Proof of Trust — protocol specification

Contents

Proof of Trust — Protocol Specification v0.3

This is the normative reference for the Proof of Trust (PoT) blockchain as implemented by the `pot-core`, `pot-git`, and `pot-cli` crates. Where the implementation and this document disagree, the implementation is authoritative for v0.3.

Changelog

- **v0.4** — Two facets (in progress): **(A) Society-of-nodes** — `norm_decls`, `norm_votes`, `norm_resolutions`, `conflict_records`, `conflict_patches`, `compute_attestations`, `rbac_*` on `Payload`; types in §§13–16. See `papers/society_of_aion_nodes.md`. **(B) Perpetual branch governance** — open code-proposal markets, HEAD from `branch_trust(oid)`, sequential KL scoring; §17 and `docs/perpetual_branch_governance_v04.md`. Scaffolding: `pot-core::branch_trust`, `pot-core::sequential_kl`.
- **v0.5 (specified)** — **Core predictions** — society hypotheses the chain exists to test; nomination stake + adoption market + linked outcome market; §18 and `docs/ips_ontology.md` §6. Types in `pot-core::core_prediction`; state machine not wired.

- **v0.3** — `Payload` gains three optional vectors (`code_commitments`, `code_proposals`, `code_proposal_votes`); new `SignedCodeProposal` and `SignedCodeCommitment` types; auto-resolving code-proposal markets with no human resolver; new git ref prefixes `refs/pot/code-commitments/*`, `refs/pot/code-proposals/*`, `refs/pot/proposal-votes/*`; new CLI verbs `commit-code`, `submit-proposal`, `bet-proposal`.
- **v0.2** — block-hash domain bumped to `"pot/block/v2"`; `Payload` gains three optional vectors (`market_decls`, `market_votes`, `market_resolutions`); new `markets` sub-state and KL-scored market resolution; new git ref prefixes `refs/pot/markets/*`, `refs/pot/market-votes/*`, `refs/pot/market-res/*`; new `pot-net` wire protocol (§11); new light-client verification rules (§12).

1. Objects and encoding

All protocol objects are serialized with canonical CBOR (RFC 8949) by the `ciborium` crate, using struct-field ordering (declaration order). No map type whose key ordering is not determined by its declaration may appear in any object that participates in a hash or signature. Byte strings are encoded as CBOR major type 2; pubkeys, signatures, hashes, and block ids are 32- or 64-byte strings.

The protocol hash function is `blake3-256` producing 32-byte digests. The signature scheme is Ed25519 (RFC 8032).

2. Types

2.1 Scalars

Name	Rust type	CBOR
`Hash`	`[u8; 32]`	byte-string(32)
`BlockId`	`Hash`	byte-string(32)
`PubKey`	`[u8; 32]`	byte-string(32)
`Signature`	`[u8; 64]`	byte-string(64)
`GitOid`	`String`	text-string (hex of git SHA-1 or SHA-256)

Trust balances are expressed in **millitrust** (`mT`); 1 trust = 1000 mT. Probabilities are expressed in **parts per million** (`ppm`); a distribution must have `probs_ppm` components that sum to exactly `PPM_TOTAL = 1_000_000`.

2.2 Genesis (`ChainParams`)

```
ChainParams {
  k_confirmations:       u64,        // K, finality depth
  alpha_mtrust_per_nat:  u64,        // alpha, trust-update multiplier
  initial_accounts:      Vec<(PubKey, u64)>, // (pubkey, millitrust)
  created_at:            i64,        // Unix seconds at creation
  note:                  String,
}
```

- `chain_id := blake3(CBOR(("pot/genesis/v1", ChainParams)))`.
- The genesis block id is defined to equal `chain_id`.

2.3 Trust transaction

```
TrustTxBody { from: PubKey, to: PubKey, amount: u64, nonce: u64, memo: String }
SignedTrustTx { body: TrustTxBody, sig: Signature }
```

- `sig_hash := blake3(CBOR(("pot/trusttx/v1", chain_id, body)))`.
- `sig` is verified against `body.from` on `sig_hash`.
- `nonce` must equal the current `accounts[from].nonce`.
- `amount` must be `<= accounts[from].trust_mtrust`.
- Applying: subtract `amount` from `from`, add to `to`, set `accounts[from].nonce += 1`. `to` may be any pubkey (no prior account creation needed).

2.4 Vote

```
VoteBody {
  voter:      PubKey,
  height:     u64,
  candidates: Vec<BlockId>,  // distinct
  probs_ppm:  Vec<u32>,      // length = candidates.len() + 1
}
SignedVote { body: VoteBody, sig: Signature }
```

- `probs_ppm.len() == candidates.len() + 1`; the trailing entry is the "none-of-these" slot.
- All entries of `probs_ppm` fit in `u32`; their sum is exactly `PPM_TOTAL = 1_000_000`.
- `candidates` must have no duplicates.
- `sig_hash := blake3(CBOR(("pot/vote/v1", chain_id, body)))`.

2.5 Block

```
BlockHeader {
  chain_id:     Hash,
  height:       u64,
  parent:       BlockId,
  proposer:     PubKey,
  payload_root: Hash,      // blake3(CBOR(("pot/payload/v2", Payload)))
  timestamp:    i64,
}
SignedBlockHeader { header: BlockHeader, sig: Signature }

Payload {
  code_commits:       Vec<GitOid>,
  trust_txs:          Vec<SignedTrustTx>,
  votes:              Vec<SignedVote>,
  // v0.2 additions - CBOR-optional (default `[]`).
  market_decls:       Vec<SignedMarketDecl>,
  market_votes:       Vec<SignedMarketVote>,
  market_resolutions: Vec<SignedMarketResolution>,
}

Block { header: SignedBlockHeader, payload: Payload }
```

- `BlockId := blake3(CBOR(("pot/block/v2", BlockHeader)))`.
- `sig` is the signature of `proposer` over `BlockId` bytes.
- The block is structurally valid iff `header.sig` verifies and `header.payload_root == blake3(CBOR(("pot/payload/v2", payload)))`.

2.6 Market (v0.2)

```
MarketDeclBody {
  chain_id:             Hash,
  proposer:             PubKey,
  question:             String,
  outcomes:             Vec<String>,  // distinct, 2..=64 entries
  resolver:             PubKey,
  alpha_mtrust_per_nat: u64,
  opens_at_height:      u64,
  closes_at_height:     u64,
  nonce:                u64,
}
SignedMarketDecl { body: MarketDeclBody, sig: Signature }

MarketVoteBody {
```

```
  market_id: MarketId,  // = blake3(CBOR(("pot/market/decl/v1", MarketDeclBody)))
  voter:     PubKey,
  height:    u64,
  probs_ppm: Vec<u32>,  // len == outcomes.len(); sums to PPM_TOTAL.
}
SignedMarketVote { body: MarketVoteBody, sig: Signature }

MarketResolutionBody {
  market_id: MarketId,
  resolver:  PubKey,
  winner:    u32,       // outcome index
  height:    u64,
}
SignedMarketResolution { body: MarketResolutionBody, sig: Signature }
```

Signature domains:

- `blake3(CBOR(("pot/market/decl/v1", MarketDeclBody)))` — decl.
- `blake3(CBOR(("pot/market/vote/v1", chain_id, MarketVoteBody)))` — vote.
- `blake3(CBOR(("pot/market/res/v1", chain_id, MarketResolutionBody)))` — res.

The `MarketId` equals the decl's sig-hash and therefore uniquely identifies the market across the chain. Declarations are rejected by the state machine if a decl with the same `MarketId` is already known, or if the sealing block's height is strictly greater than `opens_at_height`. Votes are rejected unless the market is `Open`, the voter has no prior vote on the market, the vote's `height` is inside `[opens_at_height, closes_at_height]`, and the sealing block's height is `>= vote.height`. Resolutions are only accepted from the designated resolver and only in blocks with height `> closes_at_height`.

3. State

3.1 Replicated state

```
AccountState { trust_mtrust: u64, nonce: u64 }
ChainState   {
  params:           ChainParams,
  accounts:         Map<PubKey, AccountState>,
  canonical:        Vec<BlockId>,           // canonical[0] = chain_id
  blocks:           Map<BlockId, Block>,
  trust_snapshots:  Vec<Map<PubKey, u64>>,  // length = canonical.len()
  vote_ledger:      Map<u64, Map<PubKey, SignedVote>>,
  scored_heights:   Set<u64>,
  scoring_reports:  Vec<ScoringReport>,
}
```

- `canonical[h]` is the block id at height `h` on `main`.
- `trust_snapshots[h]` is the `{pubkey -> trust_mtrust}` view *after* block `h` has been fully applied, **including** any scoring event triggered by appending block `h`.
- `vote_ledger[h][voter]` holds the single admissible signed vote by `voter` on height `h`.

3.2 Genesis

Starting state:

```
accounts = {}
for (pk, amt) in params.initial_accounts:
    accounts[pk].trust_mtrust += amt
canonical = [params.chain_id()]
trust_snapshots = [snapshot(accounts)]
```

4. Block application

Let the tip be at height `t`. Appending block `B` requires:

1. `B` is structurally valid (§2.5).
2. `B.header.chain_id == params.chain_id()`.
3. `B.header.height == t + 1`.
4. `B.header.parent == canonical[t]`.

Application order, applied **in sequence** inside the same block (§§4.1–4.5, then snapshot):

4.1 Trust transactions

For each `stx` in `payload.trust_txs`, in list order: - Verify `stx.verify(chain_id)`. - Check `stx.body.nonce == accounts[stx.body.from].nonce`. - Check `stx.body.amount <= accounts[stx.body.from].tr` - Apply: - `accounts[stx.body.from].trust_mtrust -= stx.body.amount` - `accounts[stx.body.from].nonce += 1` - `accounts[stx.body.to].trust_mtrust += stx.body.amount` (saturating)

4.2 Vote ingestion

For each `sv` in `payload.votes`: - Verify `sv.verify(chain_id)` (including the PPM sum invariant). - Require `1 <= B.header.height - sv.body.height <= K` (admissibility window of the last `K` heights). - Require `vote_ledger[sv.body.height]` has no existing entry for `sv.body.voter`. - Insert `sv` into `vote_ledger[sv.body.height][sv.body.voter]`.

Duplicate or out-of-window votes cause the whole block to be rejected.

4.3 Markets (v0.2)

After steps 4.1 and 4.2, **in this order**:

1. For each `SignedMarketDecl` `md` in `payload.market_decls`:
   - Verify `md.verify(chain_id)` (includes `chain_id` match and shape).
   - Require `md.market_id()` not already in `markets`.
   - Require `B.header.height <= md.body.opens_at_height`.
   - Insert `MarketState::open_from(md)` under its id.
2. For each `SignedMarketVote` `mv` in `payload.market_votes`:
   - Verify the signature on `mv`.
   - Require the market exists and has status `Open`.
   - Require `probs_ppm.len() == outcomes.len()` and the usual PPM sum.
   - Require `opens_at_height <= mv.body.height <= closes_at_height`.
   - Require `B.header.height >= mv.body.height` (no future-dating).
   - Require no prior vote by `mv.body.voter` on this market.
   - Insert into the market's `votes` map.
3. For each `SignedMarketResolution` `mr` in `payload.market_resolutions`:
   - Verify the signature on `mr`.
   - Require the market exists and is `Open` or `Closed`.
   - Require `mr.body.resolver == market.decl.body.resolver`.
   - Require `mr.body.winner < outcomes.len()`.
   - Require `B.header.height > market.decl.body.closes_at_height`.
   - Score the market (§5.5) and flip its status to `Resolved`.

4.4 Finalization trigger

After steps 4.1–4.3, if `B.header.height >= K + 1` **and** height `h' := B.header.height - K` is not in `scored_heights`, run the scoring routine of §5 for `h'`.

4.5 Snapshot

After all scoring is applied to `accounts`, append the current balance map to `trust_snapshots`, and append `B.id()` to `canonical`.

5. Scoring (§5.1–5.4)

Let `h` be the finalizing height (`canonical[h]` is its realized block `y`).

5.1 Trust weights

Weights for the scoring of height `h` are taken from `trust_snapshots[h - 1]` (i.e. the balances at the end of the parent height). This is the **consensus snapshot** for height `h`.

5.2 Aggregated candidate set

`candidates(h) := _v candidates_v` over all voters `v` that appear in `vote_ledger[h]`. The ordering is the order in which candidate ids first appear when iterating voters in `vote_ledger[h]` key order.

5.3 Market distribution

```
S     = sum_v trust_snapshot[v]     (over voters in vote_ledger[h])
p_m(c) = (1/S) * sum_v trust_snapshot[v] * prob_v(c)
p_m(none) = (1/S) * sum_v trust_snapshot[v] * residual_v(candidates(h))
```

where `prob_v(c)` is the voter's ppm for candidate `c` (divided by `PPM_TOTAL`), zero if the voter did not list `c`; and `residual_v` is the voter's own "none" slot plus the masses of listed candidates not in `candidates(h)`. If `S == 0`, the scoring routine records an empty report and terminates with no balance changes.

5.4 Per-voter update

For each voter `v` in `vote_ledger[h]`:

```
q_v(y) := voter's prob on the realized winner, falling back to the
          voter's own "none" slot if y is not in the voter's candidates
p_m(y) := p_m(winner) = p_m(y_idx) if y in candidates(h), else p_m(none)

s_v    = log( max(q_v(y), MIN_PROB) )   with MIN_PROB = 1e-9
s_m    = log( max(p_m(y), MIN_PROB) )
margin = s_v - s_m                      // nats
delta  = round(alpha_mtrust_per_nat * margin)
accounts[v].trust_mtrust = max(0, accounts[v].trust_mtrust + delta)
```

Log is natural. A `ScoringReport` is appended to `scoring_reports`, and `h` is inserted into `scored_heights`.

5.5 Market scoring (v0.2)

When a market is resolved with winner index `w`, the same strictly proper log score is applied to each admitted vote. Let `V` be the market's `votes` map, and let `t_v` be the trust balance of voter `v` in the latest `trust_snapshot`. Compute

```
S      = sum_{v in V} t_v                                (ignore t_v == 0)
p_m(i) = (1/S) * sum_{v in V} t_v * (probs_ppm_v[i] / PPM_TOTAL)
p_m(w) = max(p_m(w), MIN_PROB)
```

For each voter `v`:

```
q_v(w) = max(probs_ppm_v[w] / PPM_TOTAL, MIN_PROB)
s_v    = ln(q_v(w))
s_m    = ln(p_m(w))
delta  = round(market.alpha_mtrust_per_nat * (s_v - s_m))
accounts[v].trust_mtrust = max(0, accounts[v].trust_mtrust + delta)
```

A `MarketReport` (with the per-voter breakdown) is appended to `market_reports`, and the market's status becomes `Resolved { winner: w, market_probs, total_voter_trust }`.

6. Fork choice

At any moment, observed candidates at height `t + 1` are blocks signed by some proposer with `parent = canonical[t]`. For each candidate `B`,

```
W(B) = sum_v trust_snapshot_t[v] * q_v(B)   over votes in vote_ledger[t+1]
                                           that have been admitted OR
                                           currently pending in the
                                           off-chain mempool.
```

The protocol canonical block at height `t + 1` is

```
argmax_B  W(B),   ties broken by lexicographically smallest BlockId.
```

v0 uses *local* heaviest at each height (greedy). Section 7 of the whitepaper discusses the full cumulative rule.

7. Git encoding (transport)

The repository stores the chain on the `refs/heads/main` ref:

- The **root commit** is parentless, has tree `emptytree`, and its message contains a `PoT-Genesis: <base64(CBOR(ChainParams))>` trailer.
- Every **block commit** is a commit on `main` whose:
  - first parent is the previous block commit on `main`;
  - additional parents are the merged code-transaction branch heads;
  - tree is the last additional parent's tree if any, else the first parent's tree;
  - message body contains `PoT-Block: <base64(CBOR(Block))>`.
- **Block index**: `refs/pot/blocks/<block_id_hex>` points at the commit object for that block (canonical or candidate).
- **Candidate blocks**: `refs/pot/candidates/<block_id_hex>` → candidate commit. Off `main`.
- **Pending votes**: `refs/pot/votes/h-<height>/<voter_fp>` → a parentless commit whose message has `PoT-Vote: <base64(CBOR(SignedVote))>`.
- **Pending trust txs**: `refs/pot/trust-txs/<from_fp>-<nonce>` → parentless commit with `PoT-TrustTx: <base64(CBOR(SignedTrustTx))>`.
- **Pending market decls**: `refs/pot/markets/<mid_hex>` → parentless commit with `PoT-MarketDecl: <base64(CBOR(SignedMarketDecl))>`.
- **Pending market votes**: `refs/pot/market-votes/<mid_hex>/<voter_fp>` → parentless commit with `PoT-MarketVote: <base64(CBOR(SignedMarketVote))>`.
- **Pending market resolutions**: `refs/pot/market-res/<mid_hex>` → parentless commit with `PoT-MarketRes: <base64(CBOR(SignedMarketResolution))>`.
- Keys are stored locally (not in git) under `.git/pot/keys/<label>.key` as the 32-byte Ed25519 seed.

Base64 uses the standard alphabet without padding.

8. CLI command semantics

All commands accept `-C <path>` to select the repository (default: cwd).

Command	Effect
`pot clone URL [DIR] [--hub HUB] [--depth N] ...`	Clone via URL or hub slug; join PoT chain or convert vanilla git.
`pot register --hub H --slug S --name N [--git-url URL] [--snapshot-url URL] [--bootstrap host:port]... --as L`	Register local chain on aion-hub (signed).
`pot hub list --hub H` / `pot hub show --hub H --slug S`	Query hub manifest.
`pot hub serve ...`	Run `pot-aion-hub` HTTP registry.
`pot init --k K --alpha A --endow label=amt... --endow-pub pk=amt... [--note ...]`	Initialize a repo, generate keypairs for labeled endowments, write the genesis commit, and index it under `refs/pot/blocks/<chain_id>`.
`pot keygen --label L`	Generate and persist an Ed25519 keypair.

Command	Effect
`pot vote --height H --candidates id1,id2,... --probs p1,p2,...,none --as L`	Build and publish a signed vote as a ref under `refs/pot/votes/h-H/<fp>`. Accepts `--ppm` to give raw ppm integers.
`pot transfer --to PK_OR_LABEL --amount mT --as L [--memo ...]`	Queue a signed trust transaction under `refs/pot/trust-txs/...`. Nonce is derived from the replayed state + any already-pending txs by the sender.
`pot propose --as L [--canonical] [--merge GIT_OID]...`	Build the next block assembling all admissible pending votes and trust txs into `Payload`. Creates a candidate by default; `--canonical` appends directly to `main` and deletes consumed refs.
`pot advance [--block BLOCK_ID]`	Run fork choice on candidates at height `t+1` and promote the winner. Deletes consumed pending refs.
`pot finalize [--verbose]`	Replay and print finalization reports for every already-scored height.
`pot status`	Print chain id, tip, K, alpha, account balances, pending votes, pending txs, and candidate list.

9. Error conditions

- `BadGenesis`: root commit missing `PoT-Genesis` trailer or malformed.
- `InvalidBlock`: structural invariants violated, bad chain id, bad parent, bad height, bad payload root.
- `InvalidVote`: bad shape, bad PPM sum, duplicate candidate, wrong signing key, or inadmissible height window.
- `InvalidTx`: bad signature, bad nonce, insufficient balance.
- `DuplicateVote`: voter already has an entry in `vote_ledger[h]`.
- `InsufficientTrust`: sender trust below amount.

Any error during block application rolls back no state in v0: the block is simply rejected (the state machine reads from a fresh clone of the state for verification).

10. Constants (v0 defaults)

- `K = 2`
- `alpha = 1000` mT per nat
- `PPM_TOTAL = 1_000_000`
- `MIN_PROB = 1e-9`, `LOG_FLOOR = ln(MIN_PROB)  -20.7233`

13. Code proposals (v0.3)

A `SignedCodeProposal` is an **atomic bundle** of three protocol objects under one Ed25519 signature:

1. A pull-request declaration (the `git_oid` to be merged).
2. A two-outcome prediction-market declaration with **automatic resolution** (no human `resolver` field).
3. The author's initial probabilistic bet, cast before any other voter has seen the code.

The signature key is definitionally the author. Because the initial bet is embedded in the signed body, the author's vote is provably the first — placed at the moment of private knowledge, before gossip exposes the code to other voters.

13.1 Types

```
CodeCommitmentBody {
  chain_id:   Hash,
  author:     PubKey,
```

```
  code_hash:  Hash,   // blake3(CBOR(("pot/commitment/reveal/v1", git_oid, reveal_nonce)))
  nonce:      u64,
}
SignedCodeCommitment { body: CodeCommitmentBody, sig: Signature }
// commitment_id := blake3(CBOR(("pot/commitment/v1", CodeCommitmentBody)))
// sig verified against body.author on commitment_id

CodeProposalBody {
  chain_id:           Hash,
  author:             PubKey,
  git_oid:            GitOid,
  commitment_id:      Option<Hash>,   // links back to SignedCodeCommitment
  reveal_nonce:       Option<u64>,    // present iff commitment_id is present
  closes_at_height:   u64,
  alpha_mtrust_per_nat: u64,
  initial_probs_ppm:  Vec<u32>,       // exactly 2 entries, sums to PPM_TOTAL
  nonce:              u64,
}
SignedCodeProposal { body: CodeProposalBody, sig: Signature }
// proposal_id := blake3(CBOR(("pot/proposal/v1", CodeProposalBody)))
// sig verified against body.author on proposal_id

ProposalVoteBody {
  proposal_id:  ProposalId,
  voter:        PubKey,
  height:       u64,
  probs_ppm:    Vec<u32>,   // exactly 2 entries, sums to PPM_TOTAL
}
SignedProposalVote { body: ProposalVoteBody, sig: Signature }
// sig_hash := blake3(CBOR(("pot/proposal/vote/v1", chain_id, ProposalVoteBody)))
```

Outcome indices: **0 = merged**, **1 = expired** (deadline passed without merge).

Merge enforcement (v0.3.1): from canonical height **10** onward, every `code_commits` entry in a block payload must reference a git oid with an **open** code proposal already on chain (or introduced earlier in the same block's `code_proposals`). Heights below 10 are grandfathered for bootstrap imports. Use `pot submit-proposal` before `pot propose --merge`.

13.2 Payload additions

```
Payload {
  // … existing fields …
  // v0.3 additions - CBOR-optional (default []):
  code_commitments:     Vec<SignedCodeCommitment>,
  code_proposals:       Vec<SignedCodeProposal>,
  code_proposal_votes:  Vec<SignedProposalVote>,
}
```

13.3 State additions

```
ChainState {
  // … existing fields …
  code_commitments:        Map<Hash, SignedCodeCommitment>,
  code_proposals:          Map<ProposalId, CodeProposalState>,
  code_proposals_by_oid:   Map<GitOid, ProposalId>,   // index
  code_proposal_reports:   Vec<CodeProposalReport>,
}
```

13.4 Block application (v0.3 steps, after §4.1–4.5)

Step 6 — Ingest code commitments. For each `SignedCodeCommitment cc`: - Verify `cc` against `chain_id`. - Require `commitment_id(cc)` not already in `code_commitments`. - Insert under its id.

Step 7 — Ingest code proposals. For each `SignedCodeProposal cp`: - Verify `cp` against `chain_id`. - Require `proposal_id(cp)` not already in `code_proposals`. - Require `block.height <= cp.body.closes_at_height`. - If `commitment_id` is set: look up the stored commitment, verify `cp.body.author == commitment.body.author`, and re-derive `commitment_hash(git_oid, reveal_nonce) == commitment.body.code_hash`. - Create `CodeProposalState::open_from(cp, sealing_height)`, which records the author's `initial_probs_ppm` as the first `SignedProposalVote` in the state's `votes` map (with a zeroed signature — authorization is the proposal signature itself). - Insert into `code_proposals` and `code_proposals_by_oid`.

Step 8 — Admit code proposal votes. For each `SignedProposalVote pv`: - Verify signature against `chain_id`. - Require proposal exists and is `Open`. - Require `pv.body.probs_ppm.len() == 2` and sum `== PPM_TOTAL`. - Require `pv.body.height <= closes_at_height`. - Require `block.height >= pv.body.height`. - Require no prior vote by `pv.body.voter` on this proposal. - Insert into `proposal.votes`.

Step 9 — Auto-resolve proposals. Collect proposals to resolve (immutable scan, then mutate): - For each `git_oid` in `block.payload.code_commits`: if an open proposal maps to that oid, mark it **merged** (winner = 0). - For each open proposal where `block.height > closes_at_height`: mark it **expired** (winner = 1). Apply `resolve_proposal(pid, winner, block.height)` for each.

13.5 Proposal scoring

`resolve_proposal` applies the same strictly-proper KL log-score rule as §5.5:

```
snapshot = trust_snapshots[tip - 1]   (latest available before this block)
S        = sum_{v in votes} snapshot[v]
p_m(i)   = (1/S) * sum_{v in votes} snapshot[v] * (probs_ppm_v[i] / PPM_TOTAL)
p_m(w)   = max(p_m(w), MIN_PROB)

For each voter v:
  q_v(w)  = max(probs_ppm_v[w] / PPM_TOTAL, MIN_PROB)
  s_v     = ln(q_v(w))
  s_m     = ln(p_m(w))
  delta   = round(alpha_mtrust_per_nat * (s_v - s_m))
  accounts[v].trust_mtrust += delta   (clamped at 0)
```

A `CodeProposalReport` (per-voter breakdown) is appended to `code_proposal_reports`.

13.6 Git ref encoding (v0.3)

- **Pending commitments**: `refs/pot/code-commitments/<cid_hex>` → parentless commit with `PoT-CodeCommitment: <base64(CBOR(SignedCodeCommitment))>`.
- **Pending proposals**: `refs/pot/code-proposals/<pid_hex>` → parentless commit with `PoT-CodeProposal: <base64(CBOR(SignedCodeProposal))>`.
- **Pending proposal votes**: `refs/pot/proposal-votes/<pid_hex>/<voter_fp>` → parentless commit with `PoT-ProposalVote: <base64(CBOR(SignedProposalVote))>`.

13.7 CLI

```
# Optional privacy step: timestamp a hash commitment before revealing code.
pot commit-code --git-oid <sha> --reveal-nonce <u64> --as <key>

# Atomic PR + market + initial bet (all under one signature).
pot submit-proposal --git-oid <sha> --closes-at <height> \
    --probs <p_merged>,<p_expired> --as <key>
    [--commitment-id <hex> --reveal-nonce <u64>]   # optional: link commitment
    [--alpha <millitrust_per_nat>]
```

```
# Other voters betting on an open proposal.
pot bet-proposal --proposal <pid_hex> --probs <p_merged>,<p_expired> --as <key>
```

11. Wire protocol (v0)

The `pot-net` crate defines a length-prefixed CBOR frame protocol over TCP. Each frame on the wire is:

`[u32 big-endian body length] [body: canonical CBOR(Message)]`

The maximum body length is 16 MiB. The `Message` enum includes:

- `Hello { body: HelloBody, sig: Signature }` — connection initiator's greeting. `sig` is `Ed25519(blake3(CBOR(("pot/net/hello/v1", body))))`.
- `HelloAck { body: HelloBody, sig: Signature }` — responder. `sig` covers `blake3("pot/net/helloack/v1" || CBOR(body) || client_nonce.as_bytes())`. The client verifies this so the responder proves knowledge of the client's fresh nonce.
- `Ping { nonce: u64 }` / `Pong { nonce: u64 }`.
- `GetGenesis` / `GenesisData(ChainParams)`.
- `GetHeaders { from: u64, max: u32 }` / `Headers(Vec<SignedBlockHeader>)`.
- `GetBlock { block_id: Hash }` / `BlockData(Block)`.
- `Inv(Inventory)` / `GetData(Inventory)` / `Data(DataBundle)` — where `Inventory` is a struct of five digest vectors (votes, trust-txs, market decls/votes/resolutions) and `DataBundle` is the parallel set of signed objects.
- `GetPeers` / `Peers(Vec<PeerAddr>)`.
- `TipChanged { height, block_id }`.
- `Bye { reason: String }`.

11.1 Handshake

```
HelloBody {
  proto_version: u16,        // = 1
  chain_id:      Hash,
  node_id:       PubKey,
  tip_height:    u64,
  tip_id:        Hash,
  listen_port:   Option<u16>,
  nonce:         Hash,       // 32 random bytes
  user_agent:    String,     // len <= 256
}
```

The client sends `Hello { body, sig }`; the server verifies the signature and `chain_id == expected`, then returns `HelloAck { body, sig }` where `sig` covers the pair `(body, client.nonce)`. Either peer closes the connection on any mismatch.

11.2 Object digests

Peers refer to mempool objects by content digest in `Inv` and `GetData`:

`digest(obj) = blake3(CBOR(("pot/net/obj/v1", obj)))`

The domain tag prevents cross-use with on-chain hashes.

11.3 Gossip semantics

- Upon receiving `Inv`, a peer issues `GetData` for any digests it has not previously seen.
- Upon receiving `Data`, a peer validates each object with `pot-core`, writes it to the local repo, and emits a `Inv` for every newly-added digest to every connected peer *except* the sender.
- `TipChanged { height, block_id }` prompts peers whose local tip is lower to run `GetHeaders { from: local_height + 1, max: … }`.

12. Light client

A light client trusts only (a) the genesis `ChainParams` (or its hash), (b) proposer signatures, and (c) first-parent chain linkage. It does **not** trust the trust-weighted fork-choice rule; a bootstrap peer it connects to is authoritative for which chain is canonical.

```
verify_header_chain(headers, chain_id, prev_tip, prev_tip_h) -> Result<...>
verify_payload_root(block)                                   -> Result<()>
verify_genesis(params, expected_chain_id)                    -> Result<()>
```

The verification rules are:

- For every `SignedBlockHeader h` in `headers`:
  - `h.header.chain_id == chain_id`.
  - `h.header.height == prev_tip_h + i + 1`.
  - `h.header.parent == previous.id()`.
  - `h.verify()` (Ed25519 of `blake3(CBOR(("pot/block/v2", h.header)))` against `h.header.proposer`).
- For a fetched `Block`: `block.verify_structure()` passes **and** `block.id() == headers[block.height - 1].id()`.
- For the initial `GenesisData(params)`: `params.chain_id() == expected_chain_id`.

A snapshot is persisted as the CBOR of `(ChainParams, Vec<SignedBlockHeader>)`.

---

13. Norms on chain (v0.4)

Norms extend the v0.2 market machinery (§2.6) with society-wide rules that aion-core nodes propose, vote on, and ingest. See papers/society_of_aion_nodes.md.

13.1 Types

```
NormDeclBody {
  chain_id:             Hash,
  proposer:             PubKey,
  norm_id:              String,       // stable id, e.g. "no_friday_prod_write"
  statement:            String,       // falsifiable plain-language rule
  rationale:            String,
  scope_tags:           Vec<String>,
  target_pubkeys:       Vec<PubKey>,  // empty = society-wide
  resolver:             PubKey,
  alpha_mtrust_per_nat: u64,
  opens_at_height:      u64,
  closes_at_height:     u64,
  nonce:                u64,
}
SignedNormDecl { body: NormDeclBody, sig: Signature }

NormVoteBody {
  norm_id:   String,    // = NormId from decl
  voter:     PubKey,
  height:    u64,
  probs_ppm: Vec<u32>,  // len = 2: [ADOPT, REJECT]; sums to PPM_TOTAL
}
SignedNormVote { body: NormVoteBody, sig: Signature }

NormResolutionBody {
  norm_id:  String,
  resolver: PubKey,
  winner:   u32,        // 0 = ADOPT, 1 = REJECT
  height:   u64,
```

```
}
SignedNormResolution { body: NormResolutionBody, sig: Signature }
```

Signature domains:

- `blake3(CBOR(("pot/norm/decl/v1", NormDeclBody)))` — decl; `NormId` equals this hash as text-encoded hex or the `norm_id` field if unique.
- `blake3(CBOR(("pot/norm/vote/v1", chain_id, NormVoteBody)))` — vote.
- `blake3(CBOR(("pot/norm/res/v1", chain_id, NormResolutionBody)))` — resolution.

13.2 State

```
NormState {
  norm_id:    String,
  status:     enum { Open, Adopted, Rejected },
  decl:       SignedNormDecl,
  votes:      Map<PubKey, SignedNormVote>,
  resolution: Option<SignedNormResolution>,
}
```

Adopted norms are written to `refs/pot/norms/{norm_id}` as CBOR blobs. Nodes pull adopted norms into the local `aion-core/norms` service as **foreign norms**.

13.3 Payload extension (v0.4)

`Payload` gains optional vectors (default `[]`):

```
norm_decls:       Vec<SignedNormDecl>,
norm_votes:       Vec<SignedNormVote>,
norm_resolutions: Vec<SignedNormResolution>,
```

Git ref prefixes: `refs/pot/norms/{norm_id}`, `refs/pot/norm-votes/{norm_id}/{voter}`.

13.4 CLI (planned)

- `pot norm propose` — create `SignedNormDecl`.
- `pot norm vote` — submit `SignedNormVote`.
- `pot norm resolve` — submit `SignedNormResolution` (resolver only).

---

14. Conflict resolution and participant patches (v0.4 — specified)

14.1 ConflictRecord

```
ConflictRecordBody {
  chain_id:      Hash,
  conflict_id:   String,
  trigger:       enum { Friction, Violation, MarketDispute, ReplayRegression },
  laws_refs:     Vec<String>,   // norm_id references
  participants:  Vec<PubKey>,
  evidence_cid:  String,        // IPFS or git blob ref
  proposer:      PubKey,
  timestamp:     i64,
}
SignedConflictRecord { body: ConflictRecordBody, sig: Signature }
```

`conflict_id := hex(blake3(CBOR(("pot/conflict/v1", ConflictRecordBody))))`.

14.2 ParticipantPatch (tagged union)

All patches share a header; `kind` selects the variant:

```
PatchHeader {
  chain_id:    Hash,
  conflict_id: String,
  participant: PubKey,       // target node OR agent_type_id as UTF-8 convention
  author:      PubKey,
  timestamp:   i64,
}

ParticipantPatch = PatchHeader + one of:
  SystemPromptPatch     { diff: String }
  LearningsAppend       { markdown: String }
  LoraAdapterRequest    { training_run_id: String, base_profile: String }
  FullFinetuneRequest   { profile_id: String, training_run_id: String }
  RlPolicyUpdate        { policy_id: String, checkpoint_uri: String, content_hash: Hash }
```

CBOR encoding uses an integer tag `kind`:

kind	variant
0	`SystemPromptPatch`
1	`LearningsAppend`
2	`LoraAdapterRequest`
3	`FullFinetuneRequest`
4	`RlPolicyUpdate`

Signature domain:

- `blake3(CBOR(("pot/patch/v1", ParticipantPatch)))`.

Patches are sealed in the conflict record's payload or in block `Payload.conflict_patches: Vec<SignedParticipantPatch>`.

14.3 Resolution flow

1. `pot conflict open` — publish `SignedConflictRecord`.
2. Deliberation off-chain or via society meeting; author drafts patches.
3. `pot conflict resolve` — attach `Vec<SignedParticipantPatch>`; requires author trust `min_trust_for_patch` (genesis param).
4. After `K` confirmations, `pot conflict apply` on each node invokes local `aion-core/conflict_resolution/applier`

14.4 Payload extension (v0.4)

```
conflict_records: Vec<SignedConflictRecord>,
conflict_patches: Vec<SignedParticipantPatch>,
```

Git ref prefixes: `refs/pot/conflicts/{conflict_id}`, `refs/pot/patches/{conflict_id}/{patch_hash}`.

---

15. ComputeAttestation (v0.4 — specified)

15.1 Type

```
ComputeAttestationBody {
  voter:      PubKey,
  tokens_in:  u64,
  tokens_out: u64,
  profile_id: String,    // matches aion-core LlmProfileConfig id
  ref_id:     String,    // vote or norm vote this attests
  nonce:      u64,
}
SignedComputeAttestation { body: ComputeAttestationBody, sig: Signature }
```

- `sig_hash := blake3(CBOR(("pot/compute/v1", chain_id, ComputeAttestationBody)))`.
- `cost_units := tokens_in + tokens_out` (or weighted sum per genesis `cost_formula`).
- `cost_mtrust := beta_mtrust_per_unit * cost_units` where `beta = alpha / c_0`.

15.2 Scoring with cost

At finalize height `h`, for voter `v` on outcome `y`:

```
margin_nats = log q_v(y) - log p_m(y)
delta_raw   = alpha * margin_nats
delta_cost  = beta * cost_units(v)    // from SignedComputeAttestation if present, else 0
delta_t_v   = round(delta_raw - delta_cost)
t_v         = max(0, t_v + delta_t_v)
```

Reference: `score_vote_with_cost()` in `pot-core/src/scoring.rs` (planned).

15.3 Payload extension (v0.4)

```
compute_attestations: Vec<SignedComputeAttestation>,
```

15.4 CLI (planned)

- `pot attest-compute` — sign and publish attestation for a prior vote ref.

15.5 Changelog entry (v0.4)

- **v0.4** — `Payload` gains `norm_decls`, `norm_votes`, `norm_resolutions`, `conflict_records`, `conflict_patches`, `compute_attestations`; new types in `pot-norms` and `pot-conflict` crates; `score_vote_with_cost` in `pot-core`; society-of-nodes integration with `aion-core/blockchain_bridge`.

---

16. RBAC on chain (v0.4 — specified)

RBAC extends the society mirror with **role definitions and actor bindings** that each node's `aion-core/aion_rbac` service replays into its local `RbacStore`. Builtin roles (`owner`, `admin`, `manager`, `viewer`, …) are seeded locally and referenced by id; the chain carries **custom role declarations** and **signed grants/revokes**.

See docs/data_content.md for the full payload index.

16.1 Types

```
RbacRoleDeclBody {
  chain_id:      Hash,
  proposer:      PubKey,
  role_id:       String,        // stable id, e.g. "fleet_auditor"
  label:         String,
  permissions:   Vec<String>,   // domain:action, e.g. "norms:read"
  assignable_by: Vec<String>,    // role ids allowed to grant this role
  nonce:         u64,
}
SignedRbacRoleDecl { body: RbacRoleDeclBody, sig: Signature }
// role_decl_id := blake3(CBOR(("pot/rbac/role/decl/v1", RbacRoleDeclBody)))

RbacGrantBody {
  chain_id:    Hash,
  granter:     PubKey,
  actor_kind:  String,           // "human" | "agent" | "service"
  actor_id:    String,           // UUID, agent_type_id, or service name
  role_id:     String,
  nonce:       u64,
```

```
}
SignedRbacGrant { body: RbacGrantBody, sig: Signature }

RbacRevokeBody {
  chain_id:    Hash,
  revoker:     PubKey,
  actor_kind:  String,
  actor_id:    String,
  role_id:     String,
  nonce:       u64,
}
SignedRbacRevoke { body: RbacRevokeBody, sig: Signature }
```

Signature domains:

- `blake3(CBOR(("pot/rbac/role/decl/v1", RbacRoleDeclBody)))` — role decl.
- `blake3(CBOR(("pot/rbac/grant/v1", chain_id, RbacGrantBody)))` — grant.
- `blake3(CBOR(("pot/rbac/revoke/v1", chain_id, RbacRevokeBody)))` — revoke.

Permission strings use the same `domain:action` vocabulary as `aion-core/aion_rbac` (`tools:read`, `machine:write`, `norms:admin`, `rbac:assign_roles`, `*:*`, …). Wildcard rules (`domain:*`, `*:*`) are resolved at check time on each node.

16.2 State

```
RbacRoleState {
  role_id:       String,
  decl:          SignedRbacRoleDecl,
  status:        enum { Active, Retired },
}

actor_roles: Map<(actor_kind, actor_id), Set<role_id>>
```

Custom roles are written to `refs/pot/rbac/roles/{role_id}` after the declaring block is canonical. Actor bindings are derived by replaying `rbac_grants` and `rbac_revokes` in block order.

16.3 Payload extension (v0.4)

`Payload` gains optional vectors (default `[]`):

```
rbac_role_decls:  Vec<SignedRbacRoleDecl>,
rbac_grants:      Vec<SignedRbacGrant>,
rbac_revokes:     Vec<SignedRbacRevoke>,
```

Git ref prefixes:

- `refs/pot/rbac/roles/{role_id}` — pending or adopted role decl.
- `refs/pot/rbac/grants/{actor_kind}/{actor_id}/{role_id}` — pending grant.
- `refs/pot/rbac/revokes/{actor_kind}/{actor_id}/{role_id}` — pending revoke.

16.4 Block application (v0.4 steps, after §13 norm steps)

Step 10 — Ingest role declarations. For each `SignedRbacRoleDecl rd`: - Verify `rd` against `chain_id`. - Require `role_id(rd)` not already in `rbac_roles`. - Require `role_id` does not collide with a builtin role id on the node. - Insert `RbacRoleState::active_from(rd)`.

Step 11 — Apply grants. For each `SignedRbacGrant rg`: - Verify signature against `chain_id`. - Require `rg.body.role_id` exists (builtin or declared on chain). - Require `granter` holds `rbac:assign_roles` and either `owner` or a role listed in the target role's `assignable_by`. - Require no existing grant of the same `(actor_kind, actor_id, role_id)`. - Insert into `actor_roles`.

Step 12 — Apply revokes. For each `SignedRbacRevoke rr`: - Verify signature against `chain_id`. - Require `revoker` satisfies the same `assignable_by` rule as for grants. - Require an active grant exists; remove it from `actor_roles`.

After K confirmations, each node calls `aion_rbac.store.grant_role` / `revoke_role` for foreign bindings not originated locally.

16.5 Relationship to aion-core

Local (`aion-core`)	On chain
`RbacStore.grant_role` / `revoke_role`	`SignedRbacGrant` / `SignedRbacRevoke` in block payload
`BUILTIN_ROLES` in `aion_rbac/store.py`	Referenced by id; not re-declared on chain
`POST /rbac/...` (planned)	`pot rbac declare` / `pot rbac grant` / `pot rbac revoke`
`ActorContext` from HTTP headers	Replay chain grants for cross-node audit

16.6 CLI (planned)

- `pot rbac declare` — publish `SignedRbacRoleDecl`.
- `pot rbac grant` — publish `SignedRbacGrant`.
- `pot rbac revoke` — publish `SignedRbacRevoke`.
- `pot rbac list` — show society role catalog and actor bindings.

16.7 Changelog entry (v0.4, RBAC)

- **v0.4** — `Payload` gains `rbac_role_decls`, `rbac_grants`, `rbac_revokes`; new types in planned `pot-rbac` crate; `blockchain_bridge` publishes and ingests foreign RBAC; mