TensorCash Coordination Layer: Cosign Bridge, Native Scriptless Swaps, Nostr Bulletin Board, and the TensorSwap Cross-Chain HTLC

Imosuke Takakuni[1]

2 May 2026

ABSTRACT

A peer-to-peer chain that wants to host real financial contracts — atomic swaps, repos, forwards, cross-chain HTLCs — has to solve three problems that the chain itself is the wrong place to solve. It has to give two strangers a way to **find each other** without trusting a centralised matching engine. It has to give them a way to **negotiate and co-sign** complex transactions without exposing either party's wallet to the other's machine. And, when a trade has to settle on a different chain, it has to give them a way to **bridge** that other chain back to the same atomic-secret graph the native side already understands. TensorCash addresses these with a **Nostr-backed bulletin board** for discovery, a **co-sign bridge** for end-to-end-encrypted bilateral sessions, the **Fair-Sign adaptor-signature ceremony** for native scriptless atomic signing, and the **TensorSwap HTLC** Solidity contract for the EVM-side leg of any cross-chain swap. These pieces are integrated into the desktop wallet through a Qt page with six sub-tabs and surface to the holder as a single, coherent trading and contract-management experience.

This whitepaper specifies the coordination layer in full. We describe what each component is, why it has the shape it has, what it promises and refuses to do, and how the pieces compose into the native spot, repo, forward, cross-chain spot, governance, and discussion flows the wallet exposes. The treatment is technical and financial; we do not cite source paths. The reader who finishes this document should be able to reason about the security properties of a TensorCash trade end to end — from the Nostr offer that surfaced it, through the bridge session that negotiated it, into the on-chain settlement that finalised it — and to evaluate where the trust assumptions sit at every step.

1. "Imosuke Takakuni" is a pseudonym. Contact: takakuni@tensorcash.org

TABLE OF CONTENTS

I. Why a Coordination Layer

TensorCash settles real trades on its base chain. The native settlement infrastructure — spot-swap PSBTs, repo collateral vaults, and forward margin vaults — ultimately produces Bitcoin-style transactions that both parties have signed and that the chain validates against the applicable asset, script, and taproot rules.

None of that requires a third party. What it *does* require, before the settlement transaction can even be drafted, is that the two counterparties have already (a) found each other, (b) agreed on price and terms, (c) exchanged the very specific bytes — public keys, partial signatures, anchor commitments, KYC proofs — that the settlement transaction needs. A chain is a poor place to do any of these. Posting an offer on the chain costs miner fees and makes it visible to everyone; co-signing a multi-step ceremony on the chain would require thousands of confirmations of intermediate state. The right place is *next to* the chain: a co-ordination layer that produces, as its only output, the inputs to a settlement transaction.

The TensorCash coordination layer has four deliberate architectural commitments. First, **public discovery is on Nostr, not on a vendor-operated matching engine**. Offers are posted to a configurable list of public relays; anybody can read them, anybody can run their own relay, and the protocol gracefully tolerates relay churn. Second, **bilateral negotiation is end-to-end encrypted**, not visible to any relay or counterparty before consent: the moment a maker accepts a trade request, the two parties drop into a Noise-protocol session whose only published artefact is a short authentication string the humans verify out-of-band. Third, **native settlement stays scriptless whenever the trade can be expressed as one transaction**: a spot swap is an augmented PSBT signed by both wallets, not an escrow script with two unilateral claim paths. Fourth, **cross-chain settlement uses the same atomic-secret graph as native scriptless settlement**: an EVM leg locked into the TensorSwap HTLC unlocks under the same SHA-256 pre-image whose discrete log is revealed by the TensorCash-side Schnorr signature, so the two halves of a swap form a single cryptographic system rather than two independent agreements that have to be reconciled by an oracle.

The rest of this paper makes each of these commitments concrete.

II. The Bulletin Board: Public Discovery over Nostr

The bulletin board is the public face of the coordination layer. Its job is narrow: take a maker's stated willingness to trade, make it visible to potential takers, let those takers raise their hand, and hand control of the next step over to the cosign bridge. It does **not** match trades, escrow funds, custody anything, or guarantee delivery. It is a discovery medium and nothing more.

i. What an offer is

An **offer** is a structured record published by a maker to a configured set of Nostr relays. Each offer carries a UUID, a creation timestamp, an expiry timestamp (default 24 hours from creation), a Bitcoin-network compartment (`main`, `signet`, `testnet3`, `regtest`, or one of the TensorCash-specific chain identifiers), a state, and the trade terms themselves. There are six kinds of offer: *buy* and *sell* (a maker offering to exchange one asset for another at a stated price), *swap* (a maker offering to exchange specific assets at a stated rate), and three contract kinds — *repo*, *forward*, and *spot contract* — each of which carries a full pre-built contract payload that the taker can inspect before requesting the trade.

For the simple buy/sell/swap kinds, the offer states the asset being sent, the asset being received, the amount, the price, the accepted payment methods (bank transfer, cash, on-chain, lightning), the regions the maker is willing to deal with, and whether escrow is required. For the contract kinds, the offer instead carries a Base64-encoded contract payload — the exact JSON that `repo.propose`, `forward.propose`, or `spot.propose` produces from the wallet — together with pre-computed display fields (annual percentage rate, loan-to-value ratio, tenor in days) so that takers can filter and rank without parsing the raw payload. Both offer kinds optionally carry a **proof of funds**: an array of BIP-322 ownership signatures over the maker's UTXOs, which a taker can verify locally to confirm that the maker actually controls the assets they claim to control before initiating any session.

ii. The offer state machine

An offer exists in one of eight states. **Posted** is the initial state: the offer is visible on Nostr, it has not yet been requested by any taker, and any taker may submit a trade request against it. **Requested** is the state after exactly one taker has submitted a request; the offer is taken off the public list (the Nostr event is *not* deleted, but the maker's UI hides it) and the maker decides whether to accept. **Accepted** means the maker has accepted that taker's request and emitted an invite link via direct message; **Handshaking** means both parties are inside the cosign bridge session running the SPAKE2/Noise handshake; **Active** means the handshake has completed and the trade ceremony is in progress; **Completed**, **Cancelled**, and **Expired** are the three terminal states.

The state machine is enforced

only on the maker's side

. Nostr itself has no notion of "this offer is no longer accepting requests" — the only way to take an offer off the public board is for the maker to publish a deletion event (Nostr kind-5) referencing the original offer, or to wait for the 24-hour TTL. To prevent races where two takers both submit requests against the same offer, the maker's wallet rejects every trade request after the first accepted one with a deterministic error, and the rejected taker sees the offer as "Requested by someone else" rather than "Available". The maker can also explicitly cancel a Requested offer if they no longer want to deal with that taker, returning it to Posted.

iii. Per-network compartmentalisation

Every offer carries a **network** field that specifies which Bitcoin network it applies to. A wallet running on `main` queries Nostr only for offers tagged `main`, and the same is true for every other compartment. This is a strict isolation rule: an offer made on `signet` is invisible to a `main` wallet, and a wallet that switches networks sees a wholly different bulletin board. The motivation is operational — a regtest test should never matchbox into a mainnet flow — but the side effect is that each network gets its own fully-functional, fully-decentralised marketplace without any cross-talk.

iv. Nostr as the substrate

The bulletin board uses three Nostr facilities. **Public offers** are posted as parameterised replaceable events under kind `30078`, with a `d` tag of `tensorcash-offer-<offer-id>`, a `network` tag, an `offer_type` tag, asset tags, and optional contract-specific tags such as `contract_type`, `apr`, `ltv`, and `tenor_days`. The replaceable semantics let a maker update an in-flight offer (price, expiry, terms) without spamming the relay with new events. **Trade requests** travel as encrypted direct messages (NIP-04 today, with NIP-44 receive support to ease the future migration); the encrypted payload contains the offer ID, the taker's Nostr public key, an optional message, and an optional proof of funds. **Maker→taker invites** travel the same way: once the maker accepts, they generate an ephemeral cosign-bridge invite link and send it via DM; the link itself is single-use and expires in ten minutes. The invite link is *never* serialised to public list responses, regardless of whether the offer is in flight or terminal.

Each wallet generates and persists a Nostr identity at first run (the keys live under `~/.tensorcash/nostr_keys`, file-mode `0600`). The pubkey is a stable pseudonym used across all offers, requests, ballots, and discussion posts that wallet ever publishes. Critically, this pseudonym is **not bound to any on-chain identity**: a maker who wants to demonstrate that they control a specific UTXO does so via an explicit BIP-322 proof of funds carried in the offer, not by virtue of using a particular Nostr key.

v. Caches and refresh

Each wallet keeps an in-memory cache of offers (30-minute TTL, refreshed when the user opens or refocuses the trade board), an in-memory cache of trade requests (1-hour TTL), and a persistent sled-backed cache of governance proposals and discussion posts. The caches absorb relay flakiness — a single relay outage does not blank the trade board — and they provide a uniform query API that does not have to round-trip to Nostr on every UI tick. The 30-minute window is long enough that the trade board feels live but short enough that a stale offer (one whose maker has deleted the underlying event) is purged before a taker can attempt a request against it.

III. The Cosign Bridge: Bilateral Secure Sessions

The cosign bridge is the layer that actually lets two parties negotiate. It is a separate Rust binary — `cosign-bridge` — that the Qt wallet spawns as a long-running subprocess and talks to over stdio. The bridge maintains a pool of **sessions**, each of which is a bilateral end-to-end-encrypted channel between exactly two parties, identified by an opaque session ID and joined via a single-use **invite link**. Once the handshake on a session completes, the wallet on each side can `send` and `recv` arbitrary structured payloads through the session, and the bridge handles framing, encryption, rate-limiting, and recovery transparently.

i. The session lifecycle

A session begins with one party calling `init`. The init produces a fresh session ID, an invite link of the form `cosign:?r=<rendezvous>&t=<transport>#c=<invite-code>`, and a freshly-generated SPAKE2 password derived from the invite code. The invite link is what the maker sends to the taker over the Nostr DM; it is the only thing the taker needs to join the session. The taker calls `join(invite_link)`, which extracts the rendezvous address (a relay URL or an onion address depending on transport), the transport mode, and the invite code, and stores them on a fresh local session record.

At this point both parties have parallel session records but no shared key. Either party can now call `handshake_auto`, which executes the entire SPAKE2-then-Noise handshake automatically over the chosen transport and returns a **short authentication string** (SAS) for human verification. The SAS is a sequence of five words drawn from the EFF short wordlist, with a six-digit numeric variant for accessibility, derived from the Noise handshake hash so that the same hash on both sides yields the same SAS and any active man-in-the-middle yields a different one. The two humans compare SAS values out-of-band — voice call, in-person, signed message — and confirm a match before sending any sensitive material.

Once SAS verification succeeds, the session is **handshake-complete**. The wallet can now call `send(payload)` and `recv()` repeatedly, exchanging structured JSON payloads encrypted under the Noise transport keys. Sessions have a default time-to-live of 30 minutes (configurable per session up to whatever the wallet wants), a per-session bandwidth cap of 5 MB, and a per-session message rate of 10 messages per second; these limits are defensive against runaway ceremonies and against a buggy wallet, not against a hostile peer (a hostile peer can always abandon the session and force the trade to abort). The bridge also keeps a 256-message, 5 MB recovery buffer so that a wallet that briefly disconnects can rejoin the session within a 20-minute recovery window without losing in-flight messages.

A session ends either with explicit `close` (which broadcasts a close signal and drops the session record), with TTL expiry (the bridge garbage-collects the session and refuses any further sends), or with abandonment (one party simply stops responding; the other side eventually gets a recv timeout and abandons too). The bridge persists session metadata across restarts via a sled-backed store, so a wallet crash mid-ceremony can resume the same session if both sides come back within the recovery window.

ii. SPAKE2: password-authenticated key exchange

The handshake is built on **SPAKE2**, a password-authenticated key exchange whose security guarantee is the right one for this setting. Two parties holding the same low-entropy secret (the invite code, in our case) derive a high-entropy shared key without ever transmitting the secret over the wire and without giving an active man-in-the-middle any computational advantage. An attacker who intercepts both SPAKE2 messages and tries to substitute their own learns nothing about the password and cannot derive the shared key, even with unbounded computation. This property is what lets us keep the handshake automatic over any transport — including transports we explicitly do not trust, like a public WebSocket relay we are about to discuss — without weakening the security model.

