SYSTEM ARCHITECTURE

GameFi Rails

The financial operating system for Web3 games - embedded wallets, a hybrid exchange with a Rust matching engine, an NFT marketplace, treasury, risk controls and economy analytics, delivered to game studios as one SDK.

Repository	E:\projects\gamefi-os (pnpm workspace + Cargo workspace, zero-build static UI)
Runtime	Node 24 / TypeScript ESM / Fastify 5 + Rust (matching engine) + Redpanda (Kafka API)
Data	Neon Postgres (Prisma 7 + adapter-pg), double-entry ledger as source of truth, RLS multi-tenancy
Verification	172 TypeScript + 19 Rust tests, property tests, chaos suite (kill -9), continuous reconciliation
Status	Feature-complete, runs locally via one command (pnpm dev). Hosting deliberately deferred on cost.

1. What the system guarantees

The architecture is organised around four claims a technical buyer can verify, not around features:

- **Crash safety:** kill -9 the matching engine mid-session and it restarts to a bit-identical order book (snapshot + deterministic replay, asserted by checksum). Automated in the chaos suite.
- **Money integrity:** every movement of value is a double-entry ledger transaction that sums to zero per asset; a continuous job asserts the invariant, so a wrong balance cannot hide.
- **Zero-friction UX:** a playable game (Dragon Arena) proves the loop - earn a token, loot an NFT, sell it, cash out on a real order book - with no wallet popups, seed phrases or gas.
- **Measured performance:** 97 ns order placement, 13M commands/sec sustained (criterion benchmarks on the real engine, 10k resting orders).

2. System topology

Ingress to the matching engine is **the log, not HTTP**. The API validates and reserves funds, then appends a command to a transactional outbox in the same Postgres transaction; a relay produces it to Redpanda; the Rust engine consumes, matches and emits events; independent consumers settle, fan out and police those events. The engine never holds balances and never talks to users.

