTest tests.	Add `enable_testing()` and `add_test(...)` for `local_test` and `protocol_test`.
Manual evidence log	NAT traversal, rendering, GUI cancellation, and live auth need repeatable proof.	Store commit, date, OSes, network setup, commands, logs/screenshots, and pass/fail result.
Backend CI coverage	CI builds signaling but not auth, and runs no Go tests.	Add `go test ./...` jobs for both Go modules and a Docker Compose auth `/health` smoke test.

Table 10.2

Together, these items turn the prototype evidence into a repeatable release process without changing the architecture.