CONTINUUM LEDGER

Technical Architecture Paper

Version 1.0 | 2026

SVM Runtime | PoH + PoS Consensus | Native Compliance Architecture

CONFIDENTIAL DRAFT — NOT FOR PUBLIC DISTRIBUTION

NOTICE

This document specifies the technical architecture of the Continuum Ledger protocol. It contains pseudocode, formal specifications, and Rust-based smart contract samples intended for review by engineers, auditors, and technical due diligence teams. All specifications are subject to revision during testnet development.

1. ARCHITECTURE OVERVIEW

1. Architecture Overview

The Continuum Ledger is a purpose-built Layer 1 blockchain with three architectural pillars that are inseparable by design: a Proof of History + Proof of Stake consensus engine, a Sealevel-inspired parallel execution runtime, and a native compliance layer funded and governed at the protocol level.

Unlike blockchains that add compliance as an external layer, the Continuum Ledger treats compliance as a first-class execution concern. Travel Rule checks, sanctions screening, and AI integrity verification run in parallel with transaction processing using the same account-model parallelism that gives the runtime its throughput advantage.

1.1 System Layers

Layer	Component	Technology	Purpose
L0 — Network	P2P mesh	libp2p + custom gossip	Peer discovery, block propagation, miner registry
L1 — Consensus	PoH clock + PoS finalisation	SHA-256 VDF + BFT voting	Ordering, timestamps, economic finality
L2 — Execution	Continuum Runtime (CRT)	SVM-inspired parallel VM	Smart contract execution, account state
L3 — Compliance	Compliance Execution Engine (CEE)	Parallel account spaces	Travel Rule, sanctions, AI scoring
L4 — Storage	Tiered chain store	RocksDB + merkle trees	2yr / 5yr / 10yr retention windows
L5 — Governance	On-chain DAO contracts	Native Rust programs	Parameter changes, upgrades, treasury

DESIGN PRINCI

Every layer is independently upgradeable via governance without requiring a hard fork, provided the interface between layers remains stable. The compliance layer can receive new sanction list sources without touching consensus. The storage layer can adopt new merkle schemes without touching the VM.

P
L
E

2. CONSENSUS SPECIFICATION

2. Consensus Specification

2.1 Proof of History — Formal Definition

Proof of History (PoH) is a Verifiable Delay Function (VDF) producing a sequential, cryptographically verifiable record of elapsed time. Each PoH tick is defined as:

```
PoH(n) = SHA256(PoH(n-1) || data(n))

Where:
  PoH(0)  = genesis_hash (publicly committed at chain launch)
  data(n) = any observed event (transaction hash, empty = clock tick)
  n       = sequence counter, monotonically increasing

Verification of PoH(n) from PoH(0) requires sequential computation.
Forgery of a PoH sequence at position n requires computing all n steps.
This makes backdating or reordering events computationally infeasible.
```

The PoH clock runs continuously on the leader node. Every transaction submitted to the network is hashed into the PoH sequence at the moment of receipt, creating an unforgeable timestamp. This timestamp is what enforces the 2-year, 5-year, and 10-year chain retention windows — pruning is mathematically verifiable, not policy-dependent.

2.2 Proof of Stake — Vote Weight Formula

The Continuum Ledger uses a modified PoS where vote weight is a composite of economic stake and proven operational merit. This is the formal vote weight calculation:

```
vote_weight(miner) = tier_multiplier(tier) * blocks_confirmed(miner, last_90_days)
                   * uptime_score(miner) * compliance_score(miner)

Where:
  tier_multiplier(T1) = 1
  tier_multiplier(T2) = 3
  tier_multiplier(T3) = 6

  blocks_confirmed = count of blocks confirmed in last 6,480,000 slots (90 days)
  uptime_score     = (online_slots / total_slots_last_90d), range [0.0, 1.0]
  compliance_score = AI integrity score, range [0.0, 1.0]

Example — Apex miner, 50,000 blocks, 99.9% uptime, AI score 0.97:
  vote_weight = 6 * 50000 * 0.999 * 0.97 = 290,709
```

2.3 Block Production Lifecycle

Block production follows a slot-based schedule. Each slot is 400ms, targeting sub-second finality. The lifecycle per slot is:

```
SLOT LIFECYCLE (400ms target):

  T+0ms    Leader selected via VRF from active miner set (weighted by vote_weight)
  T+0ms    Leader begins collecting transactions from mempool
  T+50ms   Compliance Execution Engine runs parallel pre-checks:
             - Travel Rule metadata validation
             - Sanctions list screening (OFAC, EU, UN, SARB)
             - AI anomaly pre-score
  T+150ms  Leader builds block, stamps transactions into PoH sequence
  T+200ms  Block broadcast to all validators via gossip protocol
  T+250ms  Validators verify PoH sequence integrity
  T+300ms  VRF-selected attestor set (3 cross-tier miners) countersigns
  T+350ms  2/3 supermajority of weighted stake confirms block
  T+400ms  Block finalised — state committed, rewards distributed

If leader fails to produce by T+400ms:
  Next leader in VRF rotation takes over. Slot marked skipped.
  Skipped slots recorded on-chain for uptime scoring.
```

2.4 Leader Selection — VRF Specification

Leader selection uses a Verifiable Random Function to prevent prediction and manipulation. The VRF seed is derived from the previous block hash, ensuring no participant can know the leader schedule more than one epoch ahead.

```
// Rust pseudocode — leader selection
fn select_leader(epoch_seed: Hash, slot: u64, miner_registry: &[Miner]) -> PublicKey
{
    let vrf_input = hash_concat(epoch_seed, slot.to_le_bytes());
    let vrf_output = vrf_prove(vrf_input, epoch_seed);
    let selection_value = vrf_output.to_u64() %
total_weighted_stake(miner_registry);
    weighted_select(miner_registry, selection_value)
}

// Weighted selection — higher vote_weight = proportionally more slots
fn weighted_select(miners: &[Miner], value: u64) -> PublicKey {
    let mut cumulative = 0u64;
    for miner in miners {
        cumulative += miner.vote_weight;
        if value < cumulative { return miner.public_key; }
    }
    miners.last().unwrap().public_key
}
```

3. CONTINUUM RUNTIME (SVM)

3. Continuum Runtime (CRT)

3.1 Account Model

The Continuum Runtime uses an account-based parallel execution model. Every transaction must declare all accounts it will read from or write to before execution begins. The runtime scheduler uses these declarations to identify non-conflicting transaction sets that can execute simultaneously.

```
// Account structure
#[derive(BorshSerialize, BorshDeserialize)]
pub struct Account {
    pub lamports:   u64,         // CONT token balance (1 CONT = 1_000_000_000
lamports)
    pub data:       Vec<u8>,     // Program state data
    pub owner:      Pubkey,      // Program that owns this account
    pub executable: bool,        // True if this account contains a program
    pub rent_epoch: u64,         // Epoch when rent was last collected
    // Continuum extensions:
    pub tier:       MinerTier,   // T1 / T2 / T3 / None
    pub kyc_hash:   Option<Hash>,// Hash of KYC record (off-chain, privacy-
preserving)
    pub travel_rule_eligible: bool, // Pre-cleared for Travel Rule transactions
}

// Transaction must declare accounts upfront
pub struct Transaction {
    pub signatures:   Vec<Signature>,
    pub message:      Message,
}
pub struct Message {
    pub account_keys:       Vec<Pubkey>,  // ALL accounts touched
    pub recent_blockhash:   Hash,
    pub instructions:       Vec<Instruction>,
    // Continuum extension:
    pub compliance_accounts: Vec<Pubkey>, // Travel Rule + sanctions accounts
}
```

3.2 Parallel Execution Scheduler

The scheduler is the core innovation that separates CRT from sequential VMs. It builds a dependency graph from account declarations and executes independent transaction batches in parallel across available CPU cores.

```
fn schedule_parallel(transactions: Vec<Transaction>) -> Vec<ExecutionBatch> {
    let mut batches: Vec<ExecutionBatch> = vec![];
    let mut locked_accounts: HashSet<Pubkey> = HashSet::new();
```

```
    for tx in transactions {
        let write_accounts = tx.write_accounts();
        let read_accounts  = tx.read_accounts();

        // Check for write-write or write-read conflicts
        let conflicts = write_accounts.iter()
            .any(|a| locked_accounts.contains(a));

        if !conflicts {
            // No conflict — add to current batch, lock its write accounts
            batches.last_mut().unwrap().push(tx);
            locked_accounts.extend(write_accounts);
        } else {
            // Conflict — start new batch, reset lock set
            locked_accounts = write_accounts.into_iter().collect();
            batches.push(ExecutionBatch::new(tx));
        }
    }
    batches
}

// Each batch executes on its own thread — true parallelism
fn execute_all(batches: Vec<ExecutionBatch>) -> Vec<ExecutionResult> {
    batches.par_iter().map(|b| b.execute()).collect()
}
```

3.3 Native Program Interface

Smart contracts on the Continuum Ledger are called Programs, written in Rust and compiled to a sandboxed bytecode. Programs interact with account state through a strict interface that prevents unauthorised state mutation.

```
// Minimal program entrypoint — all programs follow this signature
pub fn process_instruction(
    program_id: &Pubkey,
    accounts:   &[AccountInfo],
    data:       &[u8],
) -> ProgramResult {
    // Deserialise instruction data
    let instruction = ProgramInstruction::try_from_slice(data)?;
    match instruction {
        ProgramInstruction::Transfer { amount } => {
            process_transfer(accounts, amount)
        }
        ProgramInstruction::Stake { miner, amount } => {
            process_stake(accounts, miner, amount)
        }
        _ => Err(ProgramError::InvalidInstruction)
    }
}
```

4. COMPLIANCE EXECUTION ENGINE

4. Compliance Execution Engine (CEE)

4.1 Architecture

The Compliance Execution Engine is a parallel execution module that runs alongside transaction processing using separate account spaces. Because compliance accounts are declared independently of transaction accounts, the CEE never blocks transaction throughput. It is not a filter that transactions pass through — it is a parallel observer that flags and records.

```
// CEE runs in parallel with transaction execution
// Compliance accounts are separate from tx accounts — no blocking

pub struct ComplianceContext {
    pub tx_hash:          Hash,
    pub originator:       Pubkey,
    pub beneficiary:      Pubkey,
    pub amount_lamports:  u64,
    pub cross_border:     bool,
    pub miner_tier:       MinerTier,
}

pub enum ComplianceResult {
    Clear,                          // No flags
    TravelRuleRequired(TRPayload), // Attach Travel Rule data
    SanctionsFlag(SanctionRecord), // Immediate block + SAR
    AiAnomaly(AnomalyScore),       // Log for review
}

pub fn run_compliance_checks(ctx: &ComplianceContext) -> ComplianceResult {
    // Step 1: Threshold check
    if ctx.amount_lamports >= travel_rule_threshold(ctx.miner_tier) {
        let tr = collect_travel_rule_data(ctx);
        if !tr.is_complete() { return ComplianceResult::TravelRuleRequired(tr); }
    }
    // Step 2: Sanctions screening
    if sanctions_list::is_flagged(&ctx.originator)
    || sanctions_list::is_flagged(&ctx.beneficiary) {
        return ComplianceResult::SanctionsFlag(build_sar(ctx));
    }
    // Step 3: AI anomaly score
    let score = ai_integrity::score_transaction(ctx);
    if score.value > AI_FLAG_THRESHOLD {
        return ComplianceResult::AiAnomaly(score);
    }
    ComplianceResult::Clear
}
```

4.2 Travel Rule Implementation

Travel Rule data is attached to transactions as a mandatory instruction when the amount exceeds the tier-appropriate threshold. The data is encrypted to the receiving VASP's public key — only the authorised counterparty can read it. The existence of the attachment is public; the contents are private.

```
#[derive(BorshSerialize, BorshDeserialize)]
pub struct TravelRulePayload {
    // Originator fields (FATF Rec 16)
    pub originator_name:     EncryptedField,
    pub originator_wallet:   Pubkey,
    pub originator_address:  Option<EncryptedField>, // Required above $3000
    pub originator_id_hash:  Option<Hash>,           // KYC reference

    // Beneficiary fields
    pub beneficiary_name:    EncryptedField,
    pub beneficiary_wallet:  Pubkey,

    // Transaction metadata
    pub purpose_code:        PurposeCode,  // SA CFM 2026 transaction purpose
    pub cross_border:        bool,
    pub declared_currency:   CurrencyCode,
    pub timestamp_poh:       u64,          // PoH sequence number = unforgeable
timestamp

    // Miner attestation
    pub attesting_miner:     Pubkey,
    pub miner_tier:          MinerTier,
    pub miner_signature:     Signature,
}

// Threshold by tier
fn travel_rule_threshold(tier: MinerTier) -> u64 {
    match tier {
        MinerTier::T1 => 1_000 * LAMPORTS_PER_CONT,  // $1000 FATF standard
        MinerTier::T2 => 0,                           // All regulated transfers
        MinerTier::T3 => 0,                           // Every transaction
    }
}
```

4.3 AI Integrity Layer — Technical Specification

The AI integrity layer consists of a network of federated AI nodes, each running the same versioned model independently. A compliance flag is only raised when a supermajority of nodes agree — preventing any single compromised node from censoring or manipulating transaction processing.

```
// Federated AI consensus — flag only on supermajority agreement
pub fn federated_ai_consensus(
    scores: Vec<AiNodeScore>,   // One score per federated AI node
    threshold: f64,             // Flag threshold (default 0.85)
    quorum: f64,                // Supermajority required (default 0.67)
```

```
) -> AiFinalVerdict {
    let flagging_nodes = scores.iter()
        .filter(|s| s.anomaly_score > threshold)
        .count();
    let flag_ratio = flagging_nodes as f64 / scores.len() as f64;

    if flag_ratio >= quorum {
        AiFinalVerdict::Flag {
            score: scores.iter().map(|s| s.anomaly_score).sum::<f64>() /
scores.len() as f64,
            reason: aggregate_reasons(&scores),
            node_count: scores.len(),
            flagging_count: flagging_nodes,
        }
    } else {
        AiFinalVerdict::Clear
    }
}

// AI node scores four dimensions per transaction
pub struct AiNodeScore {
    pub anomaly_score:     f64,  // [0.0, 1.0] overall
    pub volume_spike:      f64,  // Unusual volume for this wallet
    pub pattern_match:     f64,  // Matches known exploit patterns
    pub graph_anomaly:     f64,  // Unusual transaction graph structure
    pub sanctions_proximity: f64,// Degrees of separation from flagged wallets
}
```

MODEL GOVERNANCE

AI model weights are versioned on-chain as a hash commitment. Any upgrade requires a 67% supermajority DAO vote with a 7-day timelock. The model itself is stored on IPFS and pinned by all Apex miners. No single party controls what the AI watches for.

5. CORE SMART CONTRACTS

5. Core Smart Contracts

5.1 Miner Registry Program

The Miner Registry is the authoritative on-chain record of all active miners, their tiers, their bonded successors, and their performance metrics. All other programs read from this registry — it is the source of truth for the entire system.

```
#[account]
pub struct MinerRecord {
    pub public_key:          Pubkey,
    pub tier:                MinerTier,
    pub registered_at_slot:  u64,
    pub bond_lamports:       u64,
    pub successor:           Pubkey,       // Bonded successor — mandatory
    pub successor_bond:      u64,          // Successor's posted bond
    pub blocks_confirmed:    u64,          // Lifetime total
    pub blocks_90d:          u64,          // Rolling 90-day window
    pub uptime_score:        f64,          // Rolling 90-day uptime
    pub ai_score:            f64,          // Latest AI integrity score
    pub compliance_status:   ComplianceStatus,
    pub last_snapshot_slot:  u64,          // Last merkle root published
    pub chain_retention_yrs: u8,           // 2, 5, or 10
    pub total_staked:        u64,          // CONT staked against this miner
    pub staker_count:        u32,
    pub slash_count:         u8,
    pub is_active:           bool,
}

// Register new miner — called at genesis or when new miner joins
pub fn register_miner(
    ctx: Context<RegisterMiner>,
    tier: MinerTier,
    successor: Pubkey,
    bond_amount: u64,
) -> ProgramResult {
    require!(bond_amount >= min_bond(tier), ErrorCode::InsufficientBond);
    require!(ctx.accounts.successor_account.is_initialized(),
ErrorCode::InvalidSuccessor);
    // Transfer bond to escrow
    transfer_lamports(&ctx.accounts.miner, &ctx.accounts.bond_escrow, bond_amount)?;
    // Initialise registry record
    let record = &mut ctx.accounts.miner_record;
    record.public_key = ctx.accounts.miner.key();
    record.tier = tier;
    record.successor = successor;
    record.bond_lamports = bond_amount;
    record.is_active = true;
    Ok(())
}
```