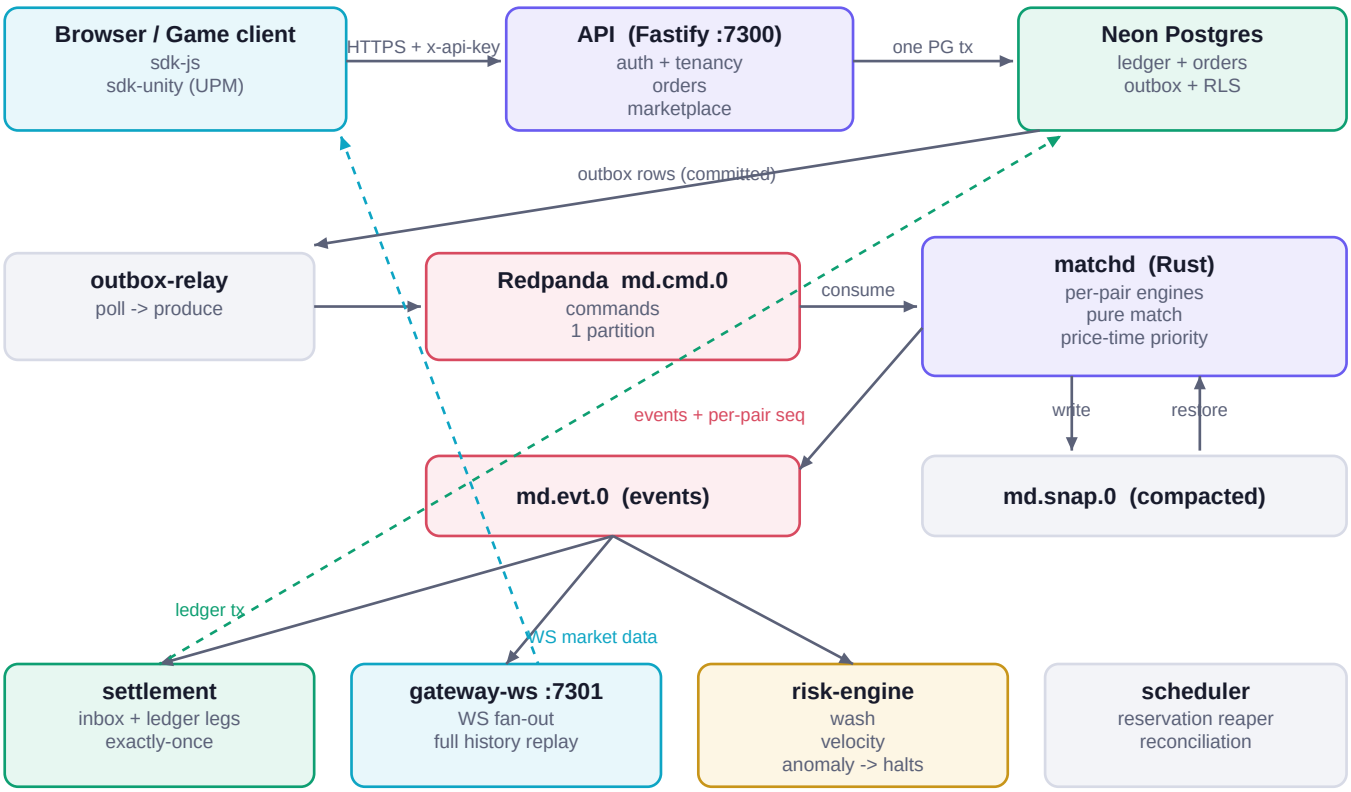

Figure 1 - runtime topology. Solid arrows are the synchronous order path; dashed arrows are continuous side-channels (settlement writes, WS fan-out, reconciliation).

3. Services and shared packages

Seven runnable services and four Rust crates share sixteen TypeScript packages. Everything money-critical lives in packages so every service computes with identical rules.

Service	Role	Key property
api (:7300)	Fastify edge: auth, orders, wallets, marketplace, treasury, admin. Serves the static UI.	Order placement = one PG tx: reserve + hold + outbox
outbox-relay	Polls committed outbox rows, produces to md.cmd.0.	Resend-never-lose
matchd (Rust)	Per-pair matching engines fed by the command log; emits events with a per-pair seq; snapshots.	Pure function of (state, log); bit-identical replay
settlement	Consumes md.evt.0; posts trade legs, fees, revshare; consumes/releases reservations.	Exactly-once via inbox PK (shardId, pairId, seq)
gateway-ws (:7301)	WebSocket fan-out of engine events to browsers.	Replays full history on connect
risk-engine	Wash trading, velocity, price anomaly on the stream; trips circuit breakers (pair halts).	Ignores events older than 2x its window on restart
scheduler	Reservation reaper + continuous reconciliation.	Zero-sum + holds-vs-reservations invariants

Packages (TS)	money (market registry - THE source for assets/pairs), ledger (post primitive, builders, invariants), db (Prisma + RLS), proto (wire types + fixtures), kafka, outbox, chain (EVM providers), vault (envelope encryption), policy (withdrawal rules), auth, sdk-core / sdk-js / sdk-unity, config, observability, testing
Crates (Rust)	matching-core (pure book + matcher), matching-proto (wire + conformance fixtures), matchd (daemon), matching-bench (criterion)

4. The life of an order

Funds are locked **before** the engine can possibly see the command. An order acknowledged at HTTP but not yet produced to Kafka never existed; the client holds a PENDING order that the reaper releases if no engine ack arrives.

