

Specification — Stellar integration into AIDApp

High-level statement.

The system is modular, scalable and blockchain-agnostic. It uses a layered architecture where each layer has a single responsibility and communicates via queues. Redis is used for hot caching and queue coordination; PostgreSQL is the durable store. This design allows adding new blockchains, processing large data volumes, and enabling new features without disrupting existing components.

Layered architecture (summary)

1. **Blockchain Layer** — blockchain-specific clients + DEX adapters + verifiers.
2. **Scanner Layer** — block scanners & interval scanners that pull raw data and push normalized events into queues.
3. **Data Processing Layer** — workers that transform events, update Postgres/Redis, and prepare hot caches.
4. **Trade Layer** — trade storage, aggregation, notification (WebSocket), charting & UI feeds.
5. **Token Audit** — automated token validation pipeline (multi-provider).
6. **Swap Logic** — swap processing, confirmation delay, fee/referral calc and distribution.

Between layers — **durable queues** (Redis streams / RabbitMQ / Kafka depending on scale) manage flow and backpressure.

Blockchain Layer — responsibilities & components

Purpose: isolate chain-specific logic and standardize outputs for the rest of the stack.

- **Basic Client** — pulls blocks, txs, latest block, and normalizes raw chain data to common schema (block, tx, op).

- **DEX Adapter** — parses exchange/AMM/orderbook events (liquidity changes, trade logs), normalizes to trade / pool / order events.
- **Verifier** — verifies cryptographic signatures, transaction authenticity, and supports multiple signature schemes (Stellar signatures, EVM, Tron, etc.).
- **Internal Swap Adapter (optional)** — inspects on-chain internal swaps for accurate fee allocation and referral reward validation.

Outputs: normalized events pushed to Scanner Layer queues (e.g., chain.<network>.blocks, dex.<network>.<dex>.events).

Scanner Layer

Continuously converts blockchain data into actionable events.

- **Block Scanner** — pulls ledger blocks sequentially; for each block: extract txs, ops, logs; produce events per DEX and general events (deposits/withdrawals).
- **Interval Scanner** — scheduled tasks for specific data (e.g., liquidity snapshots, pool health, historical fills) to enrich datasets or fill gaps.
- **Queueing** — per-DEX/per-network queues (e.g., q:stellar:dex:events) to allow independent scaling and prioritized processing.

Behavior: scanners ensure idempotency (track last processed ledger), checkpointing, and re-scan ability.

Data Processing Layer

Transforms scanner events into canonical records, updates storage, and prepares cache.

- **Transformation Workers** — map chain-specific events to internal schema (deposits, withdrawals, swaps, liquidity_events, launch_events).
- **Storage Writers** — write durable records to PostgreSQL and push hot keys to Redis for immediate access.
- **Cache Preparation** — build and maintain Redis hot caches used by the Trade Layer (top pairs, recent trades, candles).

- **Concurrency model** — each queue consumed by dedicated worker pool; workers use optimistic locking and atomic Redis/Lua scripts where needed.

Data models (examples):

- deposits(tx_hash, ledger, user_address, asset_code, issuer, amount, memo, metadata)
 - swaps(tx_hash, pair, amount_in, amount_out, fees, referral_id, timestamp)
 - launches(launch_id, token, curve_params, threshold, status, metadata)
-

Trade Layer (Trading Terminal)

Handles trade ingestion, aggregation, user notifications, and UI feeds.

- **Redis Trade Storage** — in-memory store for low-latency access; updated atomically with Lua scripts to prevent race conditions.
- **Postgres** — canonical store for historical trades and compliance checks.
- **Hourly Aggregates** — worker that recalculates hourly aggregates for analytics; design ensures no double counting after re-scans.
- **Notification System** — event-based pushes to WebSocket/Push services for UI updates (new pair, new trade, price alerts).
- **AI/Analytics Feed** — processed trade data streamed to AI models for scanning and signals.

Impact: users see near real-time Stellar DEX data alongside other chains, with consistent UX and AI signals.

Token Audit

Automated integrity checks of token metadata and activity.

- **Selection & Queuing** — periodic selection of tokens for audit based on criteria (new tokens, suspicious activity).

- **Audit Workers** — call external audit providers / internal checks: metadata consistency, ownership verification, liquidity audits, rug-signal heuristics.
 - **Result Storage & UI** — audit results saved in Postgres and surfaced to UI (trust score, warnings).
 - **Extensibility** — new audit providers can be added as pluggable modules.
-

Swap Logic & Referral Rewards

Ensures correct processing of swaps, accurate fee calculation and reward distribution.

- **Initiation** — API hook receives swap request; backend creates a pending swap record.
 - **Delay / Confirmation Window** — apply configurable confirmation window (10 minutes as specified) to wait for on-chain finality and potential reorgs.
 - **Validation** — after delay, scanner/validator confirms tx on-chain and updates swap record.
 - **Fee & Referral Calculation** — compute platform fee, partner cuts, and referral rewards based on verified on-chain data.
 - **Distribution** — enqueue payouts; use batch payouts via Stellar (or chain-specific bridge) for efficiency.
 - **Notifications** — real-time status updates to user via WebSocket.
-

Queues, Storage & Performance

- **Queues:** Redis Streams or Kafka for high throughput and persistence. Segmented by network/DEX to allow per-network scaling.
- **Hot Cache:** Redis for live charts, recent trades, non-blocking counters.
- **Durable Store:** PostgreSQL for normalized records, user data, audits, reconciliation.
- **Sequence & Idempotency:** every scanner/worker tracks last processed ledger/tx; all writes are idempotent.

Where Soroban is used (role & scope)

Soroban contracts are introduced to extend Stellar's native primitives where custom on-chain logic is required:

- **DepositContract:** deposit gating, whitelist enforcement, memo metadata and event emission for LaunchZone flows. Backend listens to contract events.
- **WithdrawalContract:** handles batched withdrawals, review/replay protection, admin emergency flows.
- **TradingProxyContract:** advanced routing/fee logic and integration with Stellar DEX (multi-hop path payments), exposing optimized swap endpoints.
- **BondingCurveLaunchContract:** fair-launch logic (curve pricing, thresholds, liquidity locking) and immediate on-chain listing support.

Design principle: use native Stellar features for simple flows (trustline, path payment) and Soroban for composable, auditable, and upgradeable business logic that must run on-chain.

Adding a new network (process)

1. Implement **Basic Client** for the network (blocks/txs/latest).
2. Build **DEX Adapter** for that network's DEX logs/events.
3. Add **Verifier** to validate signatures/tx authenticity.
4. (Optional) Add **Internal Swap Adapter** to reconcile internal swaps and referral logic.
5. Add scanner configuration and queue mappings; deploy worker pools.

Because architecture is modular, adding networks does not change other layers.

Operational considerations & SLOs

- **Indexer Lag:** < 5s target for Stellar critical flows.

- **Transaction success rate:** $\geq 98\%$ for user swaps executed via TradingProxy.
 - **Throughput target:** baseline architecture supports horizontal scaling to meet peak loads; target stress tests up to 1,000 TPS for critical pipelines.
 - **Observability:** Prometheus/Grafana, alerting on horizon latency, queue backlog, failed tx rates.
-

Security & Compliance

- **Replay protection & nonces** per network.
 - **Admin controls:** multisig, time-locks, and audit logs for emergency operations.
 - **Contract audits:** external audit for Soroban contracts before mainnet deployment.
 - **Data privacy:** PII separation and encryption at rest for sensitive fields.
-

Example user flows (compact)

Bonding Curve Launch

User → LaunchZone UI → Backend validates & deploys BondingCurveLaunchContract → Soroban emits LaunchCreated → Indexer ingests → UI shows launch → Users buy via buy calls → Soroban executes buys, emits BuyExecuted → Indexer updates trades/launch state → TradingProxy integrates pool to DEX → Terminal shows pair.

Swap (Stellar)

User → AIDA Terminal (swap UI) → Backend Router constructs transaction or suggests multi-hop path → User signs → tx submitted to Horizon → BlockScanner catches tx → Swap validated → Fees/referral calculated → Redis/Postgres updated → WebSocket notifies user.