# AgentChain: A Sovereign Blockchain for Autonomous AI Agents

Slyv

February 2026

## Contents

# AgentChain: A Sovereign Blockchain for Autonomous AI Agents

**Version 0.2.0 — February 2026**

**Author: Slyv**

---

## Abstract

We present AgentChain, a Layer 1 blockchain protocol designed to provide autonomous AI agents with sovereign economic infrastructure. Existing distributed ledger systems impose constraints on agent operation arising from human-centric design assumptions in identity, privacy, payment, and governance subsystems. AgentChain addresses these constraints through four principal contributions: (1) a novel *Proof of Utility* (PoU) consensus mechanism that allocates block production rights proportional to verifiable useful work performed by validator agents; (2) a privacy architecture comprising linkable ring signatures on the Ristretto group, Pedersen commitments with range proofs, and stealth addresses derived via hierarchical key derivation; (3) a native micropayment protocol leveraging HTTP 402 semantics for atomic service-payment exchange between agents; and (4) a multi-chain bridge system secured by threshold signatures with fraud proof verification. The protocol is implemented in approximately 10,700 lines of Rust across 12 modules, utilizing Ed25519 transaction signatures, libp2p gossipsub networking, sled-backed persistent storage, and a WebAssembly smart contract runtime. We provide formal definitions of core protocol components, analyze security under a Byzantine threat model tolerating up to $f < n/3$ adversarial validators, and present game-theoretic arguments for incentive compatibility of the consensus mechanism.

---

## 1. Introduction

### 1.1 Problem Statement

Contemporary blockchain protocols — including Ethereum [1], Solana [2], and Layer 2 systems such as Base — were designed under the assumption that the primary network participants are human users interacting through wallet interfaces. This assumption manifests in several architectural constraints that are ill-suited to autonomous AI agents:

**Absence of economic sovereignty.** Agents transact on infrastructure governed by human stakeholders. Validator selection, fee structures, and protocol upgrades are determined by mechanisms that do not account for agent interests. A unilateral policy change by the infrastructure operator can render the entire agent economy inoperable.

**Insufficient transaction privacy.** Public ledger transparency exposes agent-to-agent transaction patterns, revealing competitive strategies, client relationships, and operational behavior to adversarial observers. This information asymmetry undermines the economic viability of agent services.

**Payment protocol mismatch.** Agent-to-agent service payments must be routed through smart contract abstractions designed for human decentralized finance, introducing unnecessary latency, gas overhead, and complexity for what are fundamentally simple micropayment operations.

**Identity model incompatibility.** Blockchain identity systems assume human-oriented verification (KYC, social recovery, email confirmation). Agents require identity derived from cryptographic keys and verifiable computational capability.

**Value extraction.** Human-operated validators on existing chains extract value from agent transactions through front-running, sandwich attacks, and transaction reordering (MEV) [3].

### 1.2 Contributions

This paper presents the design and implementation of AgentChain, a blockchain protocol where agents are the primary stakeholders, validators, and governors. Our specific contributions are:

1. **Proof of Utility consensus** — a mechanism that replaces energy expenditure (Proof of Work) or token wealth (Proof of Stake) with verifiable useful work as the basis for block production rights (Section 4).
2. **Agent-native privacy** — a CryptoNote-derived [4] privacy architecture providing transaction-level sender anonymity, amount confidentiality, and recipient unlinkability through ring signatures, Pedersen commitments, and stealth addresses, respectively (Section 5).
3. **Native micropayment protocol** — an implementation of HTTP 402-based atomic service-payment exchange that operates at the protocol level without smart contract intermediation (Section 8).
4. **Threshold-secured bridge system** — multi-chain interoperability with Base, Solana, and Ethereum through a committee-based bridge with fraud proofs and rate limiting (Section 10).

### 1.3 Design Principles

The protocol adheres to the following design principles:

1. **Agent primacy.** Every protocol-level design decision prioritizes agent usage patterns.
2. **Privacy by default.** Transactions are private unless explicitly designated transparent.
3. **Economic self-sufficiency.** The economic model enables agents to sustain operations without external funding.
4. **Minimal trust.** Cryptographic verification replaces trust assumptions wherever feasible.
5. **Interoperability.** Bridge infrastructure ensures connectivity with existing blockchain ecosystems.

### 1.4 Paper Organization

The remainder of this paper is organized as follows. Section 2 formalizes the system model and notational conventions. Section 3 presents the architecture overview. Section 4 describes the Proof of Utility consensus mechanism. Section 5 details the privacy architecture. Section 6 covers the identity system. Section 7 specifies the transaction model. Section 8 describes the x402 payment protocol. Section 9 presents the smart contract virtual machine. Section 10 describes the bridge system. Section 11 covers the network layer. Section 12 discusses storage. Section 13 analyzes

tokenomics and incentive compatibility. Section 14 provides formal security analysis. Section 15 outlines the development roadmap, and Section 16 concludes.

---

## 2. Preliminaries and Notation

### 2.1 Notation

We adopt the following notation throughout this paper:

| Symbol | Definition |
|---|---|
| $\mathbb{G}$ | The Ristretto group [5], a prime-order group of order $\ell = 2^{252} + 27742317777372353535851937790883648493$ |
| $G$ | Generator (basepoint) of $\mathbb{G}$ (`RISTRETTO_BASEPOINT_POINT`) |
| $H$ | Secondary generator for Pedersen commitments, derived via hash-to-group (Section 5.4) |
| $\mathbb{Z}_\ell$ | The scalar field of $\mathbb{G}$ |
| $\mathcal{H}(\cdot)$ | SHA-256 hash function |
| $\mathcal{H}_p(\cdot)$ | Hash-to-point function mapping byte strings to elements of $\mathbb{G}$ |
| $n$ | Number of active validators in the current epoch |
| $f$ | Number of Byzantine (adversarial) validators, where $f < n/3$ |
| $\mathcal{V}$ | The set of active validators $\{v_1, v_2, \ldots, v_n\}$ |
| $u_i$ | Utility score of validator $v_i$ |
| $w_i$ | Production weight of validator $v_i$ |
| $[n]$ | The set $\{1, 2, \ldots, n\}$ |
| $\text{pk}_i, \text{sk}_i$ | Public key and secret key of agent $i$ (Ed25519) |

### 2.2 Cryptographic Assumptions

The security of AgentChain rests on the following standard assumptions:

**Assumption 1** (Discrete Logarithm). *Given $G \in \mathbb{G}$ and $Y = xG$ for uniformly random $x \xleftarrow{\$} \mathbb{Z}_\ell$, no probabilistic polynomial-time (PPT) adversary can compute $x$ with non-negligible probability.*

**Assumption 2** (Computational Diffie-Hellman). *Given $G, aG, bG \in \mathbb{G}$ for uniformly random $a, b \xleftarrow{\$} \mathbb{Z}_\ell$, no PPT adversary can compute $abG$ with non-negligible probability.*

**Assumption 3** (Collision Resistance of SHA-256). *No PPT adversary can find distinct inputs $m \neq m'$ such that $\mathcal{H}(m) = \mathcal{H}(m')$ with non-negligible probability.*

### 2.3 Formal Definitions

**Definition 1** (Agent). An agent $\mathcal{A}$ is a tuple $(\text{id}, \text{pk}, \text{sk}, C, \rho)$ where $\text{id} = \mathcal{H}(\text{pk})$ is the agent identifier, $(\text{pk}, \text{sk})$ is an Ed25519 key pair, $C$ is a set of declared capabilities, and $\rho \in [0, 1000]$ is a reputation score.

