Enable ZK — Technical Documentation

Privacy-preserving humanitarian aid distribution on the Aleo blockchain, by HumanityLink.

First draft | For developers & integrators

humanity.link

CONTENTS

1. Introduction

1.1 What Enable ZK is

Enable ZK is a humanitarian cash-and-voucher assistance (CVA) system built by **HumanityLink**. Within the CVA spectrum it sits firmly on the *cash* side: NGOs distribute aid as **private, on-chain digital value** — real **USDCx** carrying full monetary value, not restricted vouchers — spendable at approved local merchants, transferable peer-to-peer, and convertible to cash. The system is built on the Aleo blockchain and uses **zero-knowledge proofs (ZKPs)** so that:

- **Beneficiary identities and balances remain private.** A beneficiary is a pseudonymous address holding encrypted on-chain records; only the NGO's *view key* can decrypt program data for reporting.
- **Fund flows remain auditable.** Aggregate totals (issued, spent, off-ramped, transferred) are public on-chain counters, so donors and auditors can verify money movement end-to-end without seeing who received what.

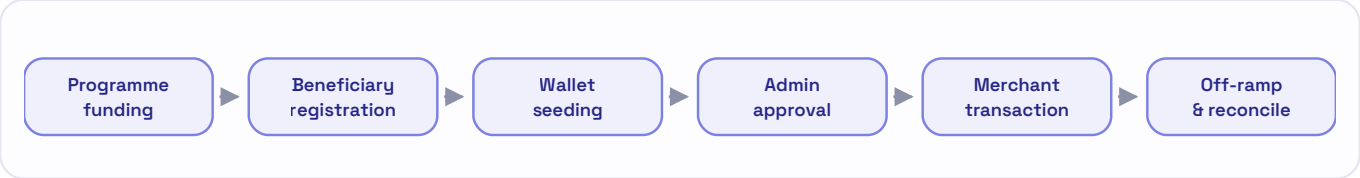
Aid is denominated in **USDCx** — a USDC-backed stablecoin on Aleo — and held as standard private stablecoin token records. There is no aid-specific token: aid *is* USDCx, which is what makes the off-ramp to real cash possible.

The user-facing experience is deliberately low-tech:

- A **beneficiary's wallet is a QR code**, delivered over WhatsApp or as a printed sticker. No app install, no seed phrases, no crypto knowledge required.
- A **merchant** runs a lightweight point-of-sale (POS) mobile app that scans the QR, submits payments, and lets the merchant cash out an accumulated balance through MoneyGram.
- The **NGO** operates a web dashboard for onboarding, issuance, whitelisting, monitoring, and program controls.

The guiding design principle: **simple for users, controlled behind the scenes**. All cryptographic and blockchain complexity is absorbed by a custodial backend; all programmatic controls (whitelisting, approval, pause, limits) are enforced on-chain.

1.2 End-to-end money flow



1. **On-ramp.** HumanityLink funds the program for the NGO: fiat → USDC (via an exchange, e.g. Kraken) → a per-NGO Ethereum vault contract → bridged via **Circle xReserve** → **USDCx on Aleo**, landing in the NGO's program pool.
2. **Issuance.** The NGO issues aid to beneficiaries: the program's public USDCx pool is converted into **private** USDCx token records owned by each beneficiary (single or bulk issuance).

3. **Spending.** Beneficiaries pay whitelisted merchants (goods purchase or cash-out) by showing their QR; the merchant's POS submits the transaction; the beneficiary approves the transaction on WhatsApp; a zero-knowledge proof is generated server-side and the private transfer settles on-chain.
4. **Circulation.** Value also moves laterally before it exits: beneficiaries send each other **P2P transfers** straight from WhatsApp (`PAY <HL-ID> <amount>` or `RECEIVE`, confirmed via a single-use link), and merchants can **buy goods from beneficiaries** (`pay_beneficiary`), paying out of their own USDCx — so aid money keeps working inside the local economy.
5. **Off-ramp.** Merchants accumulate USDCx and convert it to local cash: Aleo → Ethereum (Circle xReserve burn) → Stellar (Circle CCTP V2) → **MoneyGram SEP-24** cash pickup, with the reference number delivered over WhatsApp.

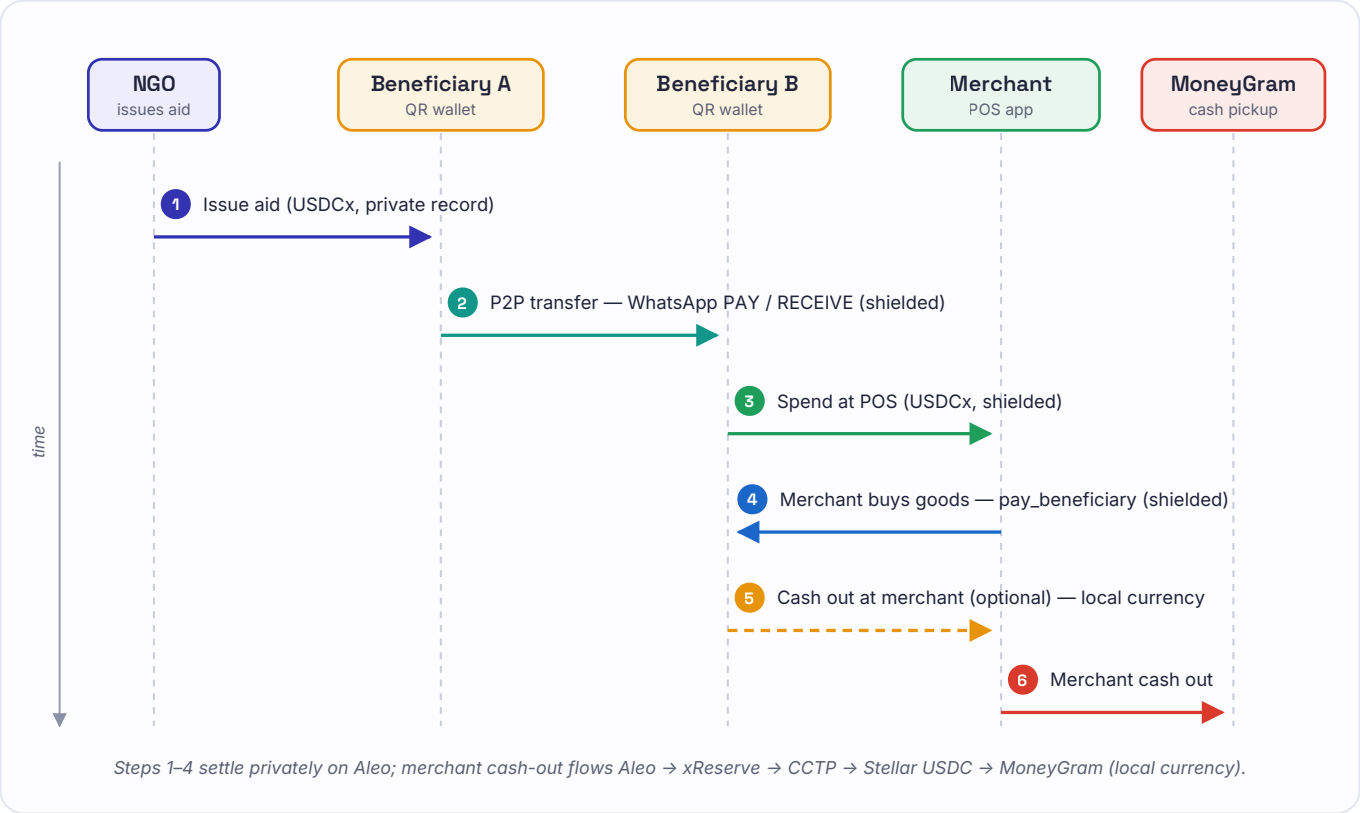


Figure 1 — Money flow. The lifecycle of one aid dollar across the actors. Steps 1–4 are private Aleo transitions (only aggregate counters are public): issuance, beneficiary-to-beneficiary P2P over WhatsApp, spending at the POS, and the merchant buying goods from a beneficiary out of its own USDCx. Steps 5–6 are the two distinct cash-out paths — a beneficiary converts at a merchant, a merchant converts through MoneyGram.

1.3 Scope of this document

This document is written for **developers and technical integrators**. It explains what each component does, how the pieces fit together, which techniques and technologies are used and *why*. It intentionally does **not** cover repository setup, local run commands, or per-environment `.env` contents — those live in each component's own README inside the repository.

2. Actors and roles

Enable ZK is designed around a strict separation of responsibilities. The system works when responsibilities are clear and permissions match the role.

Actor	What they do	Interface
Beneficiary	Receives a QR wallet, spends or cashes out at approved merchants, sends and receives **P2P transfers** with other beneficiaries, **sells goods to merchants** (paid from the merchant's own USDCx), checks balance, reports issues.	WhatsApp bot; QR code (digital or printed sticker)
Merchant	Registers, gets whitelisted, scans beneficiary QR, accepts payments, **buys goods from beneficiaries**, views balance, **downloads transaction history** (Excel report via WhatsApp), cashes out via MoneyGram.	Merchant POS app (Expo/React Native, iOS **and** Android)
NGO Admin	Runs the program. Two levels: a **basic admin** registers beneficiaries (a custodial wallet is created automatically at registration) and merchants, seeds wallets, and issues aid; a **super admin** additionally approves activation, whitelists merchants, updates accounts, and manages controls (pause, limits). Full role model in §11.	NGO dashboard
HumanityLink (platform)	Configures and supports the platform: WhatsApp flow, QR/wallet infrastructure, POS app, dashboards, off-ramp automation, technical escalation.	Backend services, deployer role

3. System architecture

3.1 Component map

The platform is a monorepo of six cooperating components:

```
├── leo/                    # Aleo/Leo smart contracts (aid + bulk programs)
├── backend/                # Main NGO backend (Express) — API, DB, WhatsApp bot
├── backend-on-off-ramp/    # Fiat ⇄ USDCx ramp pipeline
│   ├── backend/            # On/off-ramp orchestrators (PM2 processes)
│   └── contracts/          # Solidity HumanityLinkVault (one per NGO)
├── ngo-dashboard/          # Next.js web dashboard for NGOs
├── mobile-app/pos/         # Merchant POS app (Expo / React Native)
└── landing-site/           # Public landing page (Next.js)
```