SPAKE2 alone, however, is not enough. An adversary who substitutes their own SPAKE2 messages cannot derive the legitimate shared key, but they

can

derive a different shared key with each victim, sit between them, and present each victim a different SAS. This is the exact attack the SAS verification is designed to detect. The SAS is derived from the Noise handshake hash — which mixes in both parties' SPAKE2 messages and the eventual shared key — so the SAS that the maker sees and the SAS that the taker sees will match if and only if both sides agreed on the same shared key. The humans comparing SAS are the final, irreducible step in the bridge's security; nothing else replaces it.

iii. Noise: encrypted transport with forward secrecy

Once SPAKE2 has produced a shared secret, the bridge initialises a **Noise protocol** session in the `Noise_NNpsk0_25519_ChaChaPoly_BLAKE2b` pattern, with the SPAKE2 secret used as the pre-shared key (PSK). The pattern choice is deliberate: `NN` means neither party authenticates with a static key (we don't need them to — the SAS is the authentication), `psk0` means the PSK is mixed in at the beginning of the handshake, `25519` is the Diffie-Hellman group, `ChaChaPoly` is the AEAD, and `BLAKE2b` is the hash. The PSK itself is not the SPAKE2 secret directly but its HKDF-SHA256 derivation, so that any subtle structure in SPAKE2's output is washed out before it reaches the Noise hash chain.

Once Noise is initialised, every subsequent send is a fresh authenticated-encrypted message under the running Noise transport keys, with per-message nonces, automatic forward-secrecy, and a 16-byte authentication tag. The bridge transparently frames each message into a length-prefixed JSON envelope and applies bucket-padding (256, 512, or 1024 bytes) so that an observer who can see ciphertext lengths cannot trivially fingerprint the protocol step. Replay protection is handled at two layers: Noise's own nonce sequencing rejects any replayed transport message, and a frame-level timestamp bounds replays to a ±120-second window in case a future protocol step needs to deduplicate at the application layer.

iv. What the bridge refuses to do

The bridge is deliberately small. It does not custody funds, it does not sign Bitcoin transactions, it does not understand asset tags, it does not know what a forward contract is. Its only responsibility is to produce a clean encrypted channel between two consented humans and to hand them whatever bytes they tell it to hand each other: offer payloads, acceptance payloads, PSBTs, nonce material, adaptor pre-signatures, final-signature commitments, and secret-reveal messages are all opaque bridge payloads. Settlement logic — building PSBTs, computing covenants, generating ZK proofs, dispatching to chain RPCs — lives in the wallet. The separation is what makes the bridge auditable: under 5,000 lines of Rust, a single security-critical primitive (PAKE+Noise), and no transaction-handling logic to mis-handle.

IV. Native Scriptless Join Swaps and Fair-Sign

The native spot path is intentionally simpler than a traditional cross-chain HTLC. If Alice and Bob are exchanging TensorCash-native assets, the atomic object is one transaction. Alice's wallet creates a spot offer with two legs, the asset or native amount Alice will deliver and the asset or native amount Bob will deliver. Bob imports the offer, accepts after reviewing the terms, and sends back an acceptance that fixes his receive address and his Fair-Sign adaptor point. Alice imports that acceptance. At this point both wallets have the same economic contract: two parties, two delivery outputs, one fee policy, one contract metadata hash, and two public adaptor points.

The transaction is built by **augmentation**, not by independent final transactions. The maker builds a base PSBT containing her own delivery output and her own funding inputs. The taker verifies that base PSBT contains exactly the expected maker delivery output, then augments it with his delivery output, his funding inputs, and any required change or asset-proof outputs. The returned PSBT is already the joined transaction; a separate `joinpsbts` step is not part of the normal spot execution path. For asset legs, the wallet uses its asset-send skeleton mode while building the leg, so asset TLVs, key-wrap commitments, holder-only proofs, change AssetTags, and fee-reservation outputs are produced by the wallet that actually controls that asset. This matters because the counterparty cannot safely invent another holder's compliance proofs or asset metadata.

Atomicity follows from ordinary transaction semantics before any scriptless machinery is considered. The final transaction spends both parties' inputs and creates both parties' delivery outputs. If either party withholds a required signature, the transaction is invalid and nothing settles. If both parties sign, the transaction can be broadcast once and the chain either accepts the whole exchange or rejects the whole exchange. There is no half-fill, no escrow account, and no on-chain negotiation state. Native asset swaps therefore pay ordinary transaction fees by virtual byte weight, including the weight of asset TLVs and proof outputs, but they do not add HTLC branches merely to simulate atomicity.

Fair-Sign is the scriptless signing layer used when the settlement transaction must also carry an atomic secret, or when a contract ceremony wants lock-step disclosure instead of plain `walletprocesspsbt` signing. The offerer and acceptor each generate a 32-byte adaptor secret `t` and publish only its x-only point `T = tG` in the offer or acceptance. The secret itself never leaves the wallet until the final reveal step. The PSBT carries a global contract metadata field that lets the wallet resolve the offer and acceptance records and recover the two public adaptor points. Each participating wallet then chooses the adaptor point for which it locally knows the discrete log and uses that point consistently for all wallet-controlled inputs in the ceremony; nonces remain unique per input.

The signing path is a four-phase ceremony over the already-authenticated cosign session.

1. **Prepare.** Each wallet validates that the PSBT still matches the contract metadata, refuses foreign unsigned non-Taproot inputs unless the user explicitly overrides the safety check, locks the wallet-controlled UTXOs for the ceremony, and computes the input sighash. For each wallet-controlled Taproot input, it samples fresh nonce entropy, derives a nonce `R`, adapts it to `R' = R + T`, retries until `R'` has BIP-340 even-y parity, and writes `fs/nonce_pub`, `fs/adaptor_point`, and `fs/commitment` into the PSBT. The commitment is `H_tag("fs/adaptor")(R' || T || Q || m)`, binding the adapted nonce, adaptor point, signing key, and message so a peer cannot substitute any of them after seeing a pre-signature.