5.2 Staking Program

The Staking Program handles direct miner-to-staker bonds. Stakers lock CONT against a specific miner and earn proportional rewards from that miner's block confirmations. Slashing of the miner does not affect staked principal.

```
#[account]
pub struct StakePosition {
    pub staker:          Pubkey,
    pub miner:           Pubkey,
    pub amount_lamports: u64,
    pub staked_at_slot:  u64,
    pub unlock_slot:     u64,   // staked_at + lock_period(miner.tier)
    pub rewards_earned:  u64,   // Accumulated unpaid rewards
    pub last_claim_slot: u64,
}

// Lock periods by tier (in slots, 1 slot = 400ms)
fn lock_period_slots(tier: MinerTier) -> u64 {
    match tier {
        MinerTier::T1 => 1_512_000,   // 7 days
        MinerTier::T2 => 3_024_000,   // 14 days
        MinerTier::T3 => 6_480_000,   // 30 days
    }
}

// Reward calculation per block — staker's share
pub fn calculate_staker_reward(
    block_reward_lamports: u64,
    staker_position: &StakePosition,
    miner_total_staked: u64,
) -> u64 {
    // 20% of block reward goes to stakers, split pro-rata by stake size
    let staker_pool = block_reward_lamports * 20 / 100;
    staker_pool * staker_position.amount_lamports / miner_total_staked
}

// Unstake — enforces lock period, releases principal + rewards
pub fn unstake(ctx: Context<Unstake>) -> ProgramResult {
    let pos = &ctx.accounts.stake_position;
    let clock = Clock::get()?;
    require!(clock.slot >= pos.unlock_slot, ErrorCode::StillLocked);
    // Transfer principal back to staker
    transfer_lamports(&ctx.accounts.stake_vault, &ctx.accounts.staker,
pos.amount_lamports)?;
    // Transfer accumulated rewards
    transfer_lamports(&ctx.accounts.reward_vault, &ctx.accounts.staker,
pos.rewards_earned)?;
    // Close position account
    pos.close(ctx.accounts.staker.to_account_info())
}
```

5.3 Slashing Program

The Slashing Program is called by the AI integrity layer or by cross-attestation failures. It executes graduated penalties against the miner's bond and routes all slashed tokens directly to

the burn pool.

```
pub enum SlashReason {
    DoubleSigning,       // 10% of bond
    DataTampering,       // 30% of bond
    ComplianceBreach,    // 50% of bond + tier demotion
    ExtendedOffline,     //  5% of bond per event
    AiScoreManipulation, // 100% of bond + permanent blacklist
}

pub fn execute_slash(
    ctx: Context<ExecuteSlash>,
    reason: SlashReason,
) -> ProgramResult {
    let miner = &mut ctx.accounts.miner_record;
    let slash_pct = slash_percentage(&reason);
    let slash_amount = miner.bond_lamports * slash_pct / 100;

    // Deduct from bond
    miner.bond_lamports -= slash_amount;
    miner.slash_count += 1;

    // Route ALL slashed tokens to burn pool — not to treasury
    transfer_lamports(
        &ctx.accounts.bond_escrow,
        &ctx.accounts.burn_pool,
        slash_amount
    )?;

    // Handle tier demotion for compliance breach
    if matches!(reason, SlashReason::ComplianceBreach) {
        miner.tier = demote_tier(miner.tier);
        emit!(TierDemotionEvent { miner: miner.public_key, new_tier: miner.tier });
    }

    // Permanent blacklist for AI manipulation
    if matches!(reason, SlashReason::AiScoreManipulation) {
        miner.is_active = false;
        miner.blacklisted = true;
    }

    // Trigger successor activation if bond falls below minimum
    if miner.bond_lamports < min_bond(miner.tier) {
        activate_successor(ctx)?;
    }
    Ok(())
}
```

6. CHAIN STORAGE ARCHITECTURE

6. Chain Storage Architecture

6.1 Tiered Retention Model

Chain storage is managed per-miner according to their tier. The PoH timestamp on every block makes retention window enforcement cryptographically verifiable — a miner cannot claim to hold data they have pruned, and cannot claim to have pruned data they are required to retain.

