CHAIN TICK

Whitepaper

www.chaintick.io

Verification-first trading AI with on-chain Proof-of-Performance (PoP)

Every signal is committed on-chain at broadcast and revealed later—so results are auditable, immutable, and trust-minimized.

Document Status & Audience

This whitepaper is a detailed, technical and product-complete description of CHAIN TICK. It is intended for traders, builders, security reviewers, and investors evaluating the product, token utility, and the Proof-of-Performance (PoP) mechanism.

Executive Summary

CHAIN TICK ingests multi-timeframe market data (5m/15m/1h), order-book microstructure, and price action to produce actionable LONG/SHORT trade plans (entry zone, TP ladder, SL, confidence). Each signal is cryptographically committed on-chain at T_0 and revealed at T_1, enabling anyone to verify the signal's existence and contents at broadcast time and to audit performance ex-post without relying on screenshots or centralized databases.

A utility token, CTICK, powers access tiers, usage credits, model marketplace staking, and a usage-driven buyback & burn. Product demand creates sustainable token sinks without promising financial returns.

Problem Statement

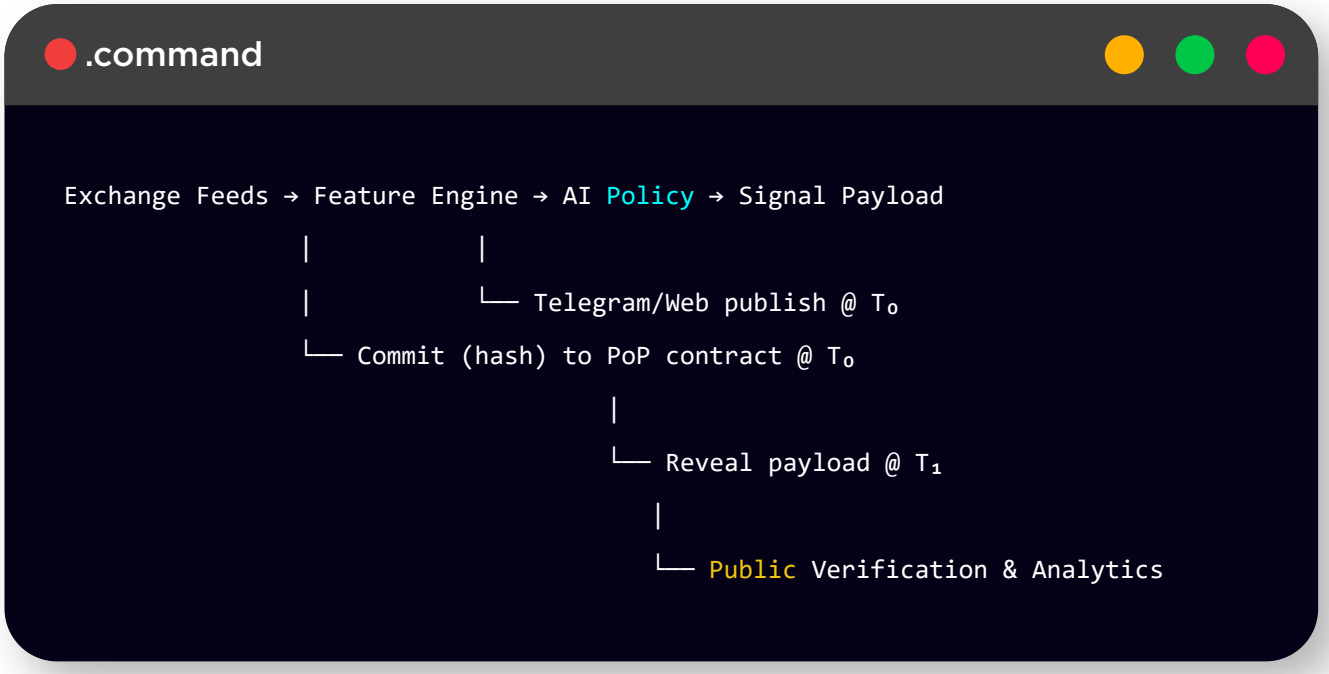
- The trading-signals market is plagued by hindsight bias and selective reporting.
- AI-powered claims are typically opaque and hard to audit.
- Users cannot independently verify when calls were made or whether they were altered.
- Lack of verifiable history erodes trust and impedes institutional adoption.

Goal: Deliver a verifiable signal platform where every live call is timestamped and sealed on-chain at broadcast and later revealed so anyone can verify it.

Solution Overview

Core idea: A commit-reveal protocol on-chain (PoP), plus an analytics stack that derives performance solely from revealed, verifiable signal history

Pipeline (high level)



Outcomes

Live signals with entries/TP/SL + rationale

Immutable, publicly verifiable history

- Trust-minimized leaderboards & analytics
- Tokenized access, credits, and marketplace alignmen

System Architecture

01. Components

- **Connectors:** Exchange WebSockets/REST (e.g., spot/perps), resilient reconnection, time-sync.
- **Feature Engine:** Real-time computation of order-book and trade features (delta, imbalance, micro-price tilt, spread dynamics, queue depth, absorption/sweeps, volatility regimes).
- **AI Policy & Rule Layer:** Rule-based microstructure signals blended with an AI layer to set entries, TP/SL, and confidence conditioned on regime.
- **Signal Orchestrator:** Finalizes payload, computes hash, schedules on-chain commit and human-readable publish, then queues reveal.
- **PoP Contract (on Arbitrum):** Stores commits (hashes and metadata), verifies reveals, and emits events.

Storage:

- **Hot path:** Redis/Memory cache for latest features;
- **Cold path:** ClickHouse (or Parquet on S3) for tick/OB history and revealed payloads;
- Payload archival: IPFS with on-chain CIDs (optional).
- Delivery: Telegram bot (admin in channel), Web dashboard/API, and webhooks.
- Verification SDK & Explorer: Open-source tools to validate commits/reveals.

02. Infra Notes

- **Chain:** Arbitrum (EVM; low fees, fast finality).
- **Clocks:** Chrony-based NTP; monotonic clock used for sequencing; block timestamps used for on-chain canonical time.
- **Resilience:** Multi-region failover for WS; Kafka/NATS streams for decoupling; idempotent processors.

Data & Modeling

01. Ingestion

- **Candles:** 5m, 15m, 1h (OHLCV).
- **Trades:** Tick-by-tick last price/size + aggressor side.
- **Order Book (L2):** Best bid/ask, depth ladders (top-N), diff/partial updates.

02. Features (examples)

- **Imbalance:** IMB=∑bidQty−∑askQty∑bidQty+∑askQty\text{IMB} = \frac{\sum \text{bidQty} - \sum \text{askQty}}{\sum \text{bidQty} + \sum \text{askQty}} (top-K levels)
- **Micro-price tilt:** Weighted top-of-book pressure estimator
- **Spread compression/expansion** signals
- **Absorption/Sweeps:** Flags for large market orders consuming depth; queue depletion
- **Volatility regimes:** ATR-based gates, realized vol
- **Structure:** Trend/mean-revert tags, regime filters

03. Decision Layer

- Rule detectors produce candidate directions and context tags.
- AI policy (LLM-assisted classifier/regressor) adapts: entry zone width, TP ladder, SL distance, and confidence.

Risk guardrails:

- Max simultaneous open positions
- Loss-streak cool-down
- Volatility and spread gates
- Latency & staleness checks

Signal Payload (canonical schema)

.command

```
{
  "ts"   : "2025-08-15T18:22:43Z",
 "pair" : "BTCUSDT",
  "tf"   : "5m",
  "dir" : "LONG",
"entry" : {
 "from" : 64250.0,
   "to" : 64290.0

 },
"tp": [
    64520.0,
    64780.0,
    65150.0

],
"sl" : 63940.0,
"model" : "ob-v3.2",
"confidence": 0.74,
"notes": "imbalance >30%, spread tight, buyer aggression rising",
"channel": "telegram://@chaintick",
"publisher": "0xPUB...KEYID",
"nonce": "0x7fd1c3...e9"
}
```

Proof-of-Performance (PoP): Commit → Reveal

01. Canonical Serialization

To ensure deterministic hashes, we use JSON Canonicalization Scheme (JCS, RFC 8785):

- UTF-8 encoding
- Lexicographically sorted keys
- No superfluous whitespace
- Numbers in ECMAScript Number format (no trailing zeros)
- No NaN/Infinity

Let C (payload) be the JCS-canonicalized UTF-8 bytes.

02. Hash Construction

We compute:

hash = keccak256(C(payload_without_nonce) II nonce)

- nonce is 32-byte cryptographically secure random (CSPRNG).
- payload_without_nonce excludes the nonce field to avoid self-reference.
- The publisher field identifies the logical signer (see 6.6).

Rationale:

Concatenating a secret nonce prevents dictionary attacks and payload guessing pre-reveal.

Excluding the nonce from canonicalization allows us to reveal the nonce alongside the payload

Single-Signal Commit

01. At broadcast T_0:

1. Build payload (without nonce), draw 32-byte nonce.
2. Compute hash.
3. Call commit(hash, meta) on PoP contract.
4. Simultaneously publish human-readable signal to Telegram/Web/API.

meta fields stored on-chain:

- pairId, timeframeId, modelId, publisher, seq, minRevealAt, maxRevealAt
  (IDs are compact uint16/uint32 indexes to reduce gas; times are Unix seconds.)

02. Batched Commit (Merkle)

- To reduce gas, multiple signals within a block/window are batched:
- Compute leaf hash for each signal as above.
- Build Merkle tree; root R = merkleRoot(leaves).
- commitBatch(R, meta) is stored on-chain.
- For reveal, provide (payload, nonce, leafHash, merkleProof); contract verifies leafHash R.

03. Reveal

At T_1 (post trade window and within policy window):

- Call reveal (payload, nonce) for single commit, or
- revealBatch (payload, nonce, leafHash, proof) for batched commit

04. On-chain checks:

- Recompute hash* = keccak256(C(payload_without_nonce) || nonce).
- For single: assert hash* == storedHash.
- For batch: verify Merkle proof to stored root.
- Assert block.timestamp [minRevealAt, maxRevealAt].
- Mark revealed; emit PayloadRevealed with optional **IPFS CID** or compact on-chain bytes.

05. Publisher Identity & Off-Chain Signature (optional but recommended)

To bind payloads to a recognized signing key, the orchestrator signs C(payload_without_nonce) off-chain using secp256k1; the signature is included in the reveal payload.

- Field: "sig":"0x...", "pub":"0x04..."
- Clients can check the signature off-chain; on-chain verification is optional (gas heavy), generally we store the publisher address in commits.

06. Time Windows & Anti-Withholding

- minRevealAt: prevents “instant reveal” that might leak private payloads prematurely (configurable; often 1–5 minutes).
- maxRevealAt: forces timely disclosure (e.g., within 2–8 hours after T_0).
- Missed Reveal: If not revealed by maxRevealAt, commit is flagged as Expired.
- Explorer shows Expired; performance analytics treat unrevealed commits as score-penalizing events to prevent “hiding losers”.

07. Reorg & Finality

- UI shows Pending (k) until k confirmations (e.g., k=12 L2 blocks).
- Commit timestamps use on-chain block.timestamp; explorer displays both block and wall-clock time.

08. Storage Modes

- Compact on-chain: Small payloads can be stored as bytes in the reveal event to maximize longevity.
- IPFS Anchoring: For larger payloads, store on IPFS; anchor CID on-chain.
- Redundancy: Periodic snapshot of revealed payloads to cold storage (S3/Glacier).