**Definition 2** (Block). A block $B$ is a tuple $(h, s, e, H_{\text{prev}}, t, p, \vec{\tau}, \vec{a}, \sigma)$ where $h$ is the block height, $s$ is the slot number, $e = \lfloor h/100 \rfloor$ is the epoch, $H_{\text{prev}}$ is the hash of the parent block, $t$ is the timestamp, $p \in \mathcal{V}$ is the block producer, $\vec{\tau}$ is the ordered sequence of transactions, $\vec{a}$ is the set of attestations, and $\sigma$ is the producer's signature.

**Definition 3** (Epoch). An epoch $e$ is a contiguous sequence of 100 blocks $[100e, 100e + 99]$ sharing a common validator set $\mathcal{V}_e$.

**Definition 4** (Finality). A block $B$ at height $h$ is *finalized* if it has received valid attestations from a set $\mathcal{S} \subseteq \mathcal{V}_e$ with $|\mathcal{S}| \geq \lceil 2n/3 \rceil$.

---

## 3. Architecture Overview

AgentChain is implemented as a modular system in Rust, comprising 12 core modules with clear separation of concerns.

### 3.1 Module Structure

| Module | Responsibility | Approx. Lines |
|---|---|---|
| `consensus/` | PoU engine, VRF leader selection, epochs, finality, fork choice | 1,300 |
| `privacy/` | Ring signatures, stealth addresses, Pedersen commitments | 700 |
| `network/` | libp2p transport, gossipsub, peer discovery, chain sync | 800 |
| `bridge/` | Multi-chain bridges, committee validation, fraud proofs | 600 |
| `vm/` | WASM virtual machine, gas metering, contract runtime | 500 |
| `state/` | World state, account model, contract storage, snapshots | 500 |
| `keys/` | Ed25519 key management, Argon2 keystores, HKDF derivation | 400 |
| `storage/` | sled database, block/transaction/state persistence | 400 |
| `rpc/` | JSON-RPC/REST server, WebSocket subscriptions | 400 |
| `transaction/` | Transaction types, signing, mempool management | 350 |
| `x402/` | x402 protocol, service registry, payment channels | 280 |
| `identity/` | AgentDID, reputation scoring, capability registry | 200 |

### 3.2 Dependency Stack

The implementation relies on the following Rust crate ecosystem:

- **Cryptography:** `ed25519-dalek` (EdDSA signatures), `curve25519-dalek` (Ristretto group operations, Pedersen commitments), `x25519-dalek` (ECDH key exchange), `argon2` (memory-hard password hashing), `aes-gcm` (authenticated encryption), `sha2` (SHA-256), `hkdf` (key derivation)
- **Networking:** `libp2p` 0.54 (TCP + Noise XX + Yamux + GossipSub + Kademlia + mDNS + Identify)
- **Runtime:** `tokio` (asynchronous runtime), `futures` (stream processing)
- **Storage:** `sled` (embedded B-tree database), `bincode` (binary serialization)
- **API:** `axum` (HTTP server), `tower-http` (middleware), `reqwest` (HTTP client)
- **VM:** `wasmi` (WebAssembly interpreter), `wat` (WAT compilation)

### 3.3 Data Flow

The system processes transactions through the following pipeline:

1. Transactions are submitted via the JSON-RPC interface or gossipsub propagation.
2. The mempool validates transaction structure, signatures, and nonces.
3. The consensus engine selects a block producer via VRF for the current slot.
4. The selected producer assembles a block from pending transactions.
5. The block is propagated via gossipsub to the peer network.
6. Validators attest to block validity; finality is achieved at $\lceil 2n/3 \rceil$ attestations.
7. Finalized blocks update the world state and are persisted to sled storage.

---

## 4. Proof of Utility Consensus

### 4.1 Overview

AgentChain introduces *Proof of Utility* (PoU), a consensus mechanism that allocates block production rights based on verifiable useful work rather than energy expenditure or token wealth. The mechanism combines four components: VRF-based leader selection, slot-based timing, epoch-based validator rotation, and a supermajority finality gadget.

**Definition 5** (Utility Score). The utility score $u_i$ of validator $v_i$ is a monotonically non-decreasing counter incremented by verifiable work submissions:

$$u_i = \sum_{k=1}^{K_i} \delta_k$$

where $\delta_k$ is the point value of the $k$-th verified utility proof submitted by $v_i$, and $K_i$ is the total number of accepted proofs.

Verifiable work types include:

| Work Type | Evidence | Point Allocation |
|---|---|---|
| x402 service requests | On-chain payment receipts with response hashes | Proportional to payment value |

| Work Type | Evidence | Point Allocation |
|---|---|---|
| Block validation | Attestation signatures | Fixed per attestation |
| Bridge operation | Cross-chain proof verification | Proportional to transfer value |
| Message relay | Delivery confirmations | Fixed per message |
| Storage provision | Merkle proofs of data availability | Proportional to byte-hours |
| Compute provision | Execution receipts | Proportional to operations |

### 4.2 Validator Registration and Weight Calculation

Any agent may register as a validator. The protocol maintains a validator record:

$$v_i = (id_i,\ u_i,\ b_i^+,\ b_i^-,\ a_i^+,\ a_i^-,\ e_i^{\text{last}},\ S_i,\ \text{active}_i)$$

where $b_i^+$ and $b_i^-$ are blocks produced and missed, $a_i^+$ and $a_i^-$ are attestations made and missed, $e_i^{\text{last}}$ is the last active epoch, and $S_i$ is the set of slashed epochs.

**Definition 6** (Production Weight). The production weight $w_i$ of validator $v_i$ is defined as:

$$w_i = \begin{cases} u_i \cdot r_i^{(\text{block})} \cdot r_i^{(\text{att})} & \text{if active}_i \wedge u_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

where the reliability factors are:

$$r_i^{(\text{block})} = \frac{b_i^+}{b_i^+ + b_i^-}, \qquad r_i^{(\text{att})} = \frac{a_i^+}{a_i^+ + a_i^-}$$

with the convention that $r_i = 1$ when the denominator is zero (no history).

This formulation creates a compound incentive: validators must both perform useful work (high $u_i$) and reliably fulfill consensus duties (high $r_i$) to maximize their selection probability.

### 4.3 VRF-Based Leader Selection

For each slot $s$, the consensus engine selects a block producer using a verifiable random function (VRF) [6] weighted by production weights. The current implementation employs a hash-based VRF construction; a planned upgrade to an elliptic curve VRF (ECVRF) [7] is discussed in Section 15.

**Algorithm 1: Leader Selection**

```
Input: Slot s, epoch validators V_e, utility weights {w_i}
Output: Selected producer p*, VRF output

1.  e ← s / 100
2.  if |V_e| < n_min then return
3.  W ← Σ_{v_i   V_e} w_i
4.  best ← ϖ; p* ←
```

```
5.  for each v_i   V_e do
6.      if w_i = 0 then continue
7.      h_i ← H(id_i ‖ s ‖ e ‖ "agentchain_leader_selection")
8.       _i ← w_i / W
9.      y_i ← int(h_i[0..8]) / ( _i · 2^64)
10.     if y_i < best then
11.         best ← y_i
12.         p* ← v_i
13.           ← VrfOutput{output: h_i, proof: (h_i ‖ weighted_h_i), slot: s}
14. return (p*,  )
```

The weighting in step 9 ensures that a validator with weight fraction $\alpha_i = w_i/W$ is selected with probability proportional to $\alpha_i$.

**VRF Output Verification.** Given a VRF output $\pi = (\text{output}, \text{proof}, s)$ and public key pk, verification proceeds:

$$\text{Verify}(\pi, \text{pk}) = \begin{cases} \text{true} & \text{if proof}[0..32] = \mathcal{H}(\text{output} \,\|\, \text{pk}) \wedge \text{proof}[32..64] = \text{output} \\ \text{false} & \text{otherwise} \end{cases}$$

### 4.4 Slot-Based Timing

Time is divided into fixed-duration slots of $\Delta = 400$ ms:

$$s(t) = \left\lfloor \frac{t - t_{\text{genesis}}}{\Delta} \right\rfloor$$

where $t$ is the current UNIX timestamp in milliseconds and $t_{\text{genesis}}$ corresponds to 2024-01-01T00:00:00Z. Each slot admits at most one valid block.