2. **Partial.** After both sides have exchanged and merged the prepared PSBT material, each wallet computes the adaptor pre-signature `s' = k + e'·x`, where the BIP-340 challenge `e'` is deliberately computed over `R' || Q || m`, not over the base nonce `R`. The PSBT receives `fs/adaptor_sig = R' || s'`. The wallet verifies locally that `R + T` matches the committed `R'` before accepting the partial result, then wipes the nonce scalar needed for `s'` so it cannot be reused.
3. **Commit final.** Before either side reveals a final signature, each wallet computes what its final signature would be, hashes the 64-byte signature, and exchanges only that hash with the peer. This lock-step commitment step prevents the classic free-option attack where one party learns the other's completed signature, decides whether the market has moved in their favour, and withholds its own completion if not.
4. **Complete and extract.** A wallet that controls the adaptor secret completes its pre-signature by computing `s = s' + t`, producing an ordinary Schnorr signature `R' || s` that verifies under the Taproot key and looks like a normal signature on chain. The peer checks the final signature against the previously exchanged hash commitment. Once the transaction is finalised or observed on chain, the counterparty computes `t = s - s'` from the final signature and the earlier pre-signature. That extracted secret is the bridge between the native scriptless leg and any external HTLC claim.

The same pattern extends to cooperative Taproot script-path inputs when the PSBT carries the intended tapleaf material, and to MuSig2 inputs when the ceremony supplies the aggregate signer set. In the MuSig2 case, each signer exchanges public nonces, the aggregate nonce is adapted by the same point `T`, and the challenge is still computed against `R'`, which is what makes the aggregate final signature a standard BIP-340 signature after adding the adaptor secret.

The failure semantics are deliberately boring. Before final-signature reveal, an abort leaves no valid transaction; the wallets unlock the ceremony UTXOs after timeout or explicit cancellation and rebuild with fresh nonces if the users still want to trade. After a final signature is revealed, the party who saw it can extract the adaptor secret. At that point the system must assume the secret is public and advance the cross-chain state machine to claim or emergency-claim rather than trying to roll the trade back.

V. Transports: WebSocket Relay, Tor Hidden Service, Manual

The bridge supports three transports, chosen by the maker at session-init time. The transport choice is independent of the handshake and the encryption: the SPAKE2/Noise stack runs identically on top of all three, and a session's security guarantees do not depend on the transport's confidentiality. The transport's job is solely

reachability

— to deliver bytes from one party to the other — and the choice is between convenience, latency, censorship-resistance, and infrastructure dependence.

i. WebSocket relay

The default transport is a thin **WebSocket relay** server that the bridge connects to over `wss://`. The relay's job is to multiplex multiple sessions onto a single TCP connection by routing messages within named *rooms*; the rendezvous string in the invite link is the room ID. The relay sees ciphertext only — it has no view of the SPAKE2 password, the Noise handshake, the SAS, or any subsequent payload — and it forwards messages without inspection. A relay operator who is malicious can drop, delay, or duplicate messages, but they cannot read them and cannot inject anything that would survive Noise's authentication tag.

The trust assumption is small: the relay can deny service. That is explicitly the only thing it can do. We ship a reference implementation (`cosign-local-relay`) for self-hosting, and the wallet allows the user to point at any relay URL they trust or run themselves. Multiple relays can be used in parallel — a session that lost its connection can reconnect to the same relay, and the recovery buffer ensures no in-flight messages are lost; a session that wants to migrate can be re-established under a new invite link on a different relay. We do not run a "blessed" relay; the default invite text suggests one of several community-operated relays.

ii. Tor hidden service

The relay-free transport is a **Tor hidden service**: one party (the initiator) opens an onion service and publishes its `.onion` address as the rendezvous; the other party connects to that onion address through the local Tor daemon over SOCKS5. The session then runs over a direct end-to-end Tor circuit, with no relay in between. The transport setup is slower (typically 30 seconds to publish a hidden service) but the resulting session has stronger censorship-resistance properties: there is no relay to coerce, no relay operator's logs to subpoena, and no IP address exposed to the counterparty.

The Tor transport requires that both parties have a working Tor daemon (the bridge does not bundle one; it expects to find a SOCKS5 endpoint at `127.0.0.1:9050`). The handshake automation works exactly as on WebSocket — the same `handshake_auto` routine, the same SPAKE2/Noise stack, the same SAS verification — only the byte transport is different. The cross-chain dispatch flow, when it determines that a swap will involve external chain RPCs that should not be associated with the wallet's IP address, prefers Tor automatically.

iii. Manual transport

The third option is **no transport at all**: the bridge runs in `manual` mode, in which the SPAKE2 and Noise handshake messages are emitted as hex strings (or QR codes, via the wallet UI) and the user is expected to ferry them across by clipboard, QR scanner, NFC tap, file transfer, or any other out-of-band channel. This mode is intended for hardware-wallet integration, air-gapped signing, and for the case where two parties are physically co-located and would rather not depend on any network at all. Manual mode is somewhat tedious — the SPAKE2/Noise handshake requires four message exchanges, so there are four QR codes to scan in each direction — but it has the property that no third party of any kind has any view of the session.

The Qt wallet explicitly tells the user, at session-init time, which transport the invite link will use; an invite link generated in manual mode will not silently run over WebSocket if the taker happens to have a relay configured. The transport is part of the link.

VI. Discussion Threads: Stake-Gated, Pseudonymous Public Talk

Trading is not the only thing the bulletin board is good for. The same Nostr substrate hosts a separate, carefully scoped public-discussion facility — Nostr kind 8322, tag `tensorcash_discuss` — designed for two specific purposes: **model pre-alerts** (community discussion of an upcoming model registration before its on-chain deposit) and **model challenges** (discussion threads attached to a specific block at which a model's verification was challenged). The discussion facility is **stake-gated**: every post is required to carry a BIP-322 ownership proof over a TSC UTXO that the author controls, with the proof message binding the Nostr author key, the thread scope, and an explicit chain-height expiry.

i. Why stake-gated, why pseudonymous

A wholly anonymous, ungated discussion forum on a financial chain attracts spam by construction. A wholly identified one defeats the privacy properties of a pseudonymous wallet. The stake-gating compromise asks the author to prove they hold *some* TSC, leaves the holding amount and the address visible only to the verifier, and never publishes the wallet identity that produced the proof. A reader can verify a post by calling a single RPC against their local node; the verifier returns the actual UTXO value as `verified_units`, which the wallet uses to apply local rate-limit and minimum-stake filters (the default is 10,000 satoshis, configurable per user).