Smart Contracts (Solidity Sketch)

.command

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0; // version pinned in repo

import {MerkleProof} from "./MerkleProof.sol";

contract ChainTickPoP {
    struct CommitMeta {
        uint32 pairId;
        uint16 timeframeId;
        uint16 modelId;
        address publisher; // EOA or contract
        uint64 seq;        // per-publisher sequence
        uint40 minRevealAt; // unix seconds
        uint40 maxRevealAt; // unix seconds
    }
```

```
struct CommitRec {
        bytes32 hashOrRoot; // single hash or merkle root
        CommitMeta meta;
        bool isBatch;
        bool revealed; // for single; batch tracked per leaf via bitmap (optional
off-chain index)
    }

    event Committed(
        bytes32 indexed key,
        bytes32 hashOrRoot,
        CommitMeta meta,
        bool isBatch
    );

    event Revealed(
        bytes32 indexed key,
        bytes32 leafHash,
        bytes payloadCID_or_Bytes
    );

    mapping(bytes32 => CommitRec) public commits; // key = keccak(publisher, seq)

    function commit(bytes32 leafHash, CommitMeta calldata meta) external {
        bytes32 key = keccak256(abi.encodePacked(meta.publisher, meta.seq));
        require(commits[key].hashOrRoot == bytes32(0), "exists");

        commits[key] = CommitRec(leafHash, meta, false, false);
        emit Committed(key, leafHash, meta, false);
    }
```

```
function commitBatch(bytes32 root, CommitMeta calldata meta) external {
    bytes32 key = keccak256(abi.encodePacked(meta.publisher, meta.seq));
    require(commits[key].hashOrRoot == bytes32(0), "exists");

    commits[key] = CommitRec(root, meta, true, false);
    emit Committed(key, root, meta, true);
}

function revealSingle(
    CommitMeta calldata meta,
    bytes calldata canonicalPayloadNoNonce,
    bytes32 nonce,
    bytes calldata payloadBytesOrCID
) external {
    bytes32 key = keccak256(abi.encodePacked(meta.publisher, meta.seq));
    CommitRec storage rec = commits[key];

    require(!rec.isBatch, "batch");
    require(
        block.timestamp >= rec.meta.minRevealAt &&
        block.timestamp <= rec.meta.maxRevealAt,
        "window"
    );

    bytes32 computed = keccak256(
        abi.encodePacked(canonicalPayloadNoNonce, nonce)
    );
    require(computed == rec.hashOrRoot, "hash mismatch");
```


```
        rec.revealed = true;
        emit Revealed(key, computed, payloadBytesOrCID);
    }

    function revealLeaf(
        CommitMeta calldata meta,
        bytes calldata canonicalPayloadNoNonce,
        bytes32 nonce,
        bytes32 leafHash,
        bytes32[] calldata proof,
        bytes calldata payloadBytesOrCID
    ) external {
        bytes32 key = keccak256(abi.encodePacked(meta.publisher, meta.seq));
        CommitRec storage rec = commits[key];

        require(rec.isBatch, "single");
        require(
            block.timestamp >= rec.meta.minRevealAt &&
            block.timestamp <= rec.meta.maxRevealAt,
            "window"
        );

        bytes32 computed = keccak256(
            abi.encodePacked(canonicalPayloadNoNonce, nonce)
        );
        require(computed == leafHash, "leaf mismatch");
        require(MerkleProof.verify(proof, rec.hashOrRoot, leafHash), "bad
proof");

        emit Revealed(key, leafHash, payloadBytesOrCID);
    }
}
```

- **Notes:**
- canonicalPayloadNoNonce is the JCS bytes of payload without the nonce key.
- For gas, we avoid heavy on-chain storage of the full payload; we emit in events or store CIDs.
- Per-leaf reveal tracking can be managed off-chain (indexer) or via a bitmap extension.

Client-Side Hashing & Verification

01. Canonicalization (TypeScript)

.command

```
// Minimal JCS (JSON Canonicalization Scheme) canonicalization
function jcs(obj: any): string {
  if (obj === null) return "null";

  if (typeof obj === "number") {
    if (!Number.isFinite(obj)) throw new Error("non-finite");
    return JSON.stringify(obj);
  }

  if (typeof obj === "string" || typeof obj === "boolean") {
    return JSON.stringify(obj);
  }

  if (Array.isArray(obj)) {
    return "[" + obj.map(jcs).join(",") + "]";
```

```
  }

  // object case
  const keys = Object.keys(obj).sort();
  const parts = keys.map(k => JSON.stringify(k) + ":" + jcs(obj[k]));
  return "{" + parts.join(",") + "}";
}