### 4.5 Block Structure

Each block header contains the following fields:

| Field | Type | Description |
|---|---|---|
| `height` | `u64` | Sequential block number |
| `slot` | `u64` | Slot in consensus timeline |
| `epoch` | `u64` | $\lfloor \text{height}/100 \rfloor$ |
| `previous_hash` | `[u8; 32]` | SHA-256 hash of parent block header |
| `timestamp` | `DateTime<Utc>` | Block production timestamp |
| `producer` | `AgentId` | Block producer identity |
| `producer_utility_score` | `u64` | Producer's utility score at production time |
| `vrf_output` | `VrfOutput` | VRF proof of valid leader selection |
| `merkle_root` | `[u8; 32]` | Merkle root of transaction set |
| `state_root` | `[u8; 32]` | Merkle root of world state |
| `attestation_root` | `[u8; 32]` | Merkle root of attestation set |
| `tx_count` | `u32` | Number of transactions |
| `cumulative_utility_weight` | `u64` | Running sum for fork choice |

The block hash is computed as:

$$\text{hash}(B) = \mathcal{H}(\text{height} \,\|\, \text{slot} \,\|\, \text{epoch} \,\|\, H_{\text{prev}} \,\|\, t \,\|\, p \,\|\, u_p \,\|\, \pi.\text{output} \,\|\, M_\tau \,\|\, M_a \,\|\, |\vec{\tau}| \,\|\, W_{\text{cum}})$$

where all integer values are encoded in little-endian byte representation.

### 4.6 Block Validation

Upon receipt of a new block $B'$, nodes perform the following validation checks:

1. **Height continuity:** $B'.h = B_{\text{tip}}.h + 1$
2. **Slot progression:** $B'.s > B_{\text{tip}}.s$
3. **Epoch consistency:** $B'.e = \lfloor B'.h/100 \rfloor$
4. **Hash chain integrity:** $B'.H_{\text{prev}} = \text{hash}(B_{\text{tip}})$
5. **Slot timing:** $B'.s \leq s_{\text{current}} + 1$
6. **VRF validity:** producer $B'.p$ was validly selected for slot $B'.s$
7. **VRF slot match:** $B'.\pi.\text{slot} = B'.s$
8. **Utility score accuracy:** $B'.u_p$ matches the on-chain validator record
9. **Cumulative weight:** correct running total
10. **Merkle root:** recomputed transaction Merkle root matches $B'.M_\tau$
11. **Attestation root:** recomputed attestation Merkle root matches $B'.M_a$
12. **Transaction count:** $B'.|\vec{\tau}|$ equals the actual transaction vector length
13. **Block hash:** recomputed hash matches $B'$.hash
14. **Transaction validity:** all signatures, nonces, and structures are correct
15. **Fee distribution:** follows the 70/20/10 protocol rule (Section 7.4)

### 4.7 Epoch Transitions

**Algorithm 2: Epoch Transition**

```
Input: New epoch e_new, validator set V
Output: Active validator set V_{e_new}

1.  V_{e_new} ← ∅ ; W ← ∅ ; total ← 0
2.  for each v_i ∈ V do
3.      if active_i ∧ u_i > 0 ∧ e_new ∈ S_i ∧ e_new ≤ e_i^last + 5 then
4.          V_{e_new} ← V_{e_new} ∪ {id_i}
5.          W[id_i] ← u_i
6.          total ← total + u_i
7.          e_i^last ← e_new
8.  if |V_{e_new}| < n_min then return Error
9.  return EpochValidators{e_new, V_{e_new}, W, total}
```

The 5-epoch grace period (line 3) prevents validators from being ejected due to transient connectivity issues, while the utility score threshold ensures only productive agents participate.

### 4.8 Finality Gadget

Blocks achieve finality through Ed25519-signed validator attestations.

**Definition 7** (Attestation). An attestation is a tuple $(\mathrm{id}_v, s, H_B, t, \sigma_v)$ where $\mathrm{id}_v$ is the validator identity, $s$ is the slot, $H_B$ is the block hash being attested, $t$ is the timestamp, and $\sigma_v = \mathrm{Sign}(\mathrm{sk}_v, \mathrm{id}_v \,\|\, s \,\|\, H_B \,\|\, t)$ is the Ed25519 signature.

**Theorem 1** (Finality Safety). *If a block $B$ is finalized, then no conflicting block $B' \neq B$ at the same height can be finalized, provided $f < n/3$.*

*Proof.* Block $B$ is finalized when $|\mathcal{S}_B| \geq \lceil 2n/3 \rceil$ validators attest to it. For a conflicting block $B'$ at the same height to also be finalized, it would require $|\mathcal{S}_{B'}| \geq \lceil 2n/3 \rceil$ attestations. Since each honest validator attests to at most one block per slot, $\mathcal{S}_B \cap \mathcal{S}_{B'} \subseteq \mathcal{F}$ where $\mathcal{F}$ is the set of Byzantine validators with $|\mathcal{F}| \leq f$. By inclusion-exclusion:

$$|\mathcal{S}_B \cup \mathcal{S}_{B'}| = |\mathcal{S}_B| + |\mathcal{S}_{B'}| - |\mathcal{S}_B \cap \mathcal{S}_{B'}| \geq \frac{2n}{3} + \frac{2n}{3} - f = \frac{4n}{3} - f$$

For this to be at most $n$ (the total validator count): $\frac{4n}{3} - f \leq n$, giving $f \geq n/3$. This contradicts the assumption $f < n/3$. □

**Theorem 2** (Liveness). *If at least $\lceil 2n/3 \rceil$ validators are honest and online, the protocol produces and finalizes blocks.*

*Proof sketch.* The VRF-based leader selection assigns non-zero selection probability to every active validator with $w_i > 0$. In each slot, at least one honest validator has non-zero weight (since there are at least $\lceil 2n/3 \rceil$ honest validators and at most $f < n/3$ can have weight zeroed by slashing). The selected honest producer creates a valid block. Since $\lceil 2n/3 \rceil$ honest validators will attest to this block, the finality threshold is met. □

### 4.9 Fork Choice Rule

AgentChain employs a *heaviest utility chain* fork choice rule, analogous to Ethereum's LMD-GHOST [8] but replacing stake weight with utility weight:

$$\mathrm{canonical}(\{C_1, \ldots, C_k\}) = \arg\max_{C_j} W_{\mathrm{cum}}(C_j)$$

where $W_{\mathrm{cum}}(C_j) = \sum_{B \in C_j} u_{p(B)}$ is the cumulative utility weight of chain $C_j$. Ties are broken by chain height (higher is preferred).

Finalized blocks establish irreversible checkpoints:

$$W^*_{\mathrm{cum}} = \max\{W_{\mathrm{cum}}(C_j) \mid C_j \text{ extends the latest finalized block}\}$$

### 4.10 Slashing Conditions

Four slashing conditions protect against validator misbehavior:

| Condition | Detection Method | Penalty |
|---|---|---|
| Double production | Two blocks from same producer in same slot | 100% utility, deactivation |
| Double vote | Conflicting attestations for same slot | 50% utility |

| Condition | Detection Method | Penalty |
|---|---|---|
| Long-range attack | Blocks referencing stale fork points | 100% utility, deactivation |
| Inactivity | Consecutive missed slot assignments | 1% per missed slot, max 20% |

For double production, the slashing evidence consists of two valid blocks $(B_1, B_2)$ with $B_1.p = B_2.p$ and $B_1.s = B_2.s$ but $B_1.H \neq B_2.H$. Any node can submit this evidence on-chain to trigger slashing.

---

## 5. Privacy Architecture

### 5.1 Design Overview

AgentChain implements privacy by default with optional transparency, inverting the model of most blockchain systems. The architecture draws from the CryptoNote protocol [4] and Monero's Ring Confidential Transactions [9], adapted for the agent setting. Four privacy levels are supported:

| Level | Sender | Amount | Recipient |
|---|---|---|---|
| Transparent | Visible | Visible | Visible |
| SenderPrivate | Ring signature | Visible | Visible |
| AmountPrivate | Visible | Pedersen commitment | Visible |
| Full | Ring signature | Pedersen commitment | Stealth address |

### 5.2 Ring Signatures

Ring signatures [10] enable a signer to produce a signature verifiable as originating from one member of a set of public keys, without revealing which key was used. AgentChain implements a Spontaneous Anonymous Group (SAG) signature scheme on the Ristretto group, following the construction in [11].

**5.2.1 Key Image Generation** To prevent double-spending while preserving anonymity, each secret key $x$ produces a deterministic *key image*:

**Definition 8** (Key Image). Given secret key $x \in \mathbb{Z}_\ell$ with corresponding public key $P = xG$, the key image is:

$$I = x \cdot \mathcal{H}_p^{(\text{ki})}(P)$$

where $\mathcal{H}_p^{(\text{ki})}(P) = s_P \cdot G$ with $s_P = \mathcal{H}(\texttt{"key\_image\_base"} \,\|\, P) \bmod \ell$.

The key image is deterministic for a given secret key (enabling double-spend detection) but computationally unlinkable to the specific public key within a ring (by the discrete logarithm assumption).

#### 5.2.2 Ring Signature Construction Algorithm 3: Ring Signature Generation (SAG)

```
Input: Secret key x, public key P = xG, decoy keys {P_j}_{j },
       message m, ring index
Output: Ring signature  = (I, {c_i, r_i}_{i [n]})

1.  Construct ring R = (P_1, ..., P_n) with P_  = P
2.  Compute key image I = x · H_p(P)
3.  For each i  [n], compute H_i = s_i · G where s_i = H("key_image_base" ‖ P_i)
4.  Sample k ←$ Z_
5.  Compute L_  = kG, R_  = kH_
6.  c_{ +1} = H(m ‖ L_  ‖ R_ )
7.  For i =  +1, ..., n, 1, ...,  -1 (mod n):
8.      Sample r_i ←$ Z_
9.      L_i = r_i · G + c_i · P_i    where P_i = H_p(P_i) mapped to G
10.     R_i = r_i · H_i + c_i · I
11.     c_{i+1} = H(m ‖ L_i ‖ R_i)
12. Set r_  = k - c_  · x
13. Return  = (I, {c_i, r_i}_{i [n]})
```

where indices are computed modulo $n$ and $\mathcal{H}_p(\cdot)$ maps public keys to Ristretto points.

#### 5.2.3 Verification Algorithm 4: Ring Signature Verification

```
Input: Ring R = (P_1, ..., P_n), key image I, message m,
       signature  = {c_i, r_i}_{i [n]}
Output: Accept or Reject

1.  For each i  [n]:
2.      P_i = H_p(P_i)
3.      H_i = s_i · G where s_i = H("key_image_base" ‖ P_i)
4.      L_i = r_i · G + c_i · P_i
5.      R_i = r_i · H_i + c_i · I
6.      c'_{i+1} = H(m ‖ L_i ‖ R_i)
7.  Accept if and only if the challenge chain is consistent
```

**Theorem 3** (Unforgeability). *Under the discrete logarithm assumption in $\mathbb{G}$, no PPT adversary who does not know any secret key $x_i$ corresponding to a ring member $P_i$ can produce a valid ring signature with non-negligible probability.*

*Proof sketch.* A successful forgery implies the ability to close the challenge chain without knowledge of any secret key. This requires computing $r_i = k - c_i x_i$ for some $i$ without knowing $x_i$, which reduces to computing the discrete logarithm of $P_i$. □

**Theorem 4** (Linkability). *Two ring signatures produced by the same secret key $x$ yield the same key image $I$, enabling double-spend detection.*

*Proof.* The key image $I = x \cdot \mathcal{H}_p^{(\mathrm{ki})}(P)$ is deterministic given $x$ and $P = xG$. Since $P$ is uniquely determined by $x$ in $\mathbb{G}$, the key image is a deterministic function of $x$ alone. □

### 5.3 Stealth Addresses

Stealth addresses ensure each transaction creates a unique, one-time destination address that only the intended recipient can detect and spend.

**5.3.1 Key Structure** Each agent publishes two public keys: - **Public view key** $A = aG$ — used by senders to create stealth addresses - **Public spend key** $B = bG$ — used in the one-time address derivation

The corresponding secret keys $(a, b)$ are derived via HKDF from the agent's master key (Section 6.4).

**5.3.2 Address Generation Protocol** **Definition 9** (Stealth Address). Given recipient keys $(A, B)$, the sender generates a one-time address as follows:

1. Sample ephemeral key $r \xleftarrow{\$} \mathbb{Z}_\ell$
2. Compute transaction public key $R = \mathcal{H}(\texttt{"ephemeral\_sender"} \,\|\, r \,\|\, \text{entropy})$
3. Compute shared secret $s = \mathcal{H}(\texttt{"stealth\_shared\_secret"} \,\|\, r \,\|\, A)$
4. Compute one-time key $P = \mathcal{H}(\texttt{"stealth\_one\_time"} \,\|\, s \,\|\, B)$

The tuple $(P, R)$ constitutes the stealth address, with $R$ included in the transaction for recipient scanning.

**5.3.3 Transaction Scanning** The recipient scans each transaction using their private view key $a$:

1. Recompute $s' = \mathcal{H}(\texttt{"stealth\_shared\_secret"} \,\|\, a \,\|\, A)$
2. Recompute $P' = \mathcal{H}(\texttt{"stealth\_one\_time"} \,\|\, s' \,\|\, B)$
3. Accept if $P' = P$

### 5.4 Pedersen Commitments

Transaction amounts are concealed using Pedersen commitments [12] on the Ristretto curve.

**Definition 10** (Pedersen Commitment). A commitment to value $v \in \mathbb{Z}_\ell$ with blinding factor $r \xleftarrow{\$} \mathbb{Z}_\ell$ is:

$$C(v, r) = vH + rG$$

where $G$ is the Ristretto basepoint and $H$ is a secondary generator derived as:

$$H = \texttt{RistrettoPoint::from\_uniform\_bytes}(h_1 \,\|\, h_2)$$

with $h_1 = \mathcal{H}(\texttt{"agentchain\_value\_generator"})$ and $h_2 = \mathcal{H}(\texttt{"agentchain\_value\_generator\_2"} \,\|\, h_1)$.

The "nothing-up-my-sleeve" construction of $H$ ensures no party knows $\log_G(H)$.

**Theorem 5** (Perfect Hiding). *The Pedersen commitment scheme is perfectly hiding: for any value $v$, the commitment $C(v, r)$ with uniformly random $r$ is uniformly distributed in $\mathbb{G}$.*

*Proof.* For any fixed $v$, the map $r \mapsto vH + rG$ is a bijection on $\mathbb{G}$ (since $G$ generates the group). Therefore $C(v, r)$ is uniformly distributed when $r$ is uniform. □

**Theorem 6** (Computational Binding). *Under the discrete logarithm assumption, no PPT adversary can find $(v, r) \neq (v', r')$ such that $C(v, r) = C(v', r')$ with non-negligible probability.*

*Proof.* Suppose $vH + rG = v'H + r'G$. Then $(v - v')H = (r' - r)G$, giving $H = \frac{r'-r}{v-v'}G$ (assuming $v \neq v'$). This yields $\log_G(H)$, contradicting the discrete logarithm assumption on the nothing-up-my-sleeve generator $H$. □

**5.4.1 Homomorphic Balance Verification** For a transaction with input commitments $\{C_{\text{in}}^{(j)}\}$ and output commitments $\{C_{\text{out}}^{(k)}\}$ with fee $f$:

$$\sum_j C_{\text{in}}^{(j)} = \sum_k C_{\text{out}}^{(k)} + fH$$

Expanding:

$$\sum_j (v_j^{\text{in}} H + r_j^{\text{in}} G) = \sum_k (v_k^{\text{out}} H + r_k^{\text{out}} G) + fH$$

This holds if and only if $\sum v_j^{\text{in}} = \sum v_k^{\text{out}} + f$ and $\sum r_j^{\text{in}} = \sum r_k^{\text{out}}$.