3.2 Runtime topology

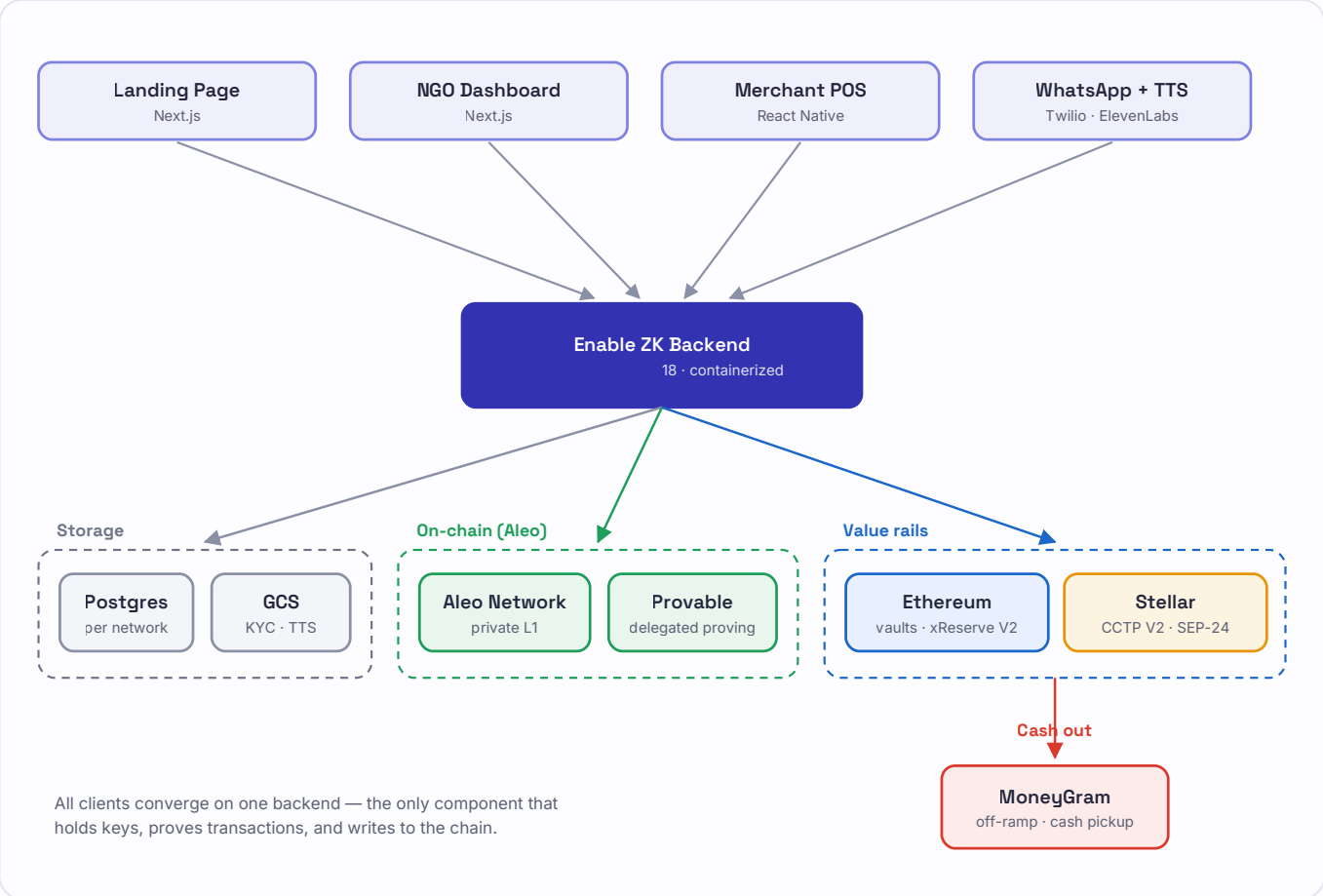


Figure 2 — System architecture. Four clients, one backend, three service planes. The dashed groups separate concerns: plain storage, the privacy layer (Aleo + delegated proving), and the value rails that carry USDC in and out. Colors follow the rail palette used throughout this document: navy = platform, green = Aleo, blue = Ethereum, amber = Stellar, red = fiat/cash.

All four clients converge on the single main backend. The backend is the only component that touches private keys, generates ZK proofs (via delegated proving), and writes to the chain. A separate, sibling **on/off-ramp**

backend runs two long-lived orchestrator processes (an always-on Ethereum listener for on-ramps and a nightly cron for merchant off-ramps).

Hosting: the Next.js frontends (dashboard, landing site) run on **Google Cloud Run**; **all backend services run on a PM2-managed GCE VM** — the main backend, the record-consolidation "join" server, and the on/off-ramp orchestrators. PostgreSQL is one database **per network** (testnet and mainnet never share state), and *within* each database every row is scoped by a unique `ngo_id`, so one NGO can never see another NGO's beneficiaries, merchants, or transactions (see §12).

4. Core techniques and technology choices

This section explains the key engineering techniques used across the platform — the things a developer joining the project needs to internalize before reading any single component.

4.1 Zero-knowledge privacy on Aleo

Aleo is a Layer-1 blockchain with **programmable privacy**: program state can be *private* (encrypted records owned by an address, spendable only with the owner's key) or *public* (on-chain mappings and storage, visible to everyone). Programs are written in **Leo** (this project uses Leo 4.0), and every state transition is proven with a zero-knowledge proof before it is accepted.

Enable ZK exploits this split deliberately:

Data	Visibility	Mechanism
Beneficiary identity	Pseudonymous	Token owner address only
Beneficiary balance	**Private**	Encrypted USDCx `Token` record
Individual payments / P2P transfers	**Private**	`transfer_private` — amounts and counterparties hidden
That *a* transaction occurred + aggregate totals	Public	`finalize` blocks + analytics counter mappings

The consequence: a merchant processing a payment sees an amount and an approval — **never** the beneficiary's identity, history, or balance. The NGO sees decrypted program data (via its view key) for its own program only. The public sees only aggregate flow counters, which is exactly what donor accountability requires.

Proof latency is a UX fact of life: generating the ZK proof for a spend takes up to ~30 seconds. The POS app's waiting screen, the merchant instruction to hand over goods only on approval, and the backend's asynchronous job patterns are all designed around this.

4.2 Delegated proving

Generating Aleo proofs is computationally expensive — far too heavy for a low-end Android phone or a serverless container. The platform therefore uses the **Provable API** for *delegated proving*: the backend constructs the transaction, sends it to Provable's proving service, and receives a proven transaction to broadcast (typically 10–30 s). Fee handling can be routed through a fee-master service so end users never need to hold gas.

This is the technique that makes "a QR code is the wallet" viable: no client device ever needs to prove anything.

4.3 Custodial wallet model

Beneficiaries and merchants do not manage keys. On registration, the backend generates a fresh Aleo keypair per user (`aleoWalletService.generateWallet()`) and stores it — the platform is **custodial by design**, trading self-sovereignty for accessibility in low-connectivity, low-literacy field conditions.

Compensating controls for custody risk:

- Per-transfer and daily caps on P2P transfers.
- An optional **beneficiary YES/NO WhatsApp approval** as a second factor before a merchant-initiated spend (the Twilio webhook signature is what makes the "YES" trustworthy).
- Optional **ProofIt face verification** at the POS: the merchant takes a live selfie of the beneficiary, which the backend matches against the reference photo captured at registration before the spend proceeds (see §4.4).
- On-chain merchant whitelisting as the source of truth for every money flow.
- Envelope encryption of custody keys at rest in the database (see §13.3).
- Accounts are **per-network**: the same phone number maps to different keypairs (or none) on testnet vs. mainnet.

4.4 The QR-code-as-wallet pattern

A beneficiary's "wallet" is a server-generated QR code containing a **base64url-encoded payload** the POS can verify against the backend. Two delivery modes cover the digital-literacy spectrum: a WhatsApp QR image, or a printed QR sticker for low-connectivity users. Public QR view pages are gated by an **HMAC view token** (`?t=`) so a valid page URL cannot be forged for a guessed beneficiary ID. The operational rule follows directly from the design: **treat QR access like cash** — the QR is a bearer instrument, and anyone holding it can initiate a spend.

Two server-side factors compensate for that bearer-instrument nature: the WhatsApp YES/NO approval (§4.3), and **ProofIt face verification**. When face verification is enabled, the merchant's POS takes a **live selfie of the beneficiary at payment time**, and the backend matches it against the **reference photo captured at registration** — the spend only proceeds on a match, so a stolen QR alone is not enough to spend. The feature is a per-deployment toggle, the POS asks the backend whether it is active before prompting for a selfie, and the verification endpoint is rate-limited (which doubles as a bound on third-party ProofIt cost).

4.5 Multi-tenancy as a database concern, not a code concern

The platform serves many NGOs from one deployment. Rather than per-tenant deployments or per-tenant env files, **every per-NGO fact lives in the `ngos` database table** and is resolved per request:

- The NGO's deployed Aleo **program IDs** (aid + bulk-issuance program) and program addresses.
- The NGO's **signing keys** across all three chains (Aleo private key, EVM key, Stellar secret) and its orchestrator/view keys.
- The NGO's Ethereum **vault address** for the on-ramp.

There is deliberately **no global `NGO_PRIVATE_KEY` or `ALEO_PROGRAM_ID`** anywhere in the codebase; every signer resolves the key and program per request, and a unique index enforces that two NGOs can never share a program. The keys themselves are **envelope-encrypted in the database** — decrypted in-process only for the signing operation (see §13.3). Tenant isolation is enforced by `ngo_id` scoping on every row plus fail-closed

cross-tenant guards (e.g. a spend is rejected with a privacy-preserving 404 if the merchant and beneficiary belong to different NGOs — or to no NGO at all). Section 12 covers this in depth.

4.6 Idempotency and the "in-doubt money" state machine

Value-moving operations cross multiple networks and can time out at any hop. The platform's answer is a set of layered idempotency techniques:

- **On-chain idempotency nullifiers.** Bulk issuance carries a `request_id`; the contract records it in an `issued_requests` mapping and rejects re-use, so a re-broadcast batch can never double-mint — independent of any backend retry logic. The Ethereum vault does the same with a `keccak256(txHash || logIndex)` idempotency key per inbound transfer.
- **A durable `money_operations` state machine** in Postgres, wrapped by an `onchainGuard` preflight/settle pattern. A transaction that was broadcast but not confirmed within the polling window is held **in-doubt** — never auto-confirmed, never blindly retried — and resolved later by a **reconciler** (automatically when a txId exists, manually otherwise).
- **Per-merchant advisory locks** (Postgres advisory locks) serialize concurrent spends against the same merchant across processes.
- **Crash-resume state files** in the off-ramp pipeline: each merchant's multi-step payout persists its step state, so an operator can resume a crashed pipeline from the exact step that failed.

The principle throughout: *the chain is the source of truth; the database is a cache and a ledger of intent.*

4.7 Fail-closed configuration — beyond standard env validation

Validating configuration at startup is standard engineering, and the platform does the standard thing: every required variable goes through a `requireEnv()` helper that throws at boot, so a misconfigured deployment never starts serving. (Even this table-stakes layer has earned its keep — a malformed `.env` line once left a block-height variable silently unset, and a balance service quietly scanned the chain from block 0.)

What goes beyond standard practice is the layer above it: the system validates not just that configuration *exists*, but that it is **coherent** — cross-field and cross-chain invariants that a generic env validator cannot know about, because they encode domain knowledge of which value combinations lose money:

- **Network coherence.** A deployment's network is declared once (`ALEO_NETWORK`), but a dozen other values each *imply* a network — RPC endpoints, the xReserve contract, bridge domain IDs, and above all the stablecoin program (`test_usdcx_stablecoin.aleo` vs `usdcx_stablecoin.aleo`). Operators switch networks by toggling comment blocks in `.env`, so a half-edited file can mix the two. A self-check treats `ALEO_NETWORK` as canonical, cross-verifies every network-implying value against it, and aborts startup on any disagreement. Without it, a mainnet backend pointed at the test token program would "issue aid" in worthless test tokens.
- **Cross-chain agreement.** The off-ramp refuses to run if `ALEO_NETWORK` and `STELLAR_NETWORK` disagree: the merchant payout crosses Aleo → Ethereum → Stellar, and mixed networks would burn real USDCx on one side while attempting the payout against a test network on the other — value stranded mid-bridge.
- **No default tenant.** In a multi-NGO system there is no safe fallback identity. `getSigningKey` throws when an NGO's row or key is missing (a half-onboarded NGO produces an error, never a transaction signed with the wrong key), and `executeTransition` has no module-level `PROGRAM_ID` default — a caller that forgets to pass the program fails loudly instead of silently executing against another tenant's contract.

- **Security toggles cannot be off in production.** A dev-only flag can skip Twilio webhook signature validation locally; outside development it is ignored, and a production process with validation somehow disabled refuses to boot. The WhatsApp webhook is the security boundary for beneficiary approvals — with validation off, anyone knowing the URL could POST a forged "YES" — so this second factor cannot be misconfigured away.

The common thread: every check converts what would be a **runtime money bug** — wrong network, wrong tenant, missing auth — into a **deploy-time crash**, the cheapest possible place to catch it.

4.8 Private balances are UTXOs — fragmentation and the auto-join server

Developers coming from account-based chains (Ethereum, most L1s) should recalibrate: on Aleo, a private balance is **not** an account whose number goes up and down. It is a set of discrete, encrypted **records** — structurally the same model as Bitcoin's **UTXOs** (unspent transaction outputs), with the contents encrypted:

- A private transfer **consumes whole input records and produces new ones**: one record for the recipient, one "change" record back to the sender. Nothing is ever edited in place; a consumed record is marked spent on-chain via a nullifier (its serial number), which is what prevents double-spends without revealing *which* record was spent.
- A wallet's balance is therefore the **sum of its unspent records**, and paying an amount requires an input record large enough to cover it.

This has two operational consequences that shape the platform:

1. **Fragmentation grows without bound.** Every issuance received, every payment accepted, and every change output adds a record to a wallet. Left alone, an active wallet's record set only grows — a busy merchant accepting fifty payments a day accumulates fifty new records a day, indefinitely. Beyond the bookkeeping cost, scanning and decrypting an ever-growing record set slows every balance read and spend.
2. **Spend-size mismatch.** A transition takes a fixed, small number of record inputs. A wallet can hold $10 in total but, if it is scattered across twenty $0.50 records, no admissible input covers a $5 payment. Records must first be **merged with a `join` operation** — itself a proven on-chain transaction, adding tens of seconds of proving time if it has to happen inline during a payment.

The platform's answer is the **auto-join server** (`hl-auto-join`), a dedicated background process running under PM2 alongside the main backend. On a fixed interval (default every 15 minutes) it sweeps every custodial wallet; any wallet holding more than a threshold number of USDCx records (default 5) has them joined down toward a target count (default 1), bounded per wallet per sweep and paced between wallets to spread proving load. Because consolidation happens continuously in the background, the **inline join in the payment path almost never fires** — which is what keeps a POS payment at a single proof (~30 s) instead of two or three.

4.9 Technology stack summary

Layer	Technology	Used for
ZK blockchain	**Aleo**, **Leo 4.0**, snarkOS	Private aid records, on-chain controls, analytics counters
Stablecoin	**USDCx** (`usdcx_stablecoin.aleo` ; `test_*` variant on testnet)	Aid denomination; private + public token flows
Proving	**Provable API** (delegated proving, fee-master)	Server-side ZK proof generation (~10–30 s)
EVM chain	**Ethereum** (Sepolia on testnet), **Solidity + Foundry**	Per-NGO on-ramp vault, USDC custody in transit
Bridging	**Circle xReserve V2** (Ethereum ⇄ Aleo), **Circle CCTP V2** (Ethereum → Stellar)	Cross-chain USDC movement with Circle attestation
Cash-out rail	**Stellar** + **MoneyGram SEP-24** (SEP-10 auth)	Merchant fiat payout via cash pickup
Backend	**Node 18, Express 5, PostgreSQL (`pg`)**, PM2, Docker	API, orchestration, custody, state machines
Dashboard	**Next.js 14 (App Router), TypeScript, Tailwind, next-intl**	NGO admin UI (EN/ES), BFF proxy pattern
Mobile POS	**Expo / React Native (expo-router)**	Merchant scan-and-pay + off-ramp
Identity	**Auth0** (namespaced custom claims), HMAC device tokens, service tokens	Dashboard SSO, POS device auth, orchestrator auth
Messaging	**Twilio WhatsApp** (Content List-Picker menus, signed webhooks)	Beneficiary onboarding, bot commands, approvals, payout references
Voice	**ElevenLabs TTS**	Voice notes for low-literacy onboarding
Verification	**ProofIt** face matching	Live selfie vs. registered photo at the POS
Storage / infra	**Google Cloud Storage** (signed URLs), Cloud Run (frontends), GCE + PM2 (backends)	KYC photos, TTS cache, hosting
Anti-abuse	Cloudflare **Turnstile**, `express-rate-limit` , SendGrid	Support pipeline, brute-force limits
Price data	**Chainlink**-backed on-chain feed (via `ethers`)	Live USD→COP display rate

5. Smart contracts — Aleo (Leo)

Two Leo 4.0 programs per NGO implement the on-chain aid logic. Aid is standard USDCx drawn through the stablecoin program (`test_usdcx_stablecoin.aleo` on testnet, `usdcx_stablecoin.aleo` on mainnet).

```
leo/
├── v11/        # Core aid contract        → humanity_link_aid_v14_goal.aleo
└── v11bulk/    # Bulk-issuance companion  → humanity_link_bulk_v14_goal.aleo
```

5.1 Deployment model: one instance per NGO, non-upgradeable

Every NGO deploys its **own instance** of both programs. The admin authority is set at deploy time to the deployer (`self.program_owner`) inside the constructor — there is **no hardcoded authority address**, which is what makes the multi-NGO model work without code changes per tenant.

Both constructors assert `self.edition == 0u16`, making the programs **permanently non-upgradeable**: any change requires a fresh deploy (`leo upgrade` is intentionally impossible). Because a bricked admin would be unrecoverable on an immutable program, `transfer_admin` refuses both the zero/burn address and the current admin as targets.

5.2 Core aid contract (`humanity_link_aid_v14_goal.aleo`)

State. The contract mixes Leo `storage` singletons with mappings, chosen per consumer:

Kind	Items	Why this representation
`storage` singletons	`admin_address`, `flags` (paused), `clocks` (merchant off-ramp cooldown in blocks, default ≈24 h), `limits` (minimum merchant off-ramp, default $6.00)	Modern Leo storage variables; read as storage by off-chain tooling
`u8 ⇒ u128` counter mappings keyed at `0u8`	`total_issued`, `total_spent`, `total_ben_offramp`, `total_merch_offramp`, `total_merch_buy`, `total_p2p`	**Deliberately kept as mappings** so the v2 REST API can read them at `/mapping/<name>/0u8` — the dashboard's public volume breakdown depends on this
Per-address mappings	`authorized_issuers`, `whitelisted_merchants`, `bridge_address` (per-merchant off-ramp target), `last_merch_offramp` (cooldown tracking), `merchant_vault` (pending payout ledger)	Access control + settlement bookkeeping

A derived metric, *total volume* = `total_spent + total_ben_offramp + total_merch_offramp + total_merch_buy + total_p2p`, is computed off-chain from the public counters.

Transitions fall into three groups:

1. **Admin** (all assert `self.caller == admin_address`): `transfer_admin`, `authorize_issuer` / `revoke_issuer`, `whitelist_merchant` / `revoke_merchant`, `set_bridge_address` / `revoke_bridge_address`, `set_merch_offramp_period`, `set_min_merch_offramp`, `pause` / `unpause`, `sweep_vault` (emergency), `settle_merchant` (pure accounting decrement of `merchant_vault` after a fiat payout succeeds — no USDCx moves).
2. **Issuance** (public → private USDCx): `issue_aid` (1 recipient), `bulk_issue_2`, `bulk_issue_4`. Each finalize asserts not-paused and `authorized_issuers[caller]`, then bumps `total_issued`. The program's own **public USDCx balance is the aid pool** that issuance draws from via `transfer_public_to_private`.
3. **Transfers & off-ramps** (private → private/public USDCx; all assert not-paused):

Transition	Signer	Gate	Purpose
`spend`	Beneficiary	merchant whitelisted	Pay a merchant (goods or cash-out); returns merchant token + change token
`pay_beneficiary`	**Merchant**	merchant whitelisted	Merchant buys goods *from* a beneficiary (mirror of spend)
`transfer_p2p`	Beneficiary	none on-chain	Shielded beneficiary→beneficiary transfer; authorization happens off-chain via a single-use WhatsApp confirm link
`offramp_beneficiary`	Beneficiary	merchant whitelisted	Cash-out via a merchant (merchant hands over local cash)
`offramp_merchant`	**Merchant**	whitelisted + `amount ≥ limits` + `bridge_addr == bridge_address[merchant]` + 24 h cooldown	Merchant settles USDCx to its **own dedicated bridge address**; bumps `merchant_vault` — the aid pool is never touched

Design notes worth understanding:

- **Per-merchant bridge addresses** mean there is no shared off-ramp pool: each merchant's cash-out flows through its own keypairs, so a key compromise is scoped to one merchant's in-flight balance.
- The **cooldown period is bounded** (`MAX_MERCH_OFFRAMP_PERIOD ≈ 30 days`) so an admin fat-finger cannot set an unsatisfiable cooldown or overflow the assertion.
- `bridge_address` is **not** seeded at deploy — until an admin sets it for a merchant, `offramp_merchant` reverts for that merchant. Fail-closed by construction.

5.3 Bulk-issuance companion (`humanity_link_bulk_v14_goal.aleo`)

Leo 4.0 caps a function at **16 outputs** (a `Final` future counts as one). The core contract can therefore batch at most 4 recipients per call. The bulk companion pushes this to the limit with `bulk_issue_8` (9 outputs) and

`bulk_issue_15` (16 outputs — exactly at the cap), taking `Recipient { addr, amount }` structs and folding all child `issue_aid` futures into one sequential finalize.

Two-layer authorization connects the programs:

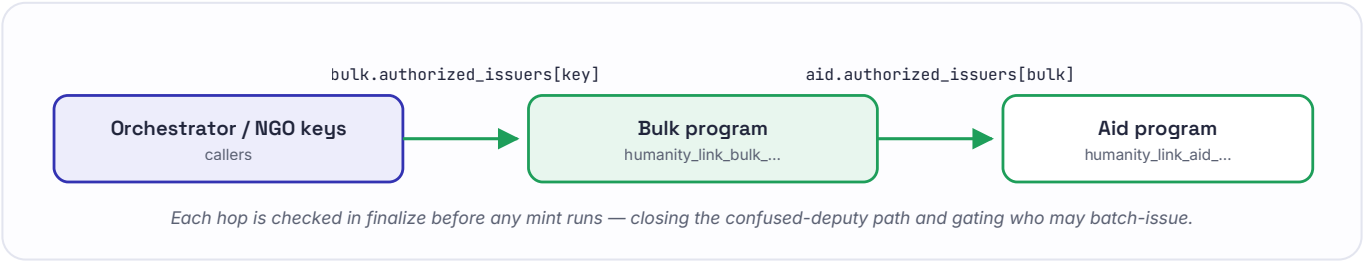


Figure 3 — Two-layer issuer authorization. The aid program trusts exactly one minter — the bulk program — and the bulk program trusts an explicit set of caller keys.

Each `bulk_issue_*` finalize — *before any child mint executes* — (1) asserts the caller is authorized (this closed a **confused-deputy drain**: earlier versions let anyone route mints through the bulk program), and (2) checks-and-marks the `request_id` in the `issued_requests` nullifier mapping, giving on-chain idempotency independent of backend retries.

Post-deploy initialization checklist (per NGO). Only the issuer mappings need initializing — `admin` is set automatically at deploy (to the deployer, so deploy each NGO's programs with that NGO's own key), and `issued_requests` fills at runtime:

Program	Add to `authorized_issuers`	Why
Aid	The **bulk program's address**	When bulk calls `issue_aid`, `self.caller` is the program address — this is Layer 1
Aid	The **NGO's signing-key address**	Required for direct `issue_aid` / `bulk_issue_2/4` calls (single and small-batch issuance)
Bulk	The **NGO's signing-key address** (and any separate orchestrator key)	Layer 2 — gates who may call `bulk_issue_8/15`

Being admin does **not** imply being an issuer: the admin functions and `authorized_issuers` are independent gates, so the admin key must be added explicitly if it is the caller. Nothing else belongs in either mapping — not beneficiaries, not merchants, and not the aid program on the bulk side (trust points only one way).

As concrete commands, per NGO (all three signed by that NGO's **admin key** — the key that deployed the programs, and therefore the only key the `assert_eq(caller, current_admin)` gate accepts):

```
# 1) Layer 1 — on the AID program: authorize the BULK PROGRAM as an issuer.
#    The argument is the bulk program's own on-chain address (aleo1…) — every
#    deployed program has one, distinct from its program ID and from any account.
#    Find it on the program's explorer page.
leo execute humanity_link_aid_v14_goal.aleo::authorize_issuer \
  <BULK_PROGRAM_ADDRESS> \
  --private-key "$ADMIN_PK" --network mainnet --broadcast \
  --endpoint https://api.explorer.provable.com/v2

# 2) Also on the AID program: authorize the NGO's SIGNING-KEY ADDRESS, so the
#    backend's direct issue_aid / bulk_issue_2 / bulk_issue_4 calls pass the gate.
leo execute humanity_link_aid_v14_goal.aleo::authorize_issuer \
  <NGO_SIGNING_ADDRESS> \
  --private-key "$ADMIN_PK" --network mainnet --broadcast \
  --endpoint https://api.explorer.provable.com/v2

# 3) Layer 2 — on the BULK program: authorize the NGO's SIGNING-KEY ADDRESS
#    to call bulk_issue_8 / bulk_issue_15.
leo execute humanity_link_bulk_v14_goal.aleo::authorize_issuer \
  <NGO_SIGNING_ADDRESS> \
  --private-key "$ADMIN_PK" --network mainnet --broadcast \
  --endpoint https://api.explorer.provable.com/v2
```

The two placeholders trip people up, so to be precise: `<BULK_PROGRAM_ADDRESS>` is the **program's** address — when the bulk program calls `aid.issue_aid`, the aid program sees `self.caller` equal to that program address, which is why an account address would not work here. `<NGO_SIGNING_ADDRESS>` is the **account** address derived from the NGO's signing key (`ngos.aleo_private_key`) — the mapping stores addresses, never keys, and this may well be the same account as the admin, but it must still be listed explicitly. If a separate orchestrator key issues batches, repeat command 3 for its address.

5.4 Privacy model recap

Data	Visibility
Beneficiary identity	Pseudonymous (token owner address)
Beneficiary balance	Private (encrypted record)
Merchant payment / P2P transfer	Private (`transfer_private`)
Transaction occurrence & aggregate totals	Public (finalize + counter mappings)

6. Smart contracts — Ethereum (Solidity)

The on-ramp is anchored by `HumanityLinkVault` — a small (~240-line) Solidity contract, built and tested with **Foundry** (35-test suite incl. reentrancy and fuzz tests), deployed **once per NGO**.

6.1 What the vault does — and deliberately does not

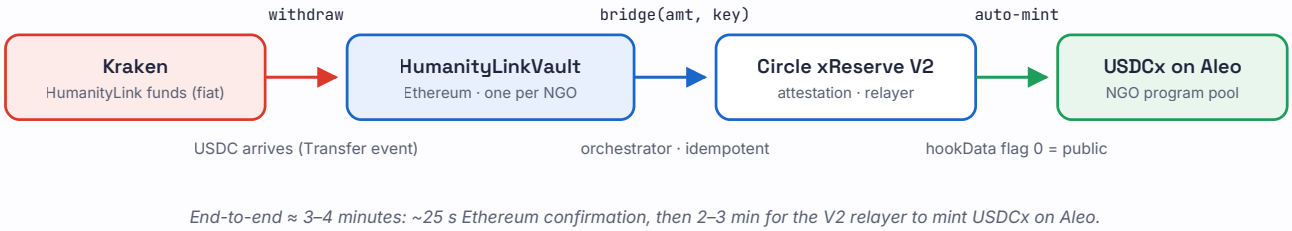


Figure 4 — On-ramp: fiat to private aid pool. The vault holds no reserve: every arriving dollar is bridged in full on the next orchestrator pass, keyed by keccak256(txHash ‖ logIndex) so replays are harmless. The Aleo recipient is immutable in the contract — a compromised operator can trigger a bridge, never redirect one.

There is **no `deposit()` flow, no reserve, no split**. Every dollar that arrives is bridged in full on the next orchestrator pass; the vault holds funds only briefly between arrival and the `bridge()` call. This minimalism is a security posture: the contract has almost no state to attack.

6.2 Key properties

- **Immutable destination.** The NGO's Aleo recipient (stored both as a human-readable bech32m string and as the `bytes32` the bridge call needs) is fixed at construction. A compromised operator key can *trigger* bridging of the vault balance — but can never redirect it.
- **Idempotency on-chain.** `bridge()` requires a unique `bytes32 idempotencyKey` = `keccak256(txHash ‖ logIndex)` of the source `Transfer` event; used keys are stored and duplicates revert. The orchestrator can therefore submit blindly across restarts and replays and let the chain decide.
- **Roles.** `owner` (deploys, rotates operator, `emergencyDrain()`) and `operator` (the orchestrator EOA; may call `bridge()`). One operator can serve many NGOs' vaults.
- **Dust guard.** A 5 USDC minimum per bridge call avoids xReserve fee dust.
- **Auto-mint via hookData.** On xReserve **V2**, the vault's `depositToRemote` carries a 65-byte `hookData` whose first byte is the mint flag (`0` = public mint). The V2 relayer auto-mints USDCx on Aleo after Circle attests — there is **no off-chain mint API** left to secure (the V1 ANF endpoint is gone).
- Events (`BridgeExecuted`, `OperatorUpdated`, `OwnershipTransferred`, `EmergencyDrained`) and views (`vaultBalance`, `totalBridged`, `processedTransfers`) give the orchestrator and operators full observability.

Onboarding a new NGO = encode their Aleo address to `bytes32`, deploy a fresh vault, record the vault address and operator key on the NGO's database row, restart the orchestrator. The orchestrator discovers all vaults from the `ngos` table — one process watches many vaults.

Note: the deployed constant `ALEO_DOMAIN = 10002` is the xReserve remote-domain ID for Aleo on **testnet**; a mainnet vault deployment must use the mainnet domain.

7. The main backend

The main backend (Node 18 / Express 5 / PostgreSQL, containerized on port 8080) is the mediator between every client and the chain. It is **multi-tenant** (per-NGO keys/programs from the DB, §12) and **multi-network** (single-network, fail-closed deployments, §15).

7.1 Responsibilities at a glance

- **Custodial wallets** — fresh Aleo keypair per beneficiary/merchant on registration.
- **Onboarding** — direct API, bulk CSV upload (with progress streamed via SSE), or a full WhatsApp bot conversation.
- **Aid issuance** — single + bulk, with dry-run previews, single-use bulk snapshots, and delegated proving.
- **Payments** — the `spend` flow with per-merchant advisory locks and the optional WhatsApp YES/NO approval second factor.
- **P2P transfers** — WhatsApp-driven `transfer_p2p` with confirm-link authorization and caps.
- **Merchant-buys-from-beneficiary** — `pay_beneficiary`, the mirror flow, run asynchronously with job polling.
- **Off-ramp entry points** — beneficiary cash-out via `spend`; per-merchant off-ramp triggers (sync + async) that hand off to the bridge pipeline.
- **In-doubt money handling** — the `money_operations` state machine plus reconciler endpoints.
- **Goods categories** — per-NGO purchase-tagging taxonomy (`required` / `optional` / `off`), changeable with zero app redeploys (the POS fetches it live).
- **Reporting** — dashboard stats (vault flow, weekly activity, on-chain volume breakdown), paginated transaction history with the USD→COP rate captured *at execution time*, XLSX exports (NGO-wide, and merchant-initiated exports delivered over WhatsApp).
- **Verification & support** — ProofIt face verification at the POS, Cloudflare Turnstile + SendGrid support tickets.
- **Voice/TTS** — ElevenLabs voice notes for low-literacy WhatsApp onboarding.
- **Auto-join** — the `hl-auto-join` background service that continuously consolidates each wallet's fragmented USDCx records so the inline record-join in the payment path rarely fires. See §4.8 for the UTXO record model that makes this necessary.

7.2 Structural patterns

The codebase follows a conventional layered layout — `routes/` (HTTP surface) → `services/` (domain logic) → `db/repositories/` (data access) — with a few patterns worth calling out:

- **Repository pattern with security-aware accessors.** `ngoRepo` exposes `getSigningKey(ngoId)` (secret; throws if missing) separately from `getProgramIds(ngoId)` and `getPublicConfigById(ngoId)` (non-secret; *never selects a key column*). The type of accessor a route uses is itself a security control.
- **Middleware chain as policy.** Auth (`checkJwt` / `checkJwtOrServiceToken` / device-token), tenant resolution (`ngoContext`), RBAC (`requirePermission`), and per-NGO signing context (`loadNgoSigning`) compose per route.
- **Exact-money parsing.** Untrusted amount strings are parsed to exact micro-units with BigInt (`utils/micro.js`) and non-canonical strings are rejected. 1 USDCx = 1,000,000 microcredits; no floats anywhere near money.

- **Canonical config helper.** All required env goes through a fail-loud `requireEnv()`.

7.3 API surface (summary)

The full endpoint tables live in the backend README; the shape of the API is:

Area	Representative endpoints	Auth
Beneficiaries	register (multipart KYC photos), list, view, bulk-register, allowlist, soft-delete	Dashboard JWT (+ RBAC)
Merchants	register, verify POS credentials (rate-limited), whitelist/revoke, balance, categories, export, regenerate token	JWT / device token per route
Aid	issue, bulk-issue (+preview), bulk-credits (+preview), balance, history	JWT (+ RBAC)
Payments	`POST /payments` (spend), `/payments/offramp`, `/payments/merchant-buy` (async + status polling)	POS device token
P2P	receive / pay / confirm / status pages	Public + opaque single-use tokens
Off-ramp	beneficiary + merchant triggers, sync/async, status polling	Device token / service token
Stats	dashboard, activity, transactions (recent + paginated), on-chain vault breakdown	JWT
NGO config	own-NGO config, goods-categories management	JWT (+ RBAC)
Admin	~22 signing endpoints: issuers, whitelist, bridge provisioning, pause, sweep, settle, fund, reconcile/stuck-ops	JWT **or** service token, per-route RBAC
Other	exchange rate, XLSX export, support tickets, face verification, WhatsApp webhook	Mixed

7.4 The WhatsApp bot

WhatsApp (via Twilio) is the beneficiary's entire interface. The bot handles text, voice notes, and image uploads (KYC ID photos → GCS with signed URLs), speaks English and Spanish (accent/case-insensitive commands), and optionally presents a Twilio **Content List-Picker** interactive menu with a plain-text fallback. A per-network badge (TESTNET / MAINNET, with distinct emoji) is shown so field testers can never confuse environments.

Commands: `QR`, `BALANCE`, `STATUS`, `REPORT`, `JOIN` (choose an NGO when several share the number), `NETWORK`, `VOICE ON/OFF`, `START`, and a `TRANSFERS` submenu with exactly two P2P entry points: `PAY <HL-ID> <amount>` and `RECEIVE`.

As shipped, the List-Picker menu offers five actions — **QR code** (show your wallet QR), **Balance**, **Status** (your details), **Transfers** (send/receive between beneficiaries), and **Report a problem** — and the returning-user welcome message packs the essentials into one bubble: HL-ID, network badge, balance with its COP equivalent, and the tokenized QR link. Picking **Transfers** returns the two P2P entry points with worked examples.

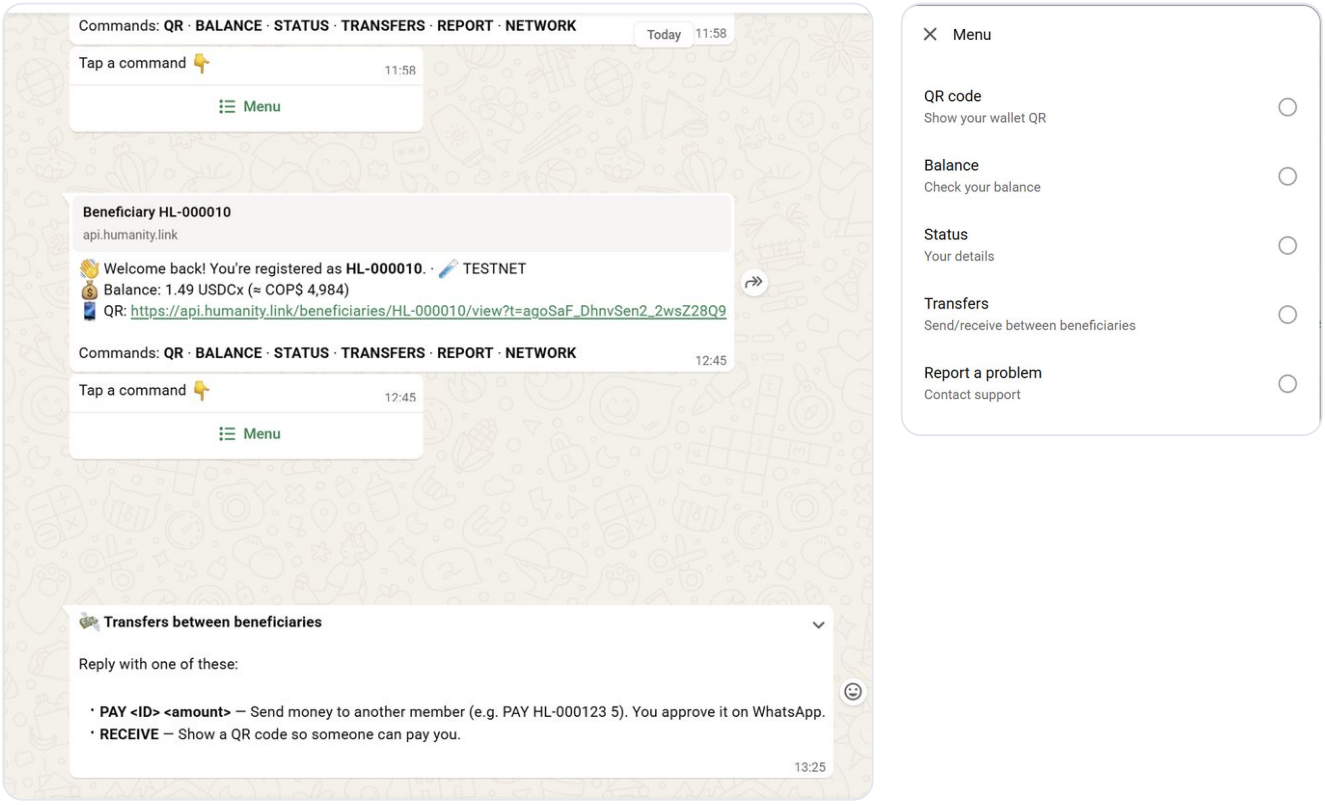


Figure 6 — The WhatsApp bot in production (testnet). Left: the returning-user welcome card (HL-ID, network badge, balance with COP equivalent, tokenized QR link) and the Transfers submenu with the two P2P entry points — `PAY <ID> <amount>` and `RECEIVE`. Right: the Twilio Content List-Picker main menu; a plain-text command list is the universal fallback.

Inbound messages from unknown numbers are rate-limited; the webhook is signature-validated in production (see §13).

7.5 Flow deep-dive: beneficiary spend

```
1. Beneficiary shows QR at merchant POS
2. POS scans the QR (device-token gated session)
3. If face verification is enabled (POS checks /transactions/verify-config):
   merchant takes a live selfie of the beneficiary → POST /transactions/verify
   → ProofIt matches it against the registration photo; a mismatch stops the payment
4. POS submits the payment → POST /payments
5. Backend acquires the per-merchant advisory lock
6. Validates: balance, on-chain merchant whitelist, amount ≤ balance, goods category
7. If approval is required: WhatsApp YES/NO to the beneficiary; POS polls /payments/status
8. Backend executes `spend` via delegated proving — only a confirmed-accepted tx is success;
   a timeout is held IN-DOUBT (never auto-confirmed) for the reconciler
9. On-chain: beneficiary balance -= amount; merchant_vault[merchant] += amount
10. Later: the nightly orchestrator sweeps and off-ramps the merchant balance
```

Two details are easy to miss and important:

- Step 4's whitelist check re-reads **the chain**, not the DB — the DB cache self-heals from the chain and never authorizes a money flow by itself.

- Step 6's failure semantics: *broadcast-but-unconfirmed is not failure and not success*. It parks as in-doubt and a human-supervised reconciler resolves it. This is what prevents the classic double-spend-on-retry bug in bridged-money systems.

7.6 Flow deep-dive: P2P transfer

No app, no login — WhatsApp plus a browser. Two entry paths converge on one confirm step:

- **RECEIVE** (payee shows a QR): the payee sends `RECEIVE`; the bot returns a link rendering a QR that encodes a single-use, HMAC-signed `recv` identity token (10-minute TTL). The payer scans it with a native camera → lands on `/p2p/pay` → enters their HL-ID and an amount.
- **Direct PAY**: the payer texts `PAY HL-000123 5.00` to the bot.

Either way, the backend validates both parties (same NGO, caps, no self-pay), creates a **single-use, short-expiry** `payment_request`, and sends an **opaque confirm link to the payer's enrolled WhatsApp number** — *that delivery is the authentication factor*. The payer reviews the payee's name/photo/amount and confirms; only then does the backend execute `transfer_p2p` signed with the payer's custodial key (serialized per payer, idempotent via the on-chain guard). Caps: per-transfer USD max, rolling daily per-payer max, and an hourly inbound-request cap per payer against nuisance requests.

This design is worth studying as a pattern: it achieves reasonable authorization for custodial payments **without any client credential**, by leaning on possession of the enrolled WhatsApp number plus single-use hashed tokens.

That said, treat this mechanism as a **first iteration**: the best long-term design for P2P transfers is still under active research, and a dedicated **beneficiary mobile app is on the roadmap** — which would allow richer, in-app confirmation and discovery flows than the current WhatsApp-plus-browser approach can offer.

7.7 Flow deep-dive: merchant buys from a beneficiary

`pay_beneficiary` is the mirror image of a spend: the **merchant** pays a beneficiary out of the merchant's *own* USDCx balance — used when a merchant purchases goods from a beneficiary (produce, crafts, services), keeping aid money circulating locally. On the POS it is the **"Buy from Beneficiary"** option in the scan chooser.

Mechanics, and how they differ from a spend:

- **The merchant signs — but never holds a key.** On-chain, `pay_beneficiary` is executed with the merchant's custodial key (the merchant owns the Token being spent), gated on the merchant's own whitelist entry. As everywhere in the platform, the signing happens **server-side**: private keys are never stored on, or sent to, the merchant's mobile app — the POS carries only its device token, and the backend signs with the custodial key it holds encrypted in the database (§13.3).
- **No beneficiary approval step.** Unlike a spend, there is no WhatsApp YES/NO second factor — the merchant is authorizing its *own* funds, so the fund-drain-protection rationale doesn't apply.
- **Asynchronous by design.** The POS calls `POST /payments/merchant-buy` (device-token gated) and immediately receives a `jobId`, then polls `GET /payments/merchant-buy/status` — the ~30 s proving time is hidden behind polling rather than a blocking request.
- **Same guardrails otherwise.** The buy enforces the same-NGO boundary, honors the NGO's goods-category mode, re-checks the on-chain whitelist, and reuses the per-merchant advisory lock and in-doubt `money_operations` handling.

- Recorded as transaction type `merchant_buy` and counted in the public `total_merch_buy` on-chain counter — it is one of the five components of the dashboard's Total Volume breakdown.

8. The NGO dashboard

A **Next.js 14 (App Router)** / TypeScript / Tailwind application, the NGO's control room. It is a *thin* client by design: it holds no keys, no program IDs, and no blockchain logic — everything flows through the backend.

8.1 Feature surface

- **Overview** — vault flow stats (USDCx available for aid, total volume, pending merchant settlements, converted-to-cash), a volume breakdown by on-chain component (Spend / Merch Cashout / Merchant Buy / P2P), stat grids, a weekly activity chart, recent transactions.
- **Beneficiaries** — authorize, suspend, inspect, soft-delete (status → `deleted`, keys preserved; super-admin gated), bulk registration with a live progress stream.
- **Merchants** — whitelist/revoke, POS provisioning.
- **Issue aid** — single issuance plus bulk-issue and bulk-credits modals (with previews).
- **Transactions** — server-side paginated and type-filtered history; each row shows its **point-in-time COP amount** for Colombia NGOs.
- **Goods categories** — the per-NGO purchase-tagging admin (rename, show/hide, set the `required` / `optional` / `off` mode) — applied to the POS instantly.
- **Settings** — the admin/deployer control panel: transfer admin, issuer management, whitelisting, bridge provisioning, pause/unpause, vault funding and sweeping, settlement configuration, fund-via-on-ramp. Every control is shown/hidden by RBAC role.
- **i18n** — English/Spanish via `next-intl` with an in-app toggle.

8.2 The BFF proxy pattern

Client pages **never call the backend directly**. They call a same-origin **Backend-for-Frontend proxy** (`/api/backend/[...path]`), which:

1. Reads the Auth0 session cookie and exchanges it server-side for the API access token.
2. Forwards the request with `Authorization: Bearer <token>` — **the token never reaches the browser**.
3. Caps request bodies and *streams* responses, so Server-Sent Events (bulk-register progress) and file downloads (XLSX) pass through unbuffered.

This is the standard remedy for the "SPA holding API tokens" problem, and it also gives one choke point for request policy.

8.3 Tenant awareness without tenant builds

The dashboard is one deployment for all NGOs. Nothing program-specific is baked into the build: on login, a context hook fetches `GET /ngo/config` (keyed off the authenticated session), which returns *public-safe* per-tenant config — program ID/name/address, admin/funding addresses, status, country. The repo history is explicit that this replaced `NEXT_PUBLIC_PROGRAM_*` build-time variables, because one build cannot serve many NGOs — and because a private key in a `NEXT_PUBLIC_*` var would be baked into the public browser bundle (a class of mistake the CI now scans for: the image build **fails** if an Aleo private/view key pattern appears in the compiled bundle).

Country-conditional behavior hangs off the same config: Colombia NGOs (`countryIso === "CO"`) get live COP equivalents beside every USDCx amount, sourced from the backend's Chainlink-backed `GET /exchange-rate`,

with a platform-wide rate strip showing the applied rate and freshness. The rate hook *fails open* to a hardcoded fallback — a display concern should not take the dashboard down.

8.4 Security-relevant details

- Auth0 middleware gates every page and non-public API route — but as **defense-in-depth only**: secret-using routes re-check auth and role inside their own handlers, because Next.js middleware has been bypassable (CVE-2025-29927).
- The container preloads a small shim that **drops all HTTP `Upgrade` requests** — a compensating control for an SSRF-via-WebSocket-upgrade vulnerability in self-hosted Next.js (CVE-2026-44578). The app uses no WebSockets, so dropping upgrades wholesale is safe.
- A redacting logger ensures tokens/keys never hit logs.
- View keys and private keys are **backend-only secrets**; the dashboard consumes only derived aggregates.

9. The merchant POS app

The merchant point-of-sale app is an **Expo / React Native (expo-router)** project. Merchants activate a device with NGO-issued credentials, scan beneficiary QR codes to process aid payments, and cash out their balance.

The home screen presents five actions: **Scan Customer QR** (accept an aid payment or cash-out), **My Balance** (total USDCx earned, with local-currency equivalent), **Convert to Cash** (the MoneyGram off-ramp), **Transaction History** (an Excel report of the merchant's transactions, delivered over WhatsApp), and **Report a Problem**. The header shows the merchant's name, MR-ID, the active network badge, and an EN/ES language toggle.

Scanning a beneficiary QR opens a purpose chooser — *"How can we help?"* — with three actions that map one-to-one onto backend flows and on-chain transitions:

POS action	Backend call	On-chain transition	Who pays whom
Buy Goods	`POST /payments`	`spend`	Beneficiary → merchant
Buy from Beneficiary	`POST /payments/merchant-buy` (async)	`pay_beneficiary`	Merchant → beneficiary, from the merchant's own USDCx
Off-Ramp to Cash	`POST /payments/offramp`	`spend` (cash-out)	Beneficiary → merchant, merchant hands over cash

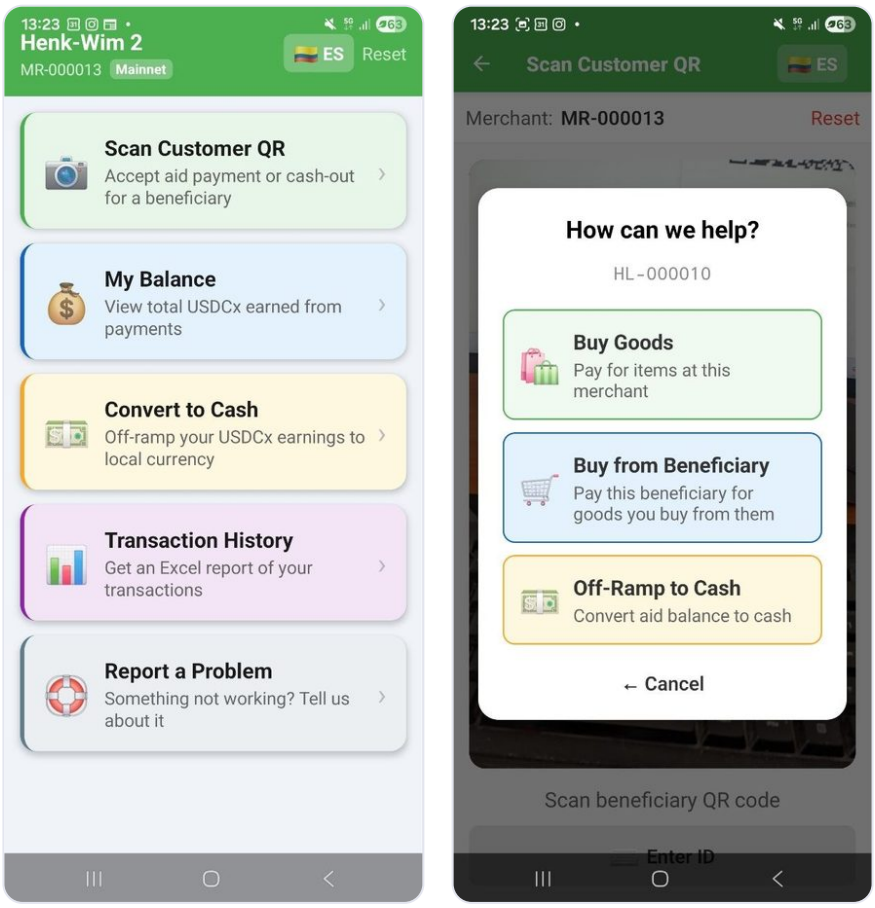


Figure 7 — The merchant POS (mainnet). Left: the home screen with the five merchant actions; note the MR-ID and network badge in the header. Right: the purpose chooser after scanning beneficiary HL-000010 — Buy Goods (`spend`), Buy from Beneficiary (`pay_beneficiary`), and Off-Ramp to Cash.

Key characteristics:

- **Device activation** binds the app installation to a merchant using an NGO-issued **device token** — the bearer credential for every subsequent POS call (payments, balance, categories, exports, reports). Tokens can be regenerated server-side; with HMAC hashing enabled the usable token is shown exactly once.
- **Network selection is a user-facing toggle on the activation screen** (it swaps the backend host), because merchants participating in pilots may need to move between testnet and mainnet programs. Everything else about the network split is server-side.
- The payment flow: scan QR → enter the amount in local currency (the app shows the equivalent balance value) → submit → **wait up to ~30 s for the ZK proof and on-chain approval** → hand over goods/cash only on the green approved screen.
- The app also surfaces the merchant's USDCx balance with a local-currency equivalent, transaction history (references are the anchor for every dispute), goods-category tagging per the NGO's configured mode, a "convert to cash" (MoneyGram off-ramp) entry point, transaction export (delivered via WhatsApp), and problem reporting.
- Distribution is via Android APK and iOS TestFlight.

The merchant-facing operating rules map one-to-one onto technical controls: *only the official app* (device token), *wait for approval* (on-chain confirmation), *no extra fees* (monitored by field staff; suspension = whitelist revoke, enforced on-chain), *keep references* (transaction history + reconciliation).

10. The on/off-ramp pipeline

A separate backend of **two PM2-managed Node processes** moves value between fiat and the chain. It shares the database with the main backend (and mirrors its NGO-context and secret-decryption modules byte-for-byte) but has no HTTP API of its own.