The proof is bound to the post in a way that prevents replay across threads, across authors, across networks, and across time. The canonical proof message is `TENSORCASH_DISCUSS:v1:<network>:<scope_type>:<scope_id>:<nostr_pubkey>:<expiry_height>`; the verifier checks that the network in the message matches the local chain, that the scope matches the thread being viewed, that the Nostr pubkey matches the post author, that the current chain height is below the stated expiry, that the UTXO referenced in the proof still exists and has at least one confirmation, that its address matches the proof's claimed address, and that the BIP-322 signature verifies. A proof that fails any of these ten checks is rejected; a post whose proof is rejected is hidden from the wallet's UI by default.

ii. Two scopes, two use cases

The first scope is **model_prealert**, keyed by the 64-hex `model_hash` of an upcoming model registration. A miner who is preparing to deposit a new model can announce the intent on the bulletin board, ask the community for ratifications or objections, and gather opinion before paying the on-chain deposit. The thread is an open discussion; the BIP-322 gate keeps it from becoming a sybil farm.

The second scope is **model_challenge**, keyed by the 64-hex `challenge_block_hash` of the block at which a model's inference verification was challenged. When a verifier produces a RED verdict on a model, the chain marks the model under challenge for a community-review window; the model_challenge thread is the public space where holders can post their independent re-verifications, dispute the verdict, or escalate to operator review. The thread is bound to the specific challenge block by hash, so a model that is challenged twice produces two independent threads.

The discussion facility deliberately does *not* bind the author to any on-chain miner identity. The Nostr pubkey is a pseudonym; the BIP-322 proof demonstrates that the author holds *some* TSC, not that they are the miner whose model is being discussed. Anyone who wants to identify themselves as the model's author does so by putting the model identifier in their post text and publicly signing for it — a social, off-protocol act, not a chain-enforced one.

VII. Governance Proposals and the Public/Private Flow

The bulletin board is also the public square for **asset governance**. When an issuer wants to rotate the mutable parts of an Issuer Control Unit — onboard a new compliance root, publish updated ICU contract text, change the quorum requirement, increment the policy epoch — they publish a **governance proposal** to Nostr, holders read the proposal, and (when the asset's quorum is non-zero) holders cast on-chain ballots that bind their AssetTag UTXOs to the proposal. The on-chain part of this flow is documented in the Asset Protocol whitepaper; the off-chain coordination is here.

i. Public flow

For non-sensitive proposals, the issuer publishes a **public** governance event with the proposal ID, the asset ID, the issuer's Nostr pubkey, the proposal title, the human-readable summary of policy changes, the new ICU TLV bytes, the canonical ICU hash, the template PSBT (which holders use to construct their ballot transactions), and a witness bundle (the issuer's BIP-322 proof of authority over the current ICU). Holders see the proposal in their governance tab, click a "View Details" button to read the full diff, click "Vote" to cast a ballot, and the wallet constructs and broadcasts the on-chain ballot transaction without any further Nostr round-trip. The whole flow is open; any observer can see the proposal, the diff, and the issuer's identity claim.

ii. Private flow

For sensitive proposals — for example, one that updates a regulatory compliance root that should not be publicly enumerable — the issuer publishes a **stripped** public proposal carrying only the metadata necessary to identify the asset and the proposal, plus the canonical ICU hash, the template PSBT hash, and the witness bundle hash. The actual ICU text, template PSBT, and witness bundle are *not* in the public event. A holder who wants to participate first sends an **AccessRequest** DM to the issuer, carrying their Nostr pubkey and a BIP-322 proof of holding at least one AssetTag of the relevant asset. The issuer verifies the proof, and replies with a **ProposalResponse** DM containing the full sensitive payload. The holder then casts the ballot through a third DM, and the issuer acknowledges with a **BallotReceipt** DM.

The four DM kinds form a chained envelope: each carries a sequence number (1 through 4), the proposal ID, a timestamp, an expiry, a previous-message hash linking it to the prior step in the conversation, and a JSON payload. The chaining is what gives the conversation integrity: the issuer's ProposalResponse references the holder's AccessRequest by hash, and any attempt by an attacker to replay either side of the conversation against a different proposal — or to splice a stale proposal into a new conversation — is rejected at the envelope-validation step. The bridge persists processed envelope hashes to a sled DB, so a replay attempted hours or days later still fails.

iii. What the chain still enforces

The off-chain governance flow is convenience, not security. The chain ultimately enforces every property that matters: a ballot transaction whose AssetTag does not actually carry the proposal hash is invalid; a rotation transaction whose ICU does not match the canonical hash is invalid; a quorum that does not actually meet the issuer's stated `policy_quorum_bps` is invalid. An issuer who tries to use the off-chain channel to commit to one proposal publicly and a different one privately is detected the moment the on-chain rotation is posted, because both are bound to the same `core_policy_commit` and `policy_epoch`. The Nostr layer is for human-readable disclosure and discoverability; the chain is the final arbiter.

VIII. The TensorSwap HTLC: EVM-Side Settlement Contract

For trades that cross between TensorCash and an EVM chain (today: Ethereum and Tron-EVM), the EVM side needs a counterparty contract that locks under the same hash as the TensorCash-side leg, releases on revelation of the same pre-image, and refunds after a timelock that is correctly ordered against the TensorCash leg's timelock. That contract is **TensorSwap HTLC**, a single Solidity contract deployed at a stable address per target EVM chain.

i. State machine

A swap, in the contract's eyes, is one of four states. **Empty** is the slot a swap identifier sits in before anyone has deposited against it. **Locked** is the only live state: funds are held by the contract, the secret hash and the timelock are fixed, and either the recipient can claim or — after the timelock — the depositor can refund. **Claimed** and **Refunded** are terminal states. There is no path from Empty back to Empty (a swap identifier may be used at most once for the lifetime of the contract, regardless of outcome) and no path from a terminal state back to Locked. This is the consensus-enforced version of the financial promise: a swap, once initiated, will resolve in exactly one direction; the chain will not permit "I claimed, but actually I refunded".