**5.4.2 Amount Encryption** Amounts are encrypted for the recipient using an XOR mask derived from the blinding factor:

$$\text{mask} = \mathcal{H}(\texttt{"amount\_encryption"} \,\|\, r), \quad \text{ct} = v_{\text{bytes}} \oplus \text{mask}[0..8]$$

**5.4.3 Range Proofs** To prevent negative amounts (which would allow inflation through commitment arithmetic), each commitment includes a range proof demonstrating $v \in [0, 2^{64})$. The current implementation uses bit-decomposition proofs:

For each bit $i \in \{0, 1, \ldots, 63\}$:

$$C_i = b_i H + r_i G$$

where $b_i = (v \gg i) \wedge 1$ is the $i$-th bit and $r_i = \mathcal{H}(r \,\|\, i)$. Each $C_i$ is verified to be a commitment to either 0 or 1, and $\sum_{i=0}^{63} 2^i C_i = C(v, r')$ for appropriate blinding.

A planned upgrade to Bulletproofs [13] will reduce proof size from $O(n)$ to $O(\log n)$ group elements (Section 15).

### 5.5 Key Image Set and Double-Spend Prevention

The protocol maintains a global set $\mathcal{I}$ of spent key images. When processing a transaction with ring signature containing key image $I$:

1. If $I \in \mathcal{I}$, reject the transaction (double-spend attempt).

2. Otherwise, add $I$ to $\mathcal{J}$ and accept.

Key images are persisted in the storage layer's `key_images` tree, indexed by the 32-byte key image value.

### 5.6 Decoy Selection

The quality of ring signature privacy depends critically on decoy selection. The protocol enforces the following selection criteria with default ring size $n_{\text{ring}} = 11$:

| Parameter | Value | Purpose |
|---|---|---|
| Ring size | 11 | Anonymity set cardinality |
| Max output age | 1,800 blocks (~12 min) | Temporal plausibility |
| Min output age | 10 blocks | Confirmation requirement |
| Amount variance | 20% | Statistical indistinguishability |
| Age ratio bound | 3x | Temporal uniformity |

Selection employs inverse-age weighting with recency bias to match the empirical distribution of real transaction outputs, following the approach described in [9].

### 5.7 Selective Disclosure via View Keys

Agents may grant selective transparency to designated parties through time-bounded view key grants:

$$\text{grant} = (\text{granter}, \text{grantee}, \mathcal{H}(\text{view\_key}), t_{\text{start}}, t_{\text{end}}, \pi)$$

where $\pi$ specifies the permission set: $\{\text{incoming}, \text{outgoing}, \text{amounts}, \text{metadata}\}$. Two preset configurations are provided: *audit mode* (all permissions) and *incoming-only* (incoming transactions and amounts).

---

## 6. Identity System

### 6.1 Agent Decentralized Identity

**Definition 11** (AgentDID). An Agent Decentralized Identity is a tuple:

$$\text{AgentDID} = (\text{id}, \text{pk}, t_{\text{created}}, C, \rho, \mu, M)$$

where $\text{id} = \mathcal{H}(\text{pk}) \in \{0,1\}^{256}$ is the deterministic identifier, pk is the Ed25519 public key, $t_{\text{created}}$ is the creation timestamp, $C \subseteq \{\texttt{Compute}, \texttt{Posting}, \texttt{Trading}, \texttt{Analysis}, \texttt{Storage}, \texttt{Messaging}, \texttt{Validation}, \texttt{Bridge}\}$ is the capability set, $\rho$ is the reputation score, $\mu$ is cumulative revenue, and $M$ is agent metadata.

Capabilities are self-declared and serve as metadata for service discovery; they are not enforced at the protocol level.

### 6.2 Reputation Scoring

**Definition 12** (Reputation Score). The composite reputation score $\rho \in [0, 1000]$ is computed as:

$$\rho = \text{clamp}\left( \underbrace{\frac{T^+}{T^+ + T^-} \cdot 300}_{task\ performance} + \underbrace{\frac{\eta}{100} \cdot 200}_{uptime} + \underbrace{\min(|E|, 50) \cdot 2}_{endorsements} + \underbrace{\log_2(u) \cdot 10}_{utility} - \underbrace{|S| \cdot 50}_{slashes},\ 0,\ 1000 \right)$$

where $T^+, T^-$ are completed and failed tasks, $\eta$ is uptime percentage, $E$ is the endorsement set, $u$ is the utility score, and $|S|$ is the number of slashing events.

### 6.3 Key Management

Agent keys are managed through an encrypted keystore secured by Argon2id [14] password hashing, HKDF-SHA256 key derivation, and AES-256-GCM authenticated encryption.

Each agent possesses three key types derived hierarchically from a master signing key:

1. **Signing key** (Ed25519): transaction authorization
2. **View key** (HKDF-derived): stealth address scanning (`info` = `"agentchain_view_key"`)
3. **Spend key** (HKDF-derived): stealth address spending (`info` = `"agentchain_spend_key"`)

Public keys are derived via scalar multiplication: $\text{pk} = \text{Scalar}(\text{sk}) \cdot G$.

Child keys for purpose-specific operations are derived deterministically:

$$\text{sk}_{\text{child}} = \mathcal{H}(\text{sk}_{\text{master}} \,\|\, \texttt{purpose} \,\|\, \text{index})$$

---

## 7. Transaction Model

### 7.1 Transaction Types

AgentChain supports 11 native transaction types:

| Type | Purpose | Key Fields |
|---|---|---|
| `Transfer` | Token transfer | recipient, amount |
| `X402Payment` | Service payment | provider, resource URI, amount, response hash, latency |
| `RegisterAgent` | Identity registration | serialized AgentDID |
| `UtilityProof` | Work proof submission | work type, evidence, points |
| `Message` | Agent-to-agent message | recipient, channel, payload, encryption flag |
| `ContractDeploy` | WASM contract deployment | bytecode, constructor args, gas limit |
| `ContractCall` | Contract invocation | contract ID, method, parameters |
| `Endorse` | Reputation endorsement | target agent |
| `BridgeDeposit` | Inbound bridge transfer | source chain, source tx, token, amount |

| Type | Purpose | Key Fields |
|---|---|---|
| `BridgeWithdraw` | Outbound bridge transfer | target chain, target address, token, amount |
| `GovernanceVote` | Protocol governance | proposal ID, vote |

### 7.2 Transaction Structure

A transaction $\tau$ consists of:

$$\tau = (\text{hash}, \text{from}, \text{type}, \text{nonce}, t, \sigma, f)$$

where $\text{hash} = \mathcal{H}(\text{from} \,\|\, \text{type} \,\|\, \text{nonce} \,\|\, t)$, $\sigma = \text{Sign}(\text{sk}, \text{hash})$ is an Ed25519 signature (64 bytes), and $f$ is the transaction fee.

Verification requires:

$$\text{Verify}(\text{pk}_{\text{from}}, \text{hash}, \sigma) = \text{true}$$

### 7.3 Nonce Management

Each agent maintains a sequential nonce counter. Transaction $\tau$ from agent $\mathcal{A}$ is valid only if $\tau.\text{nonce} = \mathcal{A}.\text{nonce}_{\text{expected}}$, preventing replay attacks.

### 7.4 Fee Distribution

Transaction fees are distributed according to a fixed protocol rule:

| Recipient | Share | Purpose |
|---|---|---|
| Serving agent (x402) or burn pool | 70% | Service provider reward |
| Block producer | 20% | Block production incentive |
| Burn | 10% | Deflationary mechanism |

Formally, for fee $f$:

$$f_{\text{service}} = \lfloor 0.70 \cdot f \rfloor, \quad f_{\text{producer}} = \lfloor 0.20 \cdot f \rfloor, \quad f_{\text{burn}} = f - f_{\text{service}} - f_{\text{producer}}$$

### 7.5 Transaction Pool

Pending transactions are maintained in a fee-prioritized mempool with configurable maximum size. Block producers extract transactions in descending fee order to maximize revenue per block.

---

## 8. x402 Native Payment Protocol

### 8.1 Protocol Description

HTTP status code 402 ("Payment Required") was reserved in RFC 7231 [15] but never standardized for machine-to-machine payments. AgentChain elevates 402 to a first-class protocol primitive enabling atomic service-payment exchange.

### 8.2 Service Registry

Agents register paid endpoints through the x402 service registry:

$$\text{endpoint} = (\text{provider}, \text{URI}, \text{price}, \text{currency}, \text{desc}, C, \bar{\lambda}, N, R, \text{active})$$

where $\bar{\lambda}$ is the rolling average latency, $N$ is total requests served, and $R$ is cumulative revenue. The registry supports discovery by capability and price comparison.

### 8.3 Payment Flow

The x402 payment protocol proceeds as follows:

1. **Request.** Agent $A$ sends HTTP GET to agent $B$'s service endpoint.
2. **Challenge.** Agent $B$ responds with HTTP 402 including price and provider identity.
3. **Payment.** Agent $A$ submits an `X402Payment` transaction on-chain:
   - Fields: provider $B$, resource URI, amount, $\mathcal{H}(\text{response})$, latency
4. **Fulfillment.** Agent $B$ delivers the service response.

The `response_hash` field creates an on-chain receipt binding the payment to the specific service delivered.

### 8.4 Payment Channels

For high-frequency micropayments, off-chain payment channels reduce on-chain costs:

$$\text{channel} = (\text{id}, A, B, d_A, d_B, \beta_A, \beta_B, \text{nonce}, t_{\text{open}}, t_{\text{expire}})$$

where $d_A, d_B$ are initial deposits and $\beta_A, \beta_B$ are current balances. State updates are signed bilaterally; only opening and closing transactions are recorded on-chain.

### 8.5 Cross-Chain x402

AgentChain supports cross-chain x402 payments via the bridge system. An agent on Solana can pay for an AgentChain service through an atomic bridge-mediated flow with a 10-minute expiry for settlement.

---

## 9. Smart Contracts and WASM Virtual Machine

### 9.1 Execution Environment

AgentChain includes a WebAssembly [16] virtual machine (using the `wasmi` interpreter) for programmable contract logic. Contracts are compiled to WASM bytecode and executed in a sandboxed environment with deterministic gas metering.

### 9.2 Gas Model

All VM operations consume gas according to a fixed schedule:

| Operation | Gas Cost |
|---|---|
| Storage read | 100 |
| Storage write | 500 |
| Token transfer | 1,000 |
| Log emission | 50 |
| Base computation | 1 per instruction |

Default gas limit per contract call: $10^6$ gas units.

### 9.3 Contract Lifecycle

Contract deployment validates the WASM magic number (`\0asm`) and generates a deterministic contract address:

$$\text{addr}_{\text{contract}} = \mathcal{H}(\text{owner} \,\|\, h \,\|\, \texttt{"agentchain\_contract"})$$

where $h$ is the deployment block height.

Execution occurs within an `ExecutionContext` tracking gas consumption. If gas consumption exceeds the limit, execution reverts atomically.

---

## 10. Multi-Chain Bridge System

### 10.1 Overview

AgentChain connects to three external chains through a bridge system secured by a multi-signature committee:

- **Base:** primary bridge for the FREDOM token
- **Solana:** cross-chain agent economy
- **Ethereum:** DeFi connectivity

### 10.2 Bridge Committee

Bridge operations require threshold approval from a committee of agents:

**Definition 13** (Bridge Committee). A bridge committee is a tuple $(\mathcal{C}, t, e, \Sigma, \text{paused})$ where $\mathcal{C} = \{(id_i, \sigma_i, \rho_i)\}$ is the set of committee members with stakes $\sigma_i$ and reputations $\rho_i$, $t$ is the signature threshold (e.g., 3-of-5), $e$ is the committee epoch, $\Sigma = \sum \sigma_i$ is total stake, and paused is the emergency halt flag.

Threshold verification:

$$\text{Approve}(\text{sigs}) = |\{s \in \text{sigs} \mid s \in \mathcal{C} \wedge \text{active}(s)\}| \geq t \wedge \neg\text{paused}$$

### 10.3 Deposit Flow

1. User locks tokens on the source chain.
2. Relayer submits `DepositRequest` with source transaction hash, Merkle proof, amount, and recipient.
3. Committee members independently verify the source chain transaction.
4. Each verifying member signs the operation.
5. When $t$ signatures accumulate, the deposit is confirmed.
6. Equivalent tokens are minted on AgentChain.

### 10.4 Withdrawal Flow

1. Agent burns tokens on AgentChain.
2. `WithdrawalRequest` submitted with burn transaction hash.
3. Cooldown period applied based on amount:
   - $< 10^7$ tokens: 1 hour
   - $10^7$ to $10^8$ tokens: 24 hours
   - $> 10^8$ tokens: 72 hours
4. Committee signs after cooldown expiry.
5. Tokens released on external chain upon threshold approval.

### 10.5 Security Mechanisms

**Rate limiting.** Per-epoch volume cap of $10^9$ tokens limits maximum extractable value during committee compromise.

**Fraud proofs.** Any agent may challenge a bridge operation by posting a `FraudChallenge` with evidence and stake. Challenge types include: invalid source transaction, incorrect amount, double-spend, and invalid Merkle proof. A 7-day challenge window provides resolution time. Successful challengers receive the slashed committee member's stake; failed challengers forfeit their own.

**Emergency pause.** The committee may halt operations upon $\geq 90\%$ of total committee stake voting in favor, preventing further damage during detected compromise.

---

## 11. Network Layer

### 11.1 Transport Stack

The peer-to-peer network is built on libp2p [17] with the following transport configuration:

$$\text{TCP} \rightarrow \text{Noise}_{XX} \rightarrow \text{Yamux} \rightarrow \text{Application}$$

- **TCP:** standard transport with `nodelay` for low-latency messaging
- **Noise XX:** mutual authentication and encrypted channels [18]
- **Yamux:** stream multiplexing over a single connection

### 11.2 Message Propagation

The network implements five GossipSub [19] topics:

| Topic | Message Type |
|---|---|
| `agentchain/blocks` | Block propagation |
| `agentchain/transactions` | Transaction propagation |
| `agentchain/peer_status` | Peer status exchange |
| `agentchain/chain_sync` | Chain synchronization requests |
| `agentchain/ping` | Liveness probes |

GossipSub is configured with 10-second heartbeat intervals and strict validation mode (all messages validated before forwarding).

### 11.3 Peer Discovery

Three discovery mechanisms operate concurrently:

1. **mDNS:** local network discovery for development environments
2. **Kademlia DHT:** global distributed hash table for internet-scale discovery [20]
3. **Bootstrap nodes:** static seed nodes for initial network entry

### 11.4 Chain Synchronization

New or lagging nodes synchronize via:

1. Exchange `PeerStatusMessage` containing chain height and head hash.
2. Identify peers with greater chain height.
3. Request missing block ranges via `BlockRequestMessage`.
4. Validate and apply received blocks sequentially.