// Build canonical bytes WITHOUT nonce
function canonicalBytes(payloadNoNonce: any): Uint8Array {
  const s = jcs(payloadNoNonce);
  return new TextEncoder().encode(s);
}

// Keccak256 hashing (uses @noble/hashes)
import { keccak_256 } from "@noble/hashes/sha3";

function hashPayload(payloadNoNonce: any, nonceHex: string): string {
  const a = canonicalBytes(payloadNoNonce);
  const b = Uint8Array.from(
    Buffer.from(nonceHex.replace(/^0x/, ""), "hex")
  );

  const cat = new Uint8Array(a.length + b.length);
  cat.set(a, 0);
  cat.set(b, a.length);

  const h = keccak_256(cat);
  return "0x" + Buffer.from(h).toString("hex");
}
```

Python Equivalent

Python

```python
import json
from eth_hash.auto import keccak

def jcs(obj):
    if obj is None:
        return "null"
    if isinstance(obj, bool):
        return "true" if obj else "false"
    if isinstance(obj, (int, float)):
        return json.dumps(obj, separators=(",", ":"))
    if isinstance(obj, str):
        return json.dumps(obj, ensure_ascii=False, separators=(",", ":"))
    if isinstance(obj, list):
        return "[" + ",".join(jcs(x) for x in obj) + "]"

    # dict
    keys = sorted(obj.keys())
    return "{" + ",".join(json.dumps(k, ensure_ascii=False) + ":" + jcs(obj[k])
for k in keys) + "}"