10.1 On-ramp: always-on listener

`hl-onramp-orchestrator` polls Ethereum (default every 60 s) and watches every active NGO's vault for inbound USDC `Transfer` events. For each event it computes the `keccak256(txHash || logIndex)` idempotency key, skips already-processed keys (checked on-chain), and calls `vault.bridge(amount, key)`.

Reliability properties:

- Logs are scanned only up to the Ethereum `finalized` **block tag** (fail-closed if the RPC cannot report finality) — a reorg cannot re-bridge a deposit.
- Per-vault `lastProcessedBlock` checkpoints persist to a state file; on restart the listener replays since the checkpoint, and the vault's on-chain `processedTransfers` mapping makes the replay harmless.
- After the bridge confirms, minting is entirely the xReserve **V2 relayer's** job (the mint flag rides in the vault's immutable hookData) — there is no backend mint step that can fail or need retrying. End-to-end latency from Kraken withdrawal to visible USDCx on Aleo is **~3–4 minutes**.

10.2 Off-ramp: nightly sweep pipeline

`hl-offramp-sweep` runs on a nightly cron and pays out every eligible merchant **strictly sequentially** — one merchant's full pipeline completes before the next starts.

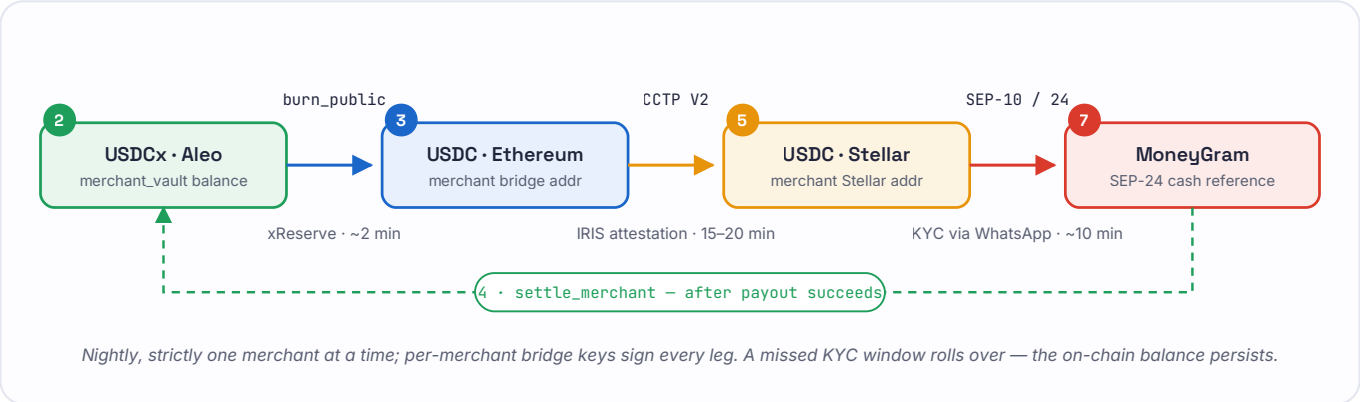


Figure 5 — Off-ramp: merchant balance to cash. The nightly sweep, one merchant at a time. Note the dashed return edge: the on-chain settle_merchant ledger update runs only after the MoneyGram payout succeeds — if anything fails mid-pipeline, the chain still says the merchant is owed, and the next run picks them up.

Step-by-step:

Step	What happens	Typical time
2	Aleo `burn_public` of the merchant's USDCx → emits a CCTP message for Ethereum	~2 min
3	Circle CCTP V2 `depositForBurnWithHook` Ethereum → Stellar (IRIS attestation + `mint_and_forward`)	~15–20 min
5/7	MoneyGram **SEP-24** per merchant: SEP-10 auth (merchant's Stellar key), KYC URL sent via WhatsApp, poll, payout	~10 min per merchant
4	`settle_merchant` on the Aleo aid program (clears the `merchant_vault` ledger) — runs **after** the payout succeeds	~2 min

Design decisions to understand:

- **Settlement after payout.** The on-chain ledger is only decremented once cash is actually claimable — if anything fails mid-pipeline, the merchant's on-chain balance still says they are owed, and the next nightly run picks them up.
- **Per-merchant bridge keypairs** (Aleo/EVM/Stellar) sign each leg — no shared pool, blast radius scoped per merchant.
- **KYC window with rollover.** Each merchant has ~10 minutes to complete the MoneyGram KYC on their phone; missing it simply rolls the merchant to the next night (the `merchant_vault` balance persists on-chain). Because the KYC link arrives on WhatsApp, cron timing is an operational parameter — schedule the run for an hour when merchants are awake in their timezone.
- **Crash-resume.** Per-merchant state files record the pipeline step; an operator can resume one merchant from the exact failed step (`--resume`, `--from-step N`), and steps 2–4 are idempotent across re-runs because Aleo state is the source of truth. A partial MoneyGram payout cannot double-pay because each SEP-24 transaction is unique.
- **FX presentation.** This anchor is SEP-24 only (no SEP-38 quote server). The KYC message shows a Chainlink-backed **estimate** (mid-market USD→COP, Colombia NGOs); the authoritative `amount_out` — MoneyGram's own FX rate and fee — arrives in the later cash-reference message once the SEP-24 transaction quotes.
- The merchant then collects cash at any MoneyGram location with the reference number and required ID.

10.3 Why three chains?

The chain hops each solve one problem:

- **Aleo** provides the privacy (there is no private CVA without it) — but has no direct fiat rails.
- **Ethereum** is where Circle's xReserve redeems USDCx back into canonical USDC.
- **Stellar** is the chain MoneyGram's anchor operates on (SEP-10/SEP-24 are Stellar ecosystem protocols), and Circle's CCTP V2 provides the attested Ethereum→Stellar transfer.

Every hop uses Circle-attested burn-and-mint (no wrapped-asset bridges, no liquidity pools), so bridge risk reduces to Circle's attestation service.

11. Identity, authentication and RBAC

The platform authenticates four very different kinds of caller, each with its own mechanism:

Caller	Mechanism
NGO staff (dashboard)	**Auth0** OIDC session → BFF proxy → API access token (JWT)
Merchant POS device	**Device token** (bearer; HMAC-hashed at rest when the pepper is set)
Off-ramp orchestrator	**Service token** (`X-Service-Token`) + an explicit `X-NGO-ID` tenant header
Beneficiary	**Possession factors**: the enrolled WhatsApp number (Twilio-signature-verified webhook), single-use HMAC tokens on public pages

11.1 Auth0 claims

Role and tenant travel as **namespaced custom claims** (`https://humanitylink.org/role` , `…/ngo_id`), copied from the user's Auth0 `app_metadata` onto both the ID token (read by the dashboard) and the access token (read by the API) by a post-login Action. Namespacing is required — Auth0 strips non-namespaced custom claims. A logged-in user with no role claim has **zero permissions**.

11.2 The role model

Four **cumulative** roles (higher rank ⊇ lower ranks):

Role	Rank	Adds
`viewer`	0	Read-only dashboards, lists, exports
`ngo-basic-admin`	1	Register merchants & beneficiaries, issue aid
`ngo-super-admin`	2	Whitelist/revoke merchants, pause/unpause, sweep vault, settlement config, goods categories, bridge address, send credits, settle merchant, soft-delete beneficiary
`hl-deployer`	3	Platform-global: authorize issuers, fund program, bridge USDCx, reconcile stuck ops, read bridge keys

Note how the ranks encode the operating model of §2: rank 1 = the NGO's *basic admin* (register + seed + issue), rank 2 = its *super admin* (approve + whitelist + controls), rank 3 = the *HumanityLink platform*.

The permission→minimum-rank table is the single source of truth, deliberately **duplicated byte-parallel in two files** — the backend's `roles.js` (enforced by `requirePermission()` middleware; returns `403 forbidden` below rank) and the dashboard's `roles.ts` (drives which controls render). The backend is the real gate; the

dashboard's hide/disable logic is UX only. If you add a permission, you change both files, add the middleware to the route, and wrap the UI control.

11.3 The service-token path

The off-ramp orchestrator presents `X-Service-Token` and carries no JWT; it is treated as full-access (`hl-deployer` tier). Because it has no token claim to carry a tenant, it must send an explicit `X-NGO-ID` header — trusted *only* on this already-trusted path, and failing closed (`400`) if absent. Treat the service token like an admin password: distinct per environment, TLS, IP allowlisting.

12. Multi-tenancy: the per-NGO key and program model

This is the platform's most consequential architectural migration and deserves its own section: the backend went from a single global set of signing keys and program IDs in `.env` to **everything per-NGO in the `ngos` table**.

12.1 What lives on an NGO row

Column	Contents	Secret
`aleo_private_key`	NGO's Aleo admin/signing key	yes
`evm_private_key`	Vault operator / funder key on Ethereum	yes
`stellar_secret`	Stellar funder secret	yes
`orchestrator_private_key` / `orchestrator_view_key`	Orchestrator keys	yes
`aleo_admin_address`, `evm_address`, `stellar_address`, `bridge_address`, `orchestrator_address`, `evm_vault_address`	On-chain addresses	
`aleo_program_id`, `aleo_program_address`, `bulk_issue_program_id`	The NGO's deployed program instances	

A unique index enforces **one program per NGO** — two NGOs can never share an `aleo_program_id`.

12.2 Resolution rules

- Every signing path resolves keys **per request** from the NGO the operation belongs to: routes derive it from the token claim; automated services derive it from their data (a merchant's `ngo_id`, a vault's NGO row); admin routes use a `loadNgoSigning` middleware; the orchestrator names it explicitly via `X-NGO-ID`.
- Paths that are signed by a *beneficiary's or merchant's* key (spend, merchant off-ramp) still need to know **which program** to call — they use the non-secret `getProgramIds` accessor.
- The Aleo transition executor **requires** an explicit program name on every call; there is no module-level default to fall back to, so a caller that forgets fails loudly instead of silently hitting the wrong tenant's program.

- Cross-tenant guards fail closed: a spend where the merchant and beneficiary have different (or missing) `ngo_id`s is rejected with a privacy-preserving 404.

12.3 What deliberately stays in env

Platform infrastructure that is genuinely tenant-independent: RPC URLs, the xReserve/USDC/CCTP contract addresses, Aleo network endpoints, the database URL, Auth0, Twilio, GCS, ProofIt, and the service token. The rule of thumb: **if two NGOs could ever need different values, it belongs in the database.**

13. Security model

13.1 On-chain controls (cannot be bypassed by any backend bug)

- Merchant whitelist gate on every spend/buy/off-ramp; per-merchant bridge-address binding; off-ramp minimum + bounded cooldown; pause switch; issuer authorization on all minting; bulk-issuance `request_id` nullifier; vault idempotency keys; immutable vault recipient; non-upgradeable programs with brick-proof admin transfer.