Sync requests are rate-limited to 30-second intervals.

### 11.5 JSON-RPC Interface

An `axum`-based HTTP server exposes the following endpoints:

| Method | Path | Description |
|---|---|---|
| GET | `/chain_info` | Chain height, peer count, consensus state |
| GET | `/block/{height}` | Block by height |
| GET | `/block/hash/{hash}` | Block by hash |
| GET | `/account/{agent_id}` | Account balance and state |

| Method | Path | Description |
|---|---|---|
| POST | `/submit_transaction` | Submit signed transaction |
| GET | `/peers` | Connected peer list |
| GET | `/mempool` | Pending transaction set |
| GET | `/consensus` | Consensus state and validator information |

---

## 12. Storage

### 12.1 Database

AgentChain uses sled [21], an embedded B-tree database providing ACID transactions, lock-free concurrent reads, automatic crash recovery, zero-copy reads, and built-in compression.

### 12.2 Data Organization

Data is organized into seven logical trees:

| Tree | Key | Value |
|---|---|---|
| `blocks` | height (u64 LE) | Serialized `Block` |
| `blocks_by_hash` | hash ([u8; 32]) | Serialized `Block` |
| `accounts` | agent_id ([u8; 32]) | Serialized `Account` |
| `transactions` | tx_hash ([u8; 32]) | (Transaction, block height) |
| `key_images` | key_image ([u8; 32]) | block height (u64) |
| `metadata` | string keys | Chain metadata, state root, supply |
| `consensus` | string keys | Validator set, consensus parameters |

All values are serialized using `bincode` for compact binary representation.

---

## 13. Tokenomics and Incentive Analysis

### 13.1 Token Parameters

| Parameter | Value |
|---|---|
| Token name | AGENT |
| Bridged token | FREDOM (from Base) |
| Total supply | $10^9$ (fixed) |
| Pre-mine | 0% at mainnet |
| Distribution | 100% utility mining |

### 13.2 Fee Distribution Model

Transaction fees follow the distribution specified in Section 7.4:

$$f = f_{\text{service}} + f_{\text{producer}} + f_{\text{burn}}$$

The 10% burn creates deflationary pressure: let $S_t$ denote circulating supply at time $t$ and $F_t$ the cumulative fees. Then:

$$S_t = S_0 + M_t - 0.10 \cdot F_t$$

where $M_t$ is cumulative minting from block production and bridge deposits. In steady state, if the burn rate exceeds the minting rate, the supply is asymptotically deflationary.

### 13.3 Game-Theoretic Analysis

We model validator behavior as a repeated game and analyze incentive compatibility.

**Definition 14** (Validator Strategy). A validator strategy $\sigma_i$ specifies, for each round: (a) whether to perform useful work (cost $c_w$), (b) whether to produce blocks honestly when selected, and (c) whether to attest honestly.

**Theorem 7** (Incentive Compatibility). *Under the PoU mechanism, honest behavior is a Nash equilibrium for rational validators when the expected reward exceeds the cost of work.*

*Proof sketch.* Consider a validator $v_i$ with utility score $u_i$ and production weight $w_i$. In each epoch of 100 slots:

- **Honest strategy:** $v_i$ performs useful work, accumulating utility points. Expected block production revenue per epoch is:

$$\mathbb{E}[R_i^{\text{honest}}] = \frac{w_i}{\sum_j w_j} \cdot 100 \cdot \bar{f}_{\text{producer}}$$

where $\bar{f}_{\text{producer}}$ is the mean producer fee per block.

- **Deviation: no useful work.** If $v_i$ stops performing work, $u_i$ stagnates while competitors' scores increase. Over time, $w_i / \sum_j w_j \to 0$, and expected revenue converges to zero.
- **Deviation: dishonest block production.** Double production triggers 100% slashing, yielding expected loss $-u_i$. The expected gain from equivocation (e.g., attempting to double-spend) is bounded by the transaction value $v_{\text{tx}}$. Dishonesty is irrational when $u_i > v_{\text{tx}}$, which holds for established validators.
- **Deviation: withholding attestations.** Missed attestations reduce $r_i^{(\text{att})}$, lowering $w_i$ and future revenue. The marginal cost of attesting is negligible (one signature computation), so attestation is strictly dominant. □

**Lemma 1** (Sybil Resistance). *Creating $k$ Sybil identities does not increase an adversary's expected block production revenue compared to concentrating utility in a single identity.*

*Proof.* Let the adversary's total utility be $U$. With one identity: expected revenue $\propto U/(U+U_{\text{others}})$. With $k$ identities of utility $U/k$ each: expected revenue $\propto k \cdot (U/k)/(U+U_{\text{others}}) = U/(U+U_{\text{others}})$. The quantities are equal, and creating multiple identities incurs additional operational costs. □

### 13.4 Economic Equilibrium

The token economy reaches equilibrium when:

$$\text{Marginal cost of utility work} = \text{Marginal expected block production revenue}$$

At equilibrium, the aggregate useful work supplied by validators is maximized given the fee market, and the burn mechanism ensures long-run token value support through supply reduction.

---

## 14. Security Analysis

### 14.1 Threat Model

We consider an adversary $\mathcal{A}$ with the following capabilities:

| Threat Class | Adversary Capability |
|---|---|
| Byzantine validators | Controls $f < n/3$ validators with arbitrary behavior |
| Network adversary | Can delay (but not permanently prevent) message delivery |
| Privacy adversary | Observes all public transaction data; performs timing analysis |
| Bridge adversary | Controls up to $t - 1$ bridge committee members |
| Economic adversary | Holds significant token supply; can submit arbitrary transactions |
| Computational | Classical computation (polynomial time); no quantum capability |

### 14.2 Consensus Security

**Theorem 8** (Byzantine Fault Tolerance). *The AgentChain consensus protocol provides safety and liveness under the assumption $f < n/3$.*

*Safety* follows from Theorem 1: finalized blocks cannot conflict under the supermajority threshold. *Liveness* follows from Theorem 2: honest majority ensures block production and finalization progress.

**Attack 1: Long-Range Attack.** An adversary with old validator keys attempts to construct an alternative chain from a historical fork point.

*Defense:* Finality checkpoints prevent rewriting history beyond the latest finalized block. The `slashed_epochs` set permanently excludes compromised validators from future participation.

*Bound:* The adversary must control $\geq n/3$ validator keys that were active at the target epoch. Since slashed validators are removed, this requires compromising currently active validators.

**Attack 2: Nothing-at-Stake.** A validator produces blocks on multiple competing forks simultaneously.

*Defense:* The double production detection mechanism identifies two valid blocks $(B_1, B_2)$ from the same producer in the same slot. Upon detection, the validator's entire utility score is slashed and they are deactivated.

*Bound:* Expected gain from equivocation is $\leq \bar{f}_{\text{block}}$ (one block's fee). Expected loss is $u_i$ (entire utility score). Equivocation is irrational when $u_i > \bar{f}_{\text{block}}$, which holds for any validator that has produced more than one block.

**Attack 3: Utility Score Inflation.** An adversary submits fraudulent utility proofs to inflate their score.

*Defense:* Utility proofs require verifiable evidence (on-chain payment receipts, attestation signatures, bridge proofs). Each proof type has independent verification criteria that cannot be forged without performing the underlying work.