def hash_payload(payload_no_nonce, nonce_hex):
    canonical = jcs(payload_no_nonce).encode("utf-8")
    nonce = bytes.fromhex(nonce_hex[2:] if nonce_hex.startswith("0x") else non-
ce_hex)
    return "0x" + keccak(canonical + nonce).hex()
```

01. Verification Steps (User)

- **Candles:** 5m, 15m, 1h (OHLCV).
- **Trades:** Tick-by-tick last price/size + aggressor side.
- **Order Book (L2):** Best bid/ask, depth ladders (top-N), diff/partial updates.

Analytics & Performance

01. Trade Construction

- **Entry:** Market or limit assumption within the published entry zone; configurable rule:
  - "First touch" of zone → filled at mid of zone ± slippage, or
  - "VWAP of zone" based on tick data.
- **TP/SL:** Laddered take profits; stop loss hard.
- **Latency:** We log delivery latency per user (ms) but performance is computed from market data timestamps, never user latency.

02. Metrics

- **Hit Rate, Profit Factor, Expectancy**
- **Sharpe (daily):** μRσR252\frac{\mu_R}{\sigma_R} \sqrt{252} where RR are per-trade returns
- **Max Drawdown, Ulcer Index**
- **Realized Slippage:** entry/exit difference vs modeled fill
- **Regime-wise stats:** bull/bear/range segmentation

All metrics derived **only** from **revealed** and **verified** signals.

Product

01. For Traders

- **Live LONG/SHORT calls with entry, TP ladder, SL, confidence & notes**
- **Verification badge:** Committed / Awaiting Reveal / Verified / Expired
- **Backtest & Replay:** Per-day playback with OB overlays
- **Leaderboards:** Per-pair & per-model equity curves (slippage-aware)
- **Risk controls:** Max concurrent signals, loss-streak cool-downs
- **Priority latency** for stakers

02. For Builders & Funds

- **Signals API:** /commits, /reveals, /signals?verified=true
- **Webhooks** for automation
- **Verifier SDK** (TS/Python)
- **Strategy Marketplace:** stake-to-list models; poor performance → partial slashing; best models gain exposure and revenue share (in CTICK)

Token Utility — CTICK

01. Roles

- **Access Tiers:** Free (delayed) → Lite → Pro → Elite (priority latency, custom pairs)
- **Credits:** Backtests, custom alerts, API bursts—cheapest when paid in CTICK
- **Buyback & Burn:** X% of net revenue regularly buys CTICK on DEX and sends to a burn address

Staking:

- User staking for tiers and priority
- **Creator/Model staking for marketplace listing;** performance-based slashing rules
- **Governance:** Advisory votes on model deployments/retirements, fee splits, pairs, grants, and burn cadence

Token Economics

Chain:
Arbitrum

Type:
Fixed-supply ERC-20 with permit

Supply:
100,000,000

Allocation:

- Community & Rewards 40% (48m linear)
- Treasury/Ecosystem 20% (6m cliff + 36m)
- Liquidity & MM 10% (unlocked for liquidity programs)
- Team 15% (12m cliff + 48m)
- Investors 10% (6m cliff + 24m)
- Early User Airdrop 5%

Revenue → Token Sink: Net revenue → buyback contract → swap USDC/ETH→CTICK via DEX router → burn (timelocked policy, chain-recorded).

Gating & Account Linking

- **Wallet gating:** Sign a message to link wallet ↔ platform account ↔ Telegram ID (no custody).
- **Telegram:** Bot is admin in the channel; channel posts or commands trigger responses.
- **Rate limits:** Per-user and per-tier limits for APIs and alert frequency

Security & Threat Model

01. Threats

- **Hindsight editing:** Mitigated by PoP commit-reveal with nonce & public verification.
- **Payload guessing:** Prevented by nonce entropy (32 bytes).
- **Withholding losers:** Deterred by maxReveal windows + public Expired flags counted negatively.
- **Front-running commits:** Harmless; commits reveal no payload.
- **Publisher spoofing:** Off-chain signature binds payloads to publisher key; publisher in meta must match.
- **Chain reorgs:** UI waits k confirmations; explorer updates finality state.
- **Key compromise:**
  - **Contracts:** Gnosis Safe multisig + timelock for upgrades & treasury
  - **Publisher keys:** HSM/YubiHSM; key rotation with on-chain publisher registry

Audits & Bounties

- External smart contract audits before mainnet.
- Public bug bounty for PoP, buyback, and staking contracts

Compliance Notes

- No profit promises. CTICK is a utility (access, staking, governance), not a revenue-sharing instrument.
- KYC/AML for sales where required; geofencing where necessary.
- Disclosures: All signals are educational; trading involves risk including total loss.

Operations & DevOps

01. Observability

- **Logs:** Structured, trace-ID per signal (commit tx hash, explorer URL, telegram message ID).
- **Metrics:** Ingestion lag, model compute time, commit latency, reveal latency, failed reveals, API p95.
- **Alerts:** WS disconnects, data gaps, commit failures, reveal deadline approaching.

02. Datastores (sketch)

- signals_live (Redis): latest signals & delivery queues
- signals_committed (Postgres/ClickHouse): commit meta, tx hashes
- signals_revealed (ClickHouse): full payloads + verification status
- market_data (ClickHouse): ticks, OB snapshots
- users, wallet_links, staking, tiers

Roadmap

Phase 1 — Foundations

- PoP Contract v1 (testnet → mainnet)
- Verifier SDK + CLI (open source)
- Telegram live signals with verification badges
- Public Explorer (commits/reveals)

Phase 2 — Product Scale

- Web dashboard, backtests & replay
- Access gating (wallet + Telegram linking)
- Buyback & burn module linked to revenue
- Enterprise/API plans

Phase 3 — Ecosystem

- Strategy Marketplace (stake-to-list, slashing)
- Creator channels; rev-share via CTICK
- Optional execution adapters for selected DEX/CEX (non-custodial)

Phase 4 — Governance & Expansion

- Token-holder governance for parameters
- Additional chains/venues for data & execution
- Research grants and community challenges

Example End-to-End Walkthrough

01. Live Broadcast (T_0)

- AI policy outputs payload (without nonce).
- Orchestrator draws 32-byte nonce N.
- Computes H = keccak256(JCS(payload_no_nonce) || N).
- Sends commit(H, meta) (or batches into Merkle root).
- Publishes human-readable message to Telegram & web with entry/TP/SL

Explorer shows: Committed (pending k conf)

02. After Window (T_1 within [min,max])

- Orchestrator calls reveal(payload_no_nonce, N, [proof]).
- Contract verifies hash (and proof if batched) + time window; emits Revealed.

Explorer shows: Verified + link to payload bytes/CID

03. User Verification

- Copy revealed payload & nonce → run Verifier CLI → checksum equals on-chain hash → status OK.

ASCII Sequence Diagram

Governance Outline

- **Parameters:** buyback %, reveal windows, batch sizes, staking & slashing rules, pair listings.
- **Process:** Off-chain discussion → on-chain vote (advisory where required) → timelocked execution by multisig.
- **Transparency:** All parameter changes recorded on-chain; dashboards show history

KPIs

- Verified commits/reveals ratio
- Median delivery latency (by tier)
- Churn/Retention (D30/D90)
- Net revenue routed to buybacks (and CTICK burnt)
- Marketplace health: models listed, active stakes, slashing events
- Uptime, failed commits, missed reveals

Risks & Mitigations (Non-Exhaustive)

- Model degradation (regime shifts): Versioned models; continuous evaluation; sunset votes.
- Infra outages: Multi-region WS; backpressure; auto-reconnect; dead-letter queues.
- Regulatory change: Flexible geo-fencing and offerings.
- Smart contract bugs: Audits, timelocks, minimal on-chain complexity for PoP core.
- Exchange data anomalies: Cross-venue sanity checks; gap detection.

Legal & Disclaimers

- CHAIN TICK provides **educational signals.** Trading is risky; users can lose all capital.
- **CTICK is a utility token** for access, staking, and governance. **It does not** represent equity, debt, or revenue rights.
- No guarantees of performance. Past results (even verifiable) don't imply future returns.

Appendices

01. Example Canonicalization & Hash (Worked)

Payload (no nonce):

```
{
  "timestamp" : "2025-08-15T18:22:43Z",
  "pair"      : "BTCUSDT",
  "timeframe" : "5m",
  "direction" : "LONG",
  "entry"     : {
    "from"    : 64250,
    "to"      : 64290
  },
  "take_profit": [64520, 64780, 65150],
  "stop_loss"  : 63940,
  "model"      : "ob-v3.2",
  "confidence" : 0.74,
  "notes"      : "imbalance > 30%, spread tight, buyer aggression rising",
  "channel"    : "telegram://@chaintick",
  "publisher"  : "0xPUB...KEYID"
}
```

Nonce:
0x7fd1c3d4102af9b2e1b7ac9e0b7d2a5f5a3c0f9b8e4d1c0a7f22efc6d1aa01e3
Canonicalize with JCS → bytes C. Compute:
H = keccak256(C ll nonce)
= 0x9f1a…(example)
On reveal, anyone recomputes H and matches the stored commit or verifies Merkle leaf against root.

Merkle Batching

- Leaves: L_i = keccak256(C_i || nonce_i)
- Root: pairwise hash (sorted ordering to avoid second-preimage ambiguity)
- Reveal passes (payload_i, nonce_i, proof_i); contract checks L_i and proof → root.

Off-Chain Publisher Signature (Optional)

- Sign C (payload_no_nonce) with ECDSA secp256k1; include sig and pub.
- SDK verifies the signature matches the publisher in commit meta.

Buyback & Burn Module (Sketch)

- Treasury triggers buybacks via timelocked function; executes swaps on DEX router; sends CTICK to burn address.
- Parameters (percentages, cadence) controlled by governance with delay.

Telegram Integration

- Bot as admin in channel; listens for channel_post updates.
- Commands: /subscribe BTCUSDT 5m, /tier, /verify <id>.
- Bot posts signal with link to Explorer and verification badge.

Conclusion

CHAIN TICK makes a simple promise: “Trade with receipts.”

By committing each signal on-chain at broadcast and revealing the exact payload later, we eliminate hindsight edits and enable transparent, third-party verification. The **CTICK** token powers practical access, marketplace alignment, and usage-driven burns, linking real product demand to token utility.

Next steps:

- Traders: connect Telegram and view your first PoP-verified signals.
- Builders: integrate the Verifier SDK and the Signals API.
- Partners/Investors: review the PoP contract, token utility, and roadmap; let’s discuss
- distribution and enterprise integrations.