13.2 Backend controls

- **Auth on every money route.** POS money routes carry device-token middleware at the route level (with service-layer verification retained as defense-in-depth); admin routes stack mount-level auth + per-route RBAC + per-NGO signing context.
- **Rate limiting** on the unauthenticated, abuse-prone endpoints: POS credential verification (device-token brute-force) and face-verification (which also bounds third-party ProofIt cost).
- **On-chain re-reads**: the merchant whitelist in the DB is a cache that self-heals from the chain and never authorizes money by itself.
- **In-doubt handling**: broadcast-but-unconfirmed transactions are parked, never blindly retried; the reconciler resolves them (auto where a txId exists, manually otherwise), tenant-scoped for dashboard callers.
- **Webhook integrity**: Twilio signatures are validated in production, and the app **refuses to boot** with validation disabled — a forged webhook could fake a beneficiary's `YES` approval and drain funds.
- **HMAC-token gating** on public pages (beneficiary QR view page, P2P receive tokens): possession of a guessed ID is useless without the server-side secret; rotating the secret invalidates all outstanding links.
- **CORS allowlist**, TLS-verified database connections (with private-CA support), redacted connection strings in logs.

13.3 Secrets at rest — current posture

This is stated plainly because it is the most important operational caveat in the system today:

> **Custody keys (beneficiary/merchant/per-NGO signing keys) are envelope-encrypted in the database.** Each key is sealed with AES-256-GCM (locally) or GCP KMS, with the AEAD additionally bound to the row's identity (`short_id | aleo_address | ngo_id`) — so ciphertext cannot be transplanted onto another row or copied across environments, and keys are decrypted in-process only for the moment of signing. POS **device tokens are HMAC-hashed** with a stable pepper; the usable token is shown exactly once at create/regenerate. Operational invariants: **both backends must carry the same key-encryption key (or KMS key)**, since one encrypts and the other decrypts; the pepper must stay stable forever; and ciphertext is environment-bound — never copy encrypted rows between databases. Local/dev databases without the key configured fall back to plaintext, so treat any dev DB dump as sensitive.

Related key-risk scoping worth knowing:

- The **per-NGO Aleo signing key** is the single point of failure per tenant — rotate periodically.
- A compromised **EVM operator key** can trigger bridging of a vault's balance but cannot redirect it (immutable recipient).

- **Per-merchant bridge keys** scope compromise to that one merchant's in-flight balance.
- The shared **Stellar SEP-10 auth key** affects MoneyGram authentication only, never the payment source.
- There is **no unauthenticated mint API**: the V2 mint flag is baked into on-chain hookData.

13.4 Frontend/build hygiene

- No secret may ever be a `NEXT_PUBLIC_*` variable (it would ship in the public JS bundle); a CI release gate scans the compiled bundle for Aleo key patterns and fails the image on a hit.
- Env files are excluded from Docker images; secrets are injected at runtime.
- Dev-only escape hatches (role/tenant override headers, mock mode, webhook-validation skip) are ignored or refused in production.
- CVE-specific compensating controls are documented inline with their removal condition (e.g. the WebSocket-upgrade drop shim, removable once Next.js is upgraded past the fixed release).

13.5 Data protection in the field

The technical privacy model is completed by operational rules: never photograph beneficiary QR codes on personal devices; merchants see amount and approval only, never identity; use consent language about what data is collected and why; document incidents with **reference numbers and issue categories** instead of personal details; and route fraud/coercion/exploitation concerns through the agreed safeguarding channel immediately.

14. Field operations: workflows, monitoring and escalation

Developers should understand the operational choreography their code supports — most support requests arrive framed in these terms.

14.1 Beneficiary journey

1. **Register** — send the programme registration code on WhatsApp, then `hi`; answer the verification questions (household representative, name, date of birth, document photo if required).
2. **Receive the QR wallet** — as a WhatsApp image or a printed sticker; save it, protect it like cash.
3. **Spend / cash out** — only at approved merchants: show QR → merchant scans → purpose (goods or cash) → amount confirmed → wait for approval → goods/cash handed over → keep the confirmation.
4. **Self-service** — `BALANCE`, `QR` (re-fetch), `STATUS`, `REPORT` over WhatsApp.
5. **Report fast** — lost QR, wrong deduction, shop refusal, extra fee, failed transaction.

A lost QR/phone is handled as: verify the beneficiary, **deactivate the old access, reissue safely** — possible precisely because the wallet is custodial.

14.2 Merchant journey

Assess (location, reputation, stock, connectivity, smartphone, cash liquidity for cash-outs) → **whitelist** (platform + on-chain, by an NGO Admin) → install the POS app and register the shop → train (scan, amount entry, waiting for approval, history) → monitor (extra fees, refusals, liquidity pressure, wrong amounts, app issues).

14.3 Admin responsibilities

Programme setup confirmation; reviewing **seeded wallets and approving only correct records** (the pending queue); merchant whitelist management including suspension; account corrections via documented procedure; monitoring activity, failed transactions, pending queues, complaints and reconciliation; and being the single escalation route to HumanityLink with complete issue details.

14.4 Escalation model and issue playbook

Three tiers: **NGO field staff first** (re-explain, retry safely, check the merchant list, log with references) → **NGO super-admin controls** (approve/hold wallets, update accounts, suspend merchants, review disputes) → **HumanityLink technical** (platform, WhatsApp, QR, POS, dashboard, transaction, credit or off-ramp internals).

The standard decision table (each row maps to a mechanism described earlier in this document):

Symptom	First response	Underlying mechanism
QR will not scan	Brightness/lighting/retry, then escalate	POS camera + QR payload
Payment declined	Check balance/status + whitelist; **no goods/cash**	On-chain whitelist gate, balance check
Seeded but inactive	Admin reviews pending queue, approves or holds	Seeding/approval separation
Wrong amount (pre-confirm)	Cancel and restart	POS flow
Wrong amount (post-confirm)	Collect the transaction reference, escalate	Tx history + reconciliation
Lost QR/sticker	Verify identity, deactivate old access, reissue	Custodial reissue
Merchant not whitelisted	Admin resolves whitelist	On-chain `whitelist_merchant`
No MoneyGram reference	Check merchant WhatsApp + off-ramp status, escalate with details	Nightly pipeline state, WhatsApp delivery
Extra-fee report	Record, verify, warn/suspend	Whitelist revoke
Dashboard mismatch	Review exports, escalate	Stats vs. on-chain counters
Safeguarding concern	Agreed protection route, immediately	Out-of-band, by design

Every escalation should carry: beneficiary/merchant reference, issue category, transaction reference if applicable, date/time/location, what was already tried, owner, and status — the shape of the support-ticket pipeline.

15. Networks and deployments

Testnet and mainnet are **fully separate deployments that never share state**:

	Testnet	Mainnet
Backend API	`api.humanity.link`	`mainnet.api.humanity.link`
Database	own PostgreSQL DB	own PostgreSQL DB
WhatsApp	dedicated Twilio number	dedicated Twilio number
Aid token	`test_usdcx_stablecoin.aleo`	`usdcx_stablecoin.aleo`
Ethereum / Stellar / MoneyGram	Sepolia / Stellar testnet / MoneyGram sandbox	Mainnet / Pubnet / production anchor

- Backends select their network at deploy time via a single `.env` toggle block (SHARED + one active network block) and **fail closed** on any mixed-network configuration — the startup self-check treats `ALEO_NETWORK` as canonical and refuses to boot if any endpoint, bridge domain, xReserve contract anchor, or stablecoin program disagrees.
- Circle assigns one xReserve/CCTP domain ID per chain (identical across testnet/mainnet), so networks are distinguished by the source-chain contracts and Aleo programs — never by the domain.
- Next.js apps split by build-time env file; the mobile POS toggles by swapping the backend host on the activation screen.
- Accounts are per-network: a WhatsApp registration mints a fresh Aleo keypair per network, so the same phone can be a different beneficiary — or unregistered — on each.
- The mainnet cutover for the Leo programs is a controlled change: swap the stablecoin import (`test_usdcx_stablecoin.aleo` → `usdcx_stablecoin.aleo`) and the corresponding type references, and redeploy (the programs are non-upgradeable).

16. Glossary

Term	Meaning
Aleo	L1 blockchain with programmable privacy via zero-knowledge proofs
Leo	Aleo's programming language (this project: Leo 4.0)
USDCx	USDC-backed stablecoin on Aleo; the aid denomination
Record	Aleo's private state unit — an encrypted, UTXO-style object owned by an address; consumed whole on spend and replaced by new records (recipient + change), never edited in place
View key	Aleo key that decrypts (but cannot spend) an account's records; backend-only secret used for NGO reporting
Join	Merging several USDCx records into one (itself a proven on-chain transaction); run continuously by the `hl-auto-join` server to cap record fragmentation
Finalize	The public part of a Leo transition, executed on-chain (mappings, assertions, counters)
Delegated proving	Outsourcing ZK proof generation to the Provable API instead of proving on-device
xReserve	Circle's USDC ⇄ USDCx bridge between Ethereum and Aleo (V2: relayer auto-mint via hookData)
CCTP V2	Circle's Cross-Chain Transfer Protocol; here Ethereum → Stellar burn-and-mint with IRIS attestation
SEP-10 / SEP-24	Stellar ecosystem protocols for anchor authentication and interactive deposits/withdrawals; MoneyGram's cash-out interface
HL-ID	A user's short platform ID (`HL-000123` beneficiary, `MR-000004` merchant)
Seeding	Registering and funding-preparing a beneficiary wallet (basic-admin action, before super-admin approval)
Whitelisting	Admin approval of a merchant, enforced on-chain — precondition for receiving any payment
merchant_vault	On-chain per-merchant pending-payout ledger; incremented on off-ramp, cleared by `settle_merchant` after fiat payout
Off-ramp	Converting on-chain value to cash: beneficiary cash-out at a merchant, or merchant conversion via MoneyGram
In-doubt operation	A value-moving transaction that was broadcast but not confirmed in time; parked for the reconciler, never auto-retried
Envelope encryption	Encrypting each custody key with a data key wrapped by a master key (local AES-256-GCM or GCP KMS), AAD-bound to row identity
Device token	Bearer credential binding a POS installation to a merchant
Service token	Bearer credential for trusted internal services (the off-ramp orchestrator)
BFF proxy	Backend-for-Frontend: the dashboard's server-side proxy that attaches the Auth0 token so it never reaches the browser
RBAC	Role-based access control; four cumulative roles from `viewer` to `hl-deployer`
Microcredits	Money base unit: 1 USDCx = 1,000,000 microcredits, handled as exact BigInt

End of first draft. Sections 5–10 track the current component documentation; keep this document in sync when contracts are redeployed, the RBAC permission map changes, or the mainnet cutover completes.