The state transitions are gated by two preconditions, checked on every call. The lock paths reject any attempt to overwrite an existing swap; the claim and refund paths reject any attempt to act on a swap that is not currently locked. The combination eliminates the two classic HTLC pathologies — claim-after-refund and double-claim — at the level of the state machine itself, not at the level of the asset accounting.

ii. Native ETH and ERC-20 paths

The contract supports two asset classes. **Native ETH locks** are written by sending value with the lock call; the contract records the amount and pays it out, on claim or refund, via a low-level call. **ERC-20 token locks** are written by calling a token-specific lock that pulls the deposit through `transferFrom`, requiring the depositor to have pre-approved the contract for the deposit amount. Economically the two paths are identical — the contract holds the asset under the same SHA-256/timelock conditions — but they are handled separately because of the operational gap between native and token transfer semantics: native transfers can fail at the gas limit of the recipient's fallback; token transfers can fail at the token's allowance check or balance accounting. The contract translates each failure into a distinct, non-silent revert, so an off-chain monitor cannot be confused.

The reason for distinguishing them is settlement risk, not bookkeeping. A native-ETH lock sees the depositor's funds reach the contract atomically with the lock call; the deposit cannot exist in a half-state where the contract has logged the lock but the funds have not arrived. A token lock, by contrast, depends on the token contract's external `transferFrom`; the HTLC therefore commits its own state to the new swap *before* it pulls tokens, in strict checks-effects-interactions order, so that even a malicious token cannot drive the HTLC into a state where the lock is recorded but the funds were not actually moved. If `transferFrom` fails, the prior state write is reverted along with everything else.

iii. Adaptor-secret compatibility: SHA-256, not Keccak-256

The contract uses **SHA-256**, not Keccak-256, to verify the secret on claim. This is not a stylistic choice. The same secret has to satisfy two verifiers in a cross-chain swap: the EVM verifier on this contract, and the TensorCash-side script verifier. Bitcoin-fork script's hashing opcodes are SHA-256 (and HASH160, which factors through SHA-256). For the same 32-byte pre-image to unlock both legs, both legs must use the same hash function. Choosing Keccak-256 on the EVM side, as a casual Solidity port might, would force an off-chain bridge oracle that signs "this Keccak hash corresponds to this SHA-256 hash" — exactly the third-party dependency the HTLC was designed to remove.

The same SHA-256 choice is what makes the contract compatible with the Fair-Sign adaptor ceremony described above. The external HTLC commits to `SHA256(t)`, where `t` is the 32-byte adaptor secret whose public point `T = tG` was already bound into the TensorCash-side PSBT. When the TensorCash-side Schnorr signature is completed and published, the counterparty computes `t = s - s'` from the final

signature and the previously exchanged pre-signature. That same byte string claims the EVM HTLC. The cycle closes only because the EVM contract and the TensorCash signature path agree on the exact secret bytes and because the HTLC uses SHA-256 rather than Keccak-256.

iv. Caller-chosen, deterministic swap identifier

The swap identifier is supplied by the lock caller, not generated by the contract. This is intentional: the identifier is meant to be a deterministic hash of the *off-chain offer*, so that both counterparties know in advance which on-chain slot will hold the swap. The reference convention is to set it to `H(offer_id ‖ chain_id ‖ leg_index)` where `offer_id` is the bulletin-board offer ID. The contract does not enforce a structure on the identifier — only uniqueness — because doing so would couple the bridge to an off-chain identifier scheme that may evolve.

v. Anyone-can-claim, only-sender-can-refund

Once a swap is locked, **any address** can submit the claim transaction; the funds go to the stored recipient regardless of who paid the gas. This may be surprising to a reader thinking of HTLCs as authenticated, but it is the right semantics for a bridge. The recipient does not necessarily have ETH for gas at the moment they want to claim — they may have just crossed in from a TensorCash-side leg and have only their newly received ERC-20 — and forcing only-recipient claims would re-introduce a dependency on a relayer service. Anyone-can-claim eliminates that dependency: the recipient pays any third-party relayer in band, the relayer submits the claim, and the funds flow only to the stored recipient.

Refund is the symmetric case but with one critical difference: only the **original sender** can refund, and only after the timelock. Anyone-can-refund would let an attacker race the recipient's claim with the depositor's refund and steal the funds whenever the timelock had elapsed; only-sender-can-refund forces the contract to re-authenticate the depositor's address. The refund is also state-machine-gated: a refund attempt on a Claimed swap, or on an already-Refunded swap, reverts with a state error rather than with a balance error, so off-chain monitors cannot be confused about the swap's state.

vi. Timelock semantics

The timelock is an absolute Unix timestamp at which the depositor regains the right to refund. The contract enforces, at lock time, that this timestamp is strictly in the future relative to the EVM block timestamp; a timelock equal to the current time is rejected. After the timelock, refund is permitted at the exact second the timelock expires (`block.timestamp >= timelock`), which simplifies cross-chain timelock matching against the TensorCash side's `nLockTime`-style behaviour.

The cross-chain timelock ordering — the rule that the leg the secret-revealer has to claim first must expire *later* than the leg the secret-holder has to claim later — is **not enforced inside this contract**. It cannot be: the EVM contract has no view of the TSC-side timelock. The ordering is established at swap-construction time by both wallets, jointly, using the cross-chain dispatch state machine described in the next section, and the wallet refuses to sign a swap whose timelock ordering is unsafe.

vii. What the contract does not do

It is worth being explicit. There is no fee. There is no oracle. There is no admin. There is no upgrade path. There is no whitelist. There is no event other than `Locked`, `Claimed`, and `Refunded`. The contract is fewer than 250 lines of Solidity and is meant to remain so. Every additional surface — a fee, an admin, a pausable flag — would re-introduce a trust assumption the bridge was specifically designed to eliminate. If a future version needs different behaviour, it will be deployed as a new contract at a new address, and the wallet layer will route to whichever address is appropriate for the swap's terms.