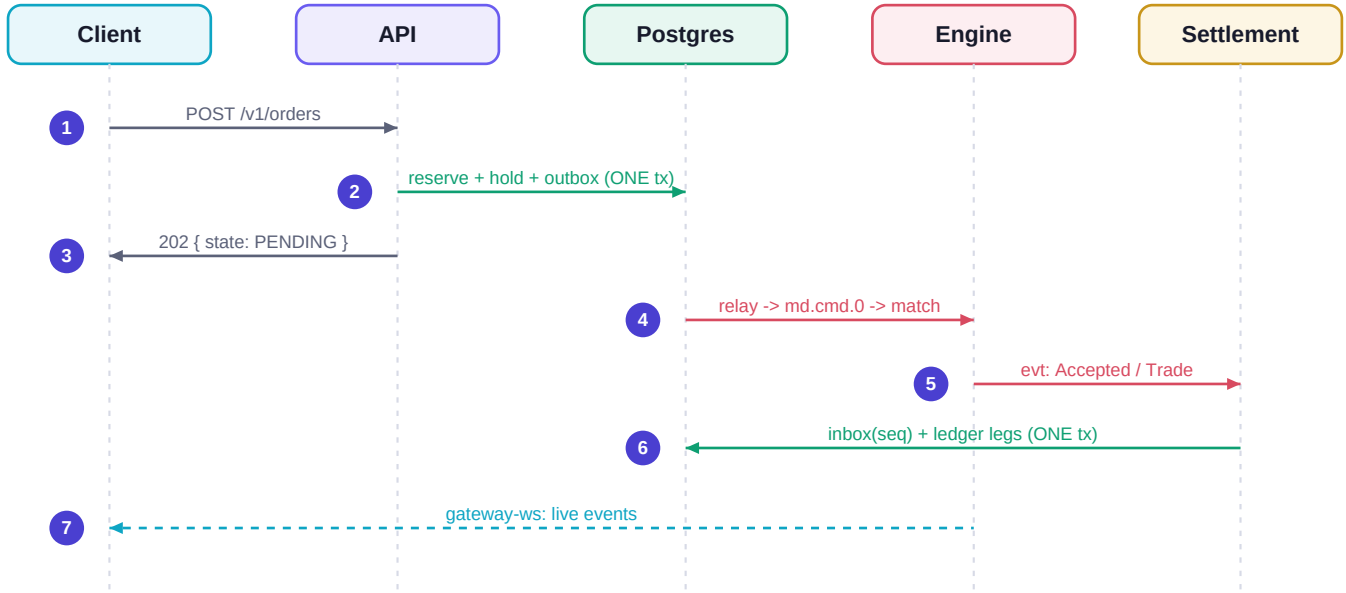

Figure 2 - order lifecycle. Steps 2 and 6 are single Postgres transactions; that is where every guarantee is anchored.

- **Reservation protocol:** validate pair rules, compute required amount, insert a reservation row, post a ledger hold (available -> hold), and insert the NewOrder command into the outbox - one commit. Fill / cancel / reject all release the residual under one shared idempotency key (reserve-release:{clientOrderId}), so whichever fires first wins and the rest are no-ops.
- **Backstop:** a reaper releases any un-acknowledged reservation after a timeout - gated on acknowledgedAt (stamped by settlement on OrderAccepted), because a resting order legitimately holds funds forever. The run is skipped entirely while the outbox has a backlog.

5. Money - the double-entry ledger

The ledger is the single source of truth; balances are cached projections. Entries are BigInt minor units (no floats anywhere), signed so that every (transaction, asset) group sums to exactly zero. Account handles follow `{scope}:{id}:{purpose}:{asset}` over scopes system | platform | tenant | user | chain, and user accounts are credit-normal (holdings stored negative), which makes an overdraft structurally visible as a positive balance.

Worked example - taker buys 10 DRAGON @ 2.50 USDC (taker 30 bps, maker 10 bps, 40% studio revshare), idempotency key `trade:{shard}:{pair}:{seq}`:

Account	Amount (USDC)	Meaning
user:TAKER:hold:USDC	+25.075	buyer hold consumed (price + taker fee)
user:MAKER:available:USDC	-25.000	seller receives proceeds
user:MAKER:available:USDC	+0.025	maker fee docked
tenant:STUDIO:fee:USDC	-0.040	studio revenue share (40%)
platform:fee:USDC	-0.060	platform share (60%)
SUM (quote leg)	0.000	enforced per (transaction, asset)
user:MAKER:hold:DRAGON	+10.000	seller hold consumed
user:TAKER:available:DRAGON	-10.000	buyer receives base

Account	Amount (USDC)	Meaning
SUM (base leg)	0.000	enforced per (transaction, asset)

Debit positive / credit negative. NFTs are ledger assets with decimals=0, supply=1 - so a marketplace sale reuses this exact settlement path with a royalty entry added, rather than a parallel money system.

Exactly-once settlement - three stacked guards

- **Inbox:** settlement inserts (shardId, pairId, seq) into an inbox table in the same transaction as the ledger entries. Redelivery hits the primary key and rolls back as a no-op.
- **Deterministic idempotency keys:** trade:{shard}:{pair}:{seq}, deposit:{chain}:{txHash}:{logIndex} - never derived from a clock or a fresh UUID - as defense in depth on the ledger itself.
- **Deterministic re-emission:** after an engine restart the replayed commands produce the same events with the same seq, so duplicates are absorbed rather than prevented. This replaces Kafka transactions with something simpler and testable.
- **Never overdraft:** a trade arriving with an insufficient reservation (a platform bug) posts against system:suspense:shortfall, emits an anomaly and halts the pair. Books stay balanced; a human reconciles.

6. Order state machine

API writes PENDING only. Every later state is written by settlement from engine events - the order table is a projection of the log, never a second source of truth.

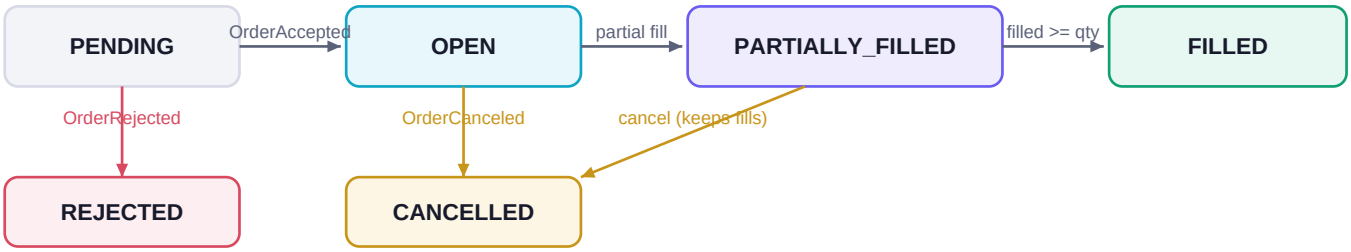

Figure 3 - FILLED is derived from filled >= qty, so a partial fill followed by a cancel keeps its fills. Cancelling enqueues a CancelOrder command through the same outbox; CANCELLED is only written on the engine's event, which is also what releases the hold.

7. The matching engine (Rust)

matching-core is pure and synchronous - no I/O, no clocks, no randomness. The daemon (matchd) wraps it with Kafka consumption and snapshotting. One process owns a shard; a shard owns N pairs; a pair belongs to exactly one shard and one Engine instance with its own seq counter - there are no cross-pair orders, ever.

- **Order book:** per side, a BTreeMap of price levels (bids keyed inverted so both sides iterate best-first) over a slab of orders with an intrusive FIFO per level and an id map for O(1) cancel. Price-time priority falls out of the structure instead of a comparator.
- **Order types:** Limit, Market, IOC, FOK, PostOnly, with self-trade prevention (cancels the resting order) and a reservation ceiling the engine respects per order.
- **Determinism:** the Kafka log assigns command order; the engine assigns a per-pair seq to every event. Restart = load newest snapshot, seek to snapshot.lastAppliedOffset + 1, replay. Matching being a pure function makes the replayed state bit-identical - asserted by a checksum in the snapshot and by a proptest equivalence suite.
- **Wire protocol:** MessagePack, never JSON - bigints survive; u128/uuid as 16-byte big-endian bins. The schema is authored twice (serde in Rust, zod in TS) with a checked-in fixture corpus encoded in Rust, decoded in TS and back, byte-compared in CI. Version field, unknown versions rejected.

Benchmark (criterion, 10k resting orders, single core)	Result
Resting limit order placement	97 ns
Sustained command throughput (in-process)	13,000,000 / sec
Honest end-to-end p99 (through Redpanda + settlement)	milliseconds - bounded by relay poll + serial settlement, not the engine

8. Multi-tenancy and security

Shared schema with a tenantId column - not database-per-tenant - because a trade between two studios' players touches both tenants' fee accounts in one atomic transaction; splitting databases would turn the ledger into a two-phase commit. Isolation is enforced in three independent layers:

Layer 1 - Edge
resolveTenant(): x-api-key -> sha256 -> tenant. Missing key = 401. PUBLIC_ROUTES allowlist with justifications; a CI guard parses main.ts and fails when a new route is neither authenticated nor allowlisted.

Layer 2 - Service
assertPlayerInTenant(): cross-tenant ids return NOT_FOUND (not FORBIDDEN) so ids cannot be probed. Sandbox tenant gets a real scoped key from /v1/sandbox/session, rotated each boot.

Layer 3 - Database
Postgres RLS: ENABLE + FORCE on tenant tables; policy reads app.tenant_id GUC set with SET LOCAL inside the transaction (pooler-safe). Missing GUC = zero rows. A DMMF-generated leakage test covers every model.

Figure 4 - defense in depth. Any single layer failing still leaves tenant isolation intact.

- **Player identity is global** with per-tenant profiles; studios see an HMAC-derived tenantPlayerRef rather than the global id, so two colluding tenants cannot join their datasets.
- **API keys** are stored as sha256 only and shown once at mint. On Neon, BYPASSRLS is not grantable, so the platform service role uses explicit allow-all policies instead - same effect, auditable.

9. Marketplace, risk and treasury

Marketplace - NFTs as ledger assets

Mint, list and buy are atomic ledger operations. A sale is a trade with extra legs: 5% creator royalty and a 2% marketplace fee split 50/50 with the studio, all enforced at settlement rather than by convention. Loot rarity is baked into the minted asset name, so it survives into listings without a metadata side-channel.

Risk engine - detection wired to enforcement

- **WASH_TRADING:** counts round-trips between counterparty pairs inside a rolling window; a simulated 3-round-trip ring auto-halts its pair in seconds (verified live).
- **VELOCITY** (order-rate abuse) and **PRICE_ANOMALY** (trades far from the rolling reference) trip the same breaker path.
- **Enforcement:** a tripped breaker writes a PairHalt row; the API refuses new orders on that pair (PAIR_HALTED) until an authenticated, audited release via /v1/admin/unhalt.

Treasury - the money-out path

- **Policy engine:** withdrawal requests are evaluated against real 24h + hourly velocity from history; above the approval threshold ($10k) they require 2 distinct approvers.
- **Deterministic signing:** a nonce derived from withdrawalId is persisted before signing, so a retry produces a byte-identical transaction - a replay is a no-op at the RPC, never a second payment.
- **Custody:** envelope encryption (AES-256-GCM) for keys, hot/warm/cold segregation, testnet-only until externally reviewed (Base Sepolia + Polygon Amoy via viem).

10. Operations and verification

Mechanism	What it proves
pnpm dev	One command boots the whole stack in dependency order: Redpanda (Docker) -> matchd -> relay / settlement / gateway / risk / scheduler -> API. The matchd PAIRS env is generated from the market registry.
pnpm chaos	Kill -9 matchd mid-session: a pre-crash order still matches after restart. Kill -9 settlement mid-stream: the backlog settles exactly once. Ledger reconciles clean afterwards.
Reconciliation job	Continuously asserts: per-asset zero-sum, cached balances match entry sums, ledger holds equal open reservations + listings, and names any orphans.
Conformance corpus	Rust <-> TypeScript wire fixtures byte-compared both directions in CI.
Route-coverage guard	A test parses main.ts and fails CI when a route is neither authenticated nor explicitly allowlisted with a justification.
Test suite	172 TypeScript tests (Vitest, PGlite for DB tests) + 19 Rust tests + property tests (no crossed book, price-time priority, quantity conservation, snapshot equivalence).

11. Deployment shape

The UI is zero-build static files served by the API, so the front-end can live on any static host or CDN. The API is request/response and could run serverless against Neon. Everything else is **always-on** by nature - matchd, Redpanda, settlement, gateway-ws, risk-engine and the scheduler are long-running consumers and cannot run on function platforms (e.g. Vercel); they need a container host or VPS. Currently everything runs locally; hosting is deferred on cost. Dockerfiles and deploy notes exist in-repo (Dockerfile.node, Dockerfile.matchd, DEPLOY.md).

Component	Static host / CDN	Serverless	Always-on host
Static UI (5 pages, app.css/js)	yes	-	yes
API (Fastify)	-	possible (Neon-backed)	yes (current shape)

Component	Static host / CDN	Serverless	Always-on host
gateway-ws (WebSockets)	-	no	required
matchd + Redpanda	-	no	required
settlement / risk / scheduler / relay	-	no	required

Open items: funded Base Sepolia wallet for the on-chain broadcast leg; Neon password rotation; hosting decision; end-to-end load test (the relay poll interval and serial settlement bound real throughput well below the engine's 13M/s).