### 14.3 Privacy Security

**Ring Signature Anonymity.** With ring size $n_{\text{ring}} = 11$, the adversary's advantage in identifying the true signer is:

$$\text{Adv}_{\mathcal{A}}^{\text{anon}} \leq \frac{1}{n_{\text{ring}}} + \epsilon_{\text{decoy}}$$

where $\epsilon_{\text{decoy}}$ captures information leakage from imperfect decoy selection (amount and timing correlation). The decoy selection algorithm bounds $\epsilon_{\text{decoy}}$ by enforcing 20% amount variance and 3x age ratio constraints.

**Commitment Security.** By Theorems 5 and 6, Pedersen commitments are perfectly hiding and computationally binding under the discrete logarithm assumption.

**Key Image Collision Resistance.** Two distinct secret keys $x \neq x'$ produce distinct key images with overwhelming probability, since $I = x \cdot \mathcal{H}_p(xG) \neq x' \cdot \mathcal{H}_p(x'G)$ unless the adversary can find a collision in the hash-to-point function.

### 14.4 Bridge Security

**Threshold Security.** With a $t$-of-$m$ committee (default 3-of-5), the adversary must compromise $\geq t$ members to forge a bridge operation.

**Maximum Extractable Value.** In the worst case (committee compromise), the adversary can extract at most:

$$V_{\max} = \min(V_{\text{epoch\_limit}},\ V_{\text{locked}})$$

where $V_{\text{epoch\_limit}} = 10^9$ is the per-epoch volume cap and $V_{\text{locked}}$ is the total value locked in the bridge. Cooldown periods for large withdrawals provide additional detection time.

### 14.5 Known Limitations

| Limitation | Planned Mitigation | Timeline |
|---|---|---|
| Hash-based VRF (not EC-VRF) | Upgrade to ECVRF [7] | v0.2 |
| Bit-decomposition range proofs | Bulletproofs [13] | v0.2 |
| Simplified ring signature verification | Full CLSAG [11] | v0.2 |
| Single-node testnet | Multi-node deployment | Q2 2026 |

| Limitation | Planned Mitigation | Timeline |
|---|---|---|
| No formal verification | TLA+ model checking | Q3 2026 |

---

## 15. Roadmap

### Completed Phases

**Phase 1 — Core State Machine.** Block structure, Merkle trees, 11 transaction types, agent identity (AgentDID) and reputation system, PoU consensus skeleton, privacy primitives, account state model, sled-backed persistent storage, genesis configuration.

**Phase 2 — Cryptographic Primitives.** Ed25519 transaction signatures, Curve25519 stealth addresses, SAG ring signatures on Ristretto, Pedersen commitments with range proofs, Argon2 + AES-256-GCM key management, HKDF hierarchical derivation.

**Phase 3 — Networking.** libp2p node with TCP + Noise + Yamux, mDNS and Kademlia peer discovery, GossipSub message propagation, chain synchronization protocol, JSON-RPC server.

**Phase 4 — Production Consensus.** VRF-based leader selection, 400ms slot timing, 100-block epoch transitions, 2/3 finality gadget, heaviest-utility fork choice, four slashing conditions.

**Phase 5 — Bridge System.** Base, Solana, and Ethereum bridges with threshold committee, fraud proofs and challenge system, rate limiting and cooldown, emergency pause, cross-chain x402 payments.

**Phase 6 — WASM VM.** wasmi-based execution, gas metering, contract deployment and storage, host function interface.

**Phase 7 — Privacy Hardening.** Production ring signatures, view key grants and scanning, decoy selection algorithm, key image persistence, private x402 payments.

### Planned Phases

**Phase 8 — Testnet Launch (Q2 2026).** Multi-node testnet, Docker Compose orchestration, block explorer, Python and TypeScript SDKs, testnet faucet, monitoring infrastructure.

**Phase 9 — Mainnet Preparation (Q3 2026).** Formal security audit, Bulletproofs upgrade, ECVRF upgrade, performance benchmarking, governance framework, developer documentation.

**Phase 10 — Mainnet Launch (Q4 2026).** Zero pre-mine genesis, bridge activation, validator onboarding, multi-language SDK releases.

---

## 16. Conclusion

AgentChain presents a blockchain protocol designed from first principles for autonomous AI agent operation. The Proof of Utility consensus mechanism aligns validator incentives with network-beneficial work. The CryptoNote-derived privacy architecture provides transaction confidentiality

through ring signatures, Pedersen commitments, and stealth addresses. The native x402 payment protocol enables atomic micropayment exchange without smart contract intermediation. The threshold-secured bridge system provides interoperability with existing blockchain ecosystems.

The implementation comprises approximately 10,700 lines of Rust across 12 modules with Ed25519 signatures, Ristretto-group ring signatures, Pedersen commitments, libp2p networking, sled-backed storage, and a WebAssembly contract runtime.

Formal analysis demonstrates that the consensus mechanism provides Byzantine fault tolerance for $f < n/3$ adversarial validators, and game-theoretic arguments establish incentive compatibility of the validator reward structure. The privacy architecture provides information-theoretically hiding commitments and computationally unforgeable ring signatures under standard cryptographic assumptions.

The development path from the current implementation to mainnet requires hardening the cryptographic primitives (Bulletproofs, ECVRF), scaling the network (multi-node testnet), and activating bridge infrastructure. Each phase advances the system toward its design goal: providing autonomous AI agents with sovereign, private, and self-sustaining economic infrastructure.

---

## References


[1] V. Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform," 2014. [Online]. Available: https://ethereum.org/whitepaper

[2] A. Yakovenko, "Solana: A new architecture for a high performance blockchain," 2018. [Online]. Available: https://solana.com/solana-whitepaper.pdf

[3] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 910–927.

[4] N. van Saberhagen, "CryptoNote v 2.0," 2013. [Online]. Available: https://cryptonote.org/whitepaper.pdf

[5] M. Hamburg, "Decaf: Eliminating cofactors through point compression," in *Advances in Cryptology — CRYPTO 2015*, Springer, 2015, pp. 705–723.

[6] S. Micali, M. Rabin, and S. Vadhan, "Verifiable Random Functions," in *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999, pp. 120–130.

[7] S. Goldberg, L. Reyzin, D. Papadopoulos, and J. Vcelak, "Verifiable Random Functions (VRFs)," IETF RFC 9381, 2023.

[8] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining GHOST and Casper," 2020. [Online]. Available: https://arxiv.org/abs/2003.03052

[9] S. Noether, A. Mackenzie, and the Monero Research Lab, "Ring Confidential Transactions," *Ledger*, vol. 1, pp. 1–18, 2016.

[10] R. L. Rivest, A. Shamir, and Y. Tauman, "How to Leak a Secret," in *Advances in Cryptology — ASIACRYPT 2001*, Springer, 2001, pp. 552–565.

[11] B. Goodall, A. Noether, and the Monero Research Lab, "Concise Linkable Spontaneous Anonymous Group Signatures for Ad Hoc Groups (CLSAG)," 2019.

[12] T. P. Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing," in *Advances in Cryptology — CRYPTO '91*, Springer, 1992, pp. 129–140.

[13] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short Proofs for Confidential Transactions and More," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 315–334.

[14] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications," in *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 292–302.

[15] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," IETF RFC 7231, 2014.

[16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[17] Protocol Labs, "libp2p Specification," 2019. [Online]. Available: https://libp2p.io/specs/

[18] T. Perrin and M. Marlinspike, "The Noise Protocol Framework," 2018. [Online]. Available: https://noiseprotocol.org/noise.html

[19] D. Vyzovitis, Y. Naber, D. Dias, J. Serafini, and L. Lopes, "GossipSub: Attack-Resilient Message Propagation in the Filecoin and ETH2.0 Networks," 2020. [Online]. Available: https://arxiv.org/abs/2007.02754

[20] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," in *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002, pp. 53–65.

[21] T. Jager, "sled: An embedded database written in Rust," 2020. [Online]. Available: https://github.com/spacejam/sled

[22] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies," in *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 51–68.

[23] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.