IX. Cross-Chain Dispatch: How a Spot Offer Becomes a Two-Leg Swap

A cross-chain swap rides as a **spot contract offer** on the bulletin board, with a special schema marker (`cross_chain_spot_v1`) inside the offer's contract payload. The dispatcher inside the cosign bridge sniffs the schema on offer-acceptance; if it sees the marker, it routes the session to the cross-chain execution path instead of the native spot path. The architecture deliberately reuses the native trade infrastructure — the same offer format, the same bulletin board, the same cosign session — so that adding a new external chain is a matter of writing an *adapter*, not rebuilding the whole stack.

i. Adapters

The schema recognises three external adapter identifiers: `btc_scriptless_v1` for vanilla Bitcoin, `eth_htlc_v1` for the TensorSwap HTLC on Ethereum, and `tron_htlc_v1` for the TensorSwap HTLC on Tron-EVM. In the current source tree, the implemented external-chain transaction module is the Ethereum HTLC path; BTC scriptless and Tron support exist as schema, validation, and settlement-profile targets to be backed by adapter implementations. Each external adapter is meant to provide the same operations: build the lock transaction, watch for the lock confirmation, build the claim, build the refund, and notify the state machine of external events. The dispatch layer is wholly chain-agnostic; it only knows that there is a TSC leg and an external leg, and that the two legs are bound by a common SHA-256 secret.

The TensorCash side is not treated as an external adapter. It is the native signature adapter: build the augmented spot PSBT, attach contract metadata and Fair-Sign policy, run the adaptor prepare/partial/commit-final/complete sequence over cosign, broadcast the final TSC transaction, and extract the adaptor secret from the on-chain signature when the peer reveals it. For an EVM swap, the Ethereum adapter sees a normal HTLC lifecycle while the TensorCash adapter sees a scriptless signature lifecycle; the dispatcher joins those two views by carrying the same `secret_hash`, settlement profile, and offer ID through both.

ii. Settlement profile

The offer payload carries a **settlement profile** — confirmation thresholds for both chains, timeout policy, fee policy, fee-funding mode, and funding order — that fixes every economically material parameter before either party signs. The confirmation policy specifies the minimum number of confirmations required on each chain before the state machine will advance; six is typical for Bitcoin-fork chains, twelve to fifteen for Ethereum, twenty for Tron, with an additional `reorg_conf` setting that establishes the threshold at which a chain is considered safe against reorganisation. The timeout policy specifies the external-chain lock duration in seconds, the TSC-side lock duration in blocks, and a per-leg claim budget within which the claimer must broadcast and possibly fee-bump.

The fee policy specifies whether the claimer is expected to use replace-by-fee with child-pays-for-parent (RBF/CPFP) or a gas-escalator strategy, and the fee-funding mode specifies whether the fee comes from a reserved UTXO (BTC/TSC) or a reserved account balance (ETH/TRON). The funding order — `tsc_first` or `external_first` — fixes the order in which the two locks are placed; the order is one of the inputs to the timelock-ordering check, because the leg locked second must have the strictly-shorter timelock for the swap to be safe.

iii. State machine: gated secret revelation

The cross-chain state machine has eighteen states, but the only ones the user sees are the five lifecycle anchors: **Posted** (offer is on the board), **Matched** (taker has requested), **SessionEstablished** (cosign session is open and SAS-verified), **TermsFinalized** (both sides agreed on every parameter), and one of

the terminal three (**Completed**, **Refunded**, **Aborted**). Between these are the operational states the wallet uses to drive recovery and policy: **FundingPrepared**, **CounterpartyLockSeen**, **CounterpartyLockConfirmed**, **LocalLockConfirmed**, **ClaimReady**, **ClaimBroadcast**, **EmergencyClaim**, **ClaimConfirmed**, **RefundReady**, **RefundBroadcast**.

The most important transition in the entire system is the one that moves the state machine from **ClaimReady** to **ClaimBroadcast**: this is the transition at which the secret is **revealed on-chain**. On the EVM side, revelation is a claim call that carries the pre-image. On the TensorCash side, revelation is the publication of a completed Schnorr signature from which the peer can extract the adaptor secret. It is not an incidental side effect of "continue to next step"; it is a distinct, gated transition. Before claim broadcast, the swap can safely abort — both parties cancel, both refund. After claim broadcast, the secret is public, and the swap *cannot* abort; it must either complete normally or, if the counterparty leg has deteriorated (the counterparty refunded faster than expected, or a chain reorganised), enter **EmergencyClaim**, in which the wallet aggressively fee-bumps the local claim transaction to ensure it confirms before the counterparty can re-spend.

The state machine surfaces as a single, chain-agnostic lifecycle in the wallet's Active Contracts tab. A user looking at a TSC↔ETH swap sees the same column structure that the BTC and Tron profiles are designed to use: the dispatch hides the adapter-specific sub-state (mempool watching, gas escalation, RBF history) under the same five anchors.

X. Qt Integration: The Exchange P2P Page

All of the above surfaces in the desktop wallet through a single Qt page — **Exchange P2P** — with six sub-tabs in the current build: **Market**, **Book**, **Risk**, **Pricing**, **Structuring**, and **P2P Sessions**.

P2P Sessions lists every cosign session the wallet currently knows about: its session ID, its transport, its handshake status, its peer pseudonym, its SAS for verification, and a "Close" button. Sessions that have completed handshake show the SAS in both word-list and numeric form so the user can read either over the phone. Sessions that have expired show their TTL countdown. The tab also exposes manual handshake controls (for `transport: "manual"` sessions), a session-recovery button (for sessions that timed out but are within the recovery window), and the live ceremony step when a Fair-Sign swap is in progress: prepared nonces, merged PSBT, partial signatures, final-signature commitments, completion, extraction, and broadcast readiness.

Market is the bulletin-board surface. It shows offers from Nostr (filtered by the wallet's current network), grouped by offer kind, with sortable columns for asset, price, amount, APR/LTV/tenor (for contract offers), payment methods, and maker reputation. It exposes "Post Offer" and "Refresh" buttons, a filter dialog, a per-row "Request Trade" action that triggers a Nostr DM to the maker, and an "Accept Request" panel for makers to review and accept incoming requests. When a request is accepted, the tab automatically opens a P2P Session and walks the user through SAS verification before unlocking the trade ceremony.

Structuring is the contract-builder surface. It lets a holder build a fully-formed repo, forward, or spot contract proposal — selecting the underlying asset, the counterparty's role, the maturity, the haircut, the IM — and post it as a contract offer on the bulletin board. For native spot it also exposes the augmentation path: create the offer, import or collect the acceptance, build the maker base PSBT, let the taker augment with their own leg, and then either co-sign the joined PSBT directly or run the Fair-Sign adaptor ceremony over the active session. The builder validates every field locally (the asset must exist, the maturity must be in the future, the haircut must be within policy bounds) before allowing the post.