Tier	Live chain	Pruning trigger	Snapshot requirement	Storage estimate
T1 — Small	2 years	Blocks older than 2yr PoH timestamp	Annual merkle root to block 1	~500 GB
T2 — Mid	5 years	Blocks older than 5yr PoH timestamp	Annual + yr-5 state image minted	~1.2 TB
T3 — Apex	10 years	Blocks older than 10yr PoH timestamp	Annual + yr-5 + yr-10 epoch mint	~2.5 TB

6.2 Merkle Snapshot Specification

Every miner publishes an annual snapshot as a merkle root of their held chain segment. This root is written to block 1 of the new year as a program instruction, making it part of the permanent chain record. The snapshot enables forensic verification of pruned data without requiring full chain storage.

```
// Annual snapshot structure
#[derive(BorshSerialize, BorshDeserialize)]
pub struct AnnualSnapshot {
    pub year:            u32,
    pub miner:           Pubkey,
    pub miner_tier:      MinerTier,
    pub slot_range:      (u64, u64),      // First and last slot of the year
    pub merkle_root:     Hash,             // Root of all block hashes in range
    pub state_root:      Hash,             // Account state root at year end
    pub tx_count:        u64,              // Total transactions in year
    pub ai_model_version: Hash,            // AI model hash at year end
    pub miner_signature: Signature,
    pub attestor_sigs:   [Signature; 3],  // Cross-tier attestation signatures
}

// Snapshot verification — any party can verify pruned data existed
pub fn verify_transaction_in_snapshot(
    tx_hash: Hash,
    merkle_proof: Vec<Hash>,
    snapshot: &AnnualSnapshot,
) -> bool {
    // Walk the merkle proof from tx_hash to snapshot.merkle_root
    let computed_root = merkle_proof.iter().fold(tx_hash, |acc, sibling| {
```

```
        hash_pair(acc, *sibling)
    });
    computed_root == snapshot.merkle_root
}
```

7. SECURITY MODEL

7. Security Model

7.1 Cryptographic Assumptions

Primitive	Used for	Security assumption	Quantum risk
SHA-256	PoH VDF, merkle trees, block hashes	Collision resistance, preimage resistance	Medium — Grover's algo halves effective bits to 128
Ed25519	Transaction signatures, miner identity	Discrete log hardness on Curve25519	High — Shor's algo breaks ECDLP
VRF (ECVRF)	Leader selection, attestor rotation	Pseudorandomness under DDH assumption	High — same as Ed25519
BFT voting	Block finalisation	2/3 honest validator assumption	None — consensus logic, not cryptographic

QUANTUM ROADMAP

The Continuum Ledger acknowledges quantum risk to Ed25519 and ECVRF. The protocol roadmap includes a post-quantum migration to CRYSTALS-Dilithium (NIST PQC standard) for signatures and CRYSTALS-Kyber for key encapsulation, targeted for the Year 5 halving upgrade. The migration path is designed as a governance-approved hard fork with a 24-month transition window for all wallet holders.

7.2 Attack Surface Analysis

Attack vector	Threat level	Mitigation
33% Byzantine validator attack	Critical	BFT requires 2/3 honest stake. Tier weighting concentrates honest stake at Apex. Economic cost to acquire 33% is prohibitive.
Long-range attack (PoS grinding)	High	PoH makes rewriting history computationally infeasible — attacker must recompute all PoH ticks from the fork point.
Leader DoS during slot	Medium	VRF rotation selects next leader automatically. Skipped slot recorded on-chain affects uptime score.
Cartel of Apex miners	Medium	Mandatory 2yr rotation + VRF attestor selection + staker veto prevents entrenchment.
AI node capture	Medium	Federated consensus requires 67% of AI nodes to agree on a flag. Single compromised node has no effect.
Supply chain attack on miner software	High	Program deployment requires governance approval. Runtime sandbox prevents unauthorised system calls.
Smart contract exploit	High	Formal verification pre-launch. Bug bounty programme. Phased TVL rollout. Governance emergency pause capability.

— End of Technical Architecture Paper —