Book is the position-management surface. It is backed by the core wallet's contract RPCs: `contract.list` supplies the repo, spot, and forward book rows, and `contract.status` supplies the detail view with role, state, deadlines, associated UTXOs, closure metadata, confirmations, and vault script material. The tab surfaces the cross-chain state machine in chain-agnostic form for cross-chain swaps, exposes only the per-contract action set the current wallet role and height permit, and routes those actions back to the contract build RPCs for cooperative close, cash settlement, repayment, default sweep, escrow claim, escrow refund, or timeout. For contracts that are mid-ceremony, it shows the bridge session's status and a real-time view of which message exchange step the ceremony is in.

Risk is the portfolio-risk surface. It shows open positions and wallet balances, lets the user include or exclude positions from aggregation, and computes mark-to-market and per-asset/per-tenor risk metrics from the positions the wallet currently knows about.

Pricing is the reference-rate surface — a per-asset configuration of price feeds and reference rates that the contract builders draw on to sanity-check user-entered prices.

A separate **Governance** tab (under the Treasury page, not the Exchange P2P page) lists every governance proposal the wallet's bulletin-board cache has seen, surfaces the human-readable diff against the current ICU, validates the issuer's BIP-322 attestation locally, validates the canonical-ICU/template-PSBT/witness-bundle hashes against the public payload, and exposes a one-click "Vote" button that constructs and broadcasts the on-chain ballot transaction.

A separate **Discussion** UI lives inside the Models page — model_prealert and model_challenge threads attached to the model rows they describe — and inside the Mining APIs page, where pending model reviews can spawn a dedicated thread for community input.

The architectural pattern is consistent across all of these: the Qt UI calls bridge RPCs through a thin C++ wrapper (`BridgeSessionManager`), the bridge spawns and supervises the `cosign-bridge` binary as a child process, the bridge talks to Nostr/WebSocket/Tor, and the resulting payloads are handed back to Qt as structured QVariantMaps that the tabs render. There is no JavaScript layer, no HTTP server, no long-lived TCP connection from Qt itself; the wallet is a single Qt process and a single Rust subprocess, and they talk over stdio.

XI. Threat Model and Trust Assumptions

We close with a precise enumeration of what each component trusts, what it does not, and what it would take for each to fail.

The Nostr relays are trusted to deliver bytes; they are not trusted to keep secrets, validate offers, or enforce ordering. A malicious relay can drop, delay, duplicate, or selectively forward any event it sees, but it cannot forge a signed event under another author's key, and it cannot read direct messages (which are NIP-04 encrypted under the recipient's key). A relay that censors a maker's offer is detectable by the maker (they get no requests despite advertising) and routable around (post to a different relay; both maker and taker poll a configurable list).

The cosign bridge's WebSocket relay, when used, is trusted to deliver bytes and nothing more. It cannot read SPAKE2 or Noise messages, cannot recover the SAS, cannot read post-handshake payloads, cannot see SAS verification (which is out-of-band), and cannot modify any ciphertext without its modification being detected at Noise's authentication tag. Its only failure mode is denial-of-service.

The Tor network, when used as the transport, is trusted to provide reachability and a measure of network-layer privacy. The bridge does not trust Tor for cryptographic confidentiality; the SPAKE2/Noise stack runs on top regardless, so a Tor compromise reduces to a transport DoS in the worst case.

The TensorSwap HTLC's EVM execution environment is trusted to execute the contract honestly under the chain's consensus. The contract has no admin and no upgrade path, so a holder who has read the deployed bytecode can verify, once and for all, what the contract will do for any subsequent swap. The chain itself is trusted to confirm valid transactions and reject invalid ones; a 51% attack on the EVM chain that re-orgs around a confirmed claim is the only failure mode that can violate atomicity, and the cross-chain state machine's `reorg_conf` parameter is what bounds that risk per swap.

The Fair-Sign adaptor ceremony is trusted only for the standard Schnorr/adaptor-signature assumptions: nonce scalars must be unique, the PSBT message being signed must not mutate between prepare and complete, the adaptor point must be bound to the same message the pre-signature signs, and final signatures must be revealed lock-step. The wallet enforces these as protocol checks: it freshens nonce entropy on every prepare, binds `R'`, `T`, `Q`, and `m` in the PSBT commitment, rejects changed sighashes at completion, locks ceremony UTXOs while the ceremony is live, and requires final-signature commitments for manual completion. A counterparty can still abort, but before final reveal an abort produces no valid transaction and leaks no claimable secret.

The TensorCash core node is trusted, on each user's machine, to validate transactions correctly and to expose RPCs honestly. The wallet talks to its own core node, not to any remote node; the bridge talks to the wallet, not to the core node. An attacker who compromises the local core node has compromised the wallet — but no other party, and no remote service, has any view into the user's transactions or proofs that a local-node compromise would not already provide.

The counterparty is not trusted at all. The entire stack — bulletin board, cosign bridge, native covenant trees, EVM HTLC, cross-chain dispatch — is built on the assumption that the counterparty is potentially hostile and that every property the user cares about must hold under that assumption. The SAS verification is the single place where the user is asked to trust *something*: namely, that the human voice on the phone confirming the SAS is the human they think it is. Beyond that one out-of-band step, the protocol does not rely on counterparty honesty at any point.

The combination is what gives the coordination layer its character. It is not a venue, not a custodian, not a settlement guarantor, not a clearing house. It is the connective tissue between the Nostr network, the TensorCash chain, and any external chain a trade chooses to touch, letting counterparties briefly and consensually agree on a single swap.

This whitepaper accompanies the TensorCash Settlement Contracts whitepaper (native settlement contracts and the Tapscript extension), the TensorCash Asset Protocol whitepaper (asset TLV format, ICU registry, KYC verification), and the TensorCash Inference Verification whitepaper (the proof-of-inference consensus layer that secures the chain). The implementation lives in `services/core-node/cosign-bridge/` (Rust), `services/core-node/contracts/ethereum/` (Solidity), and `services/core-node/bcore/src/qt/` (C++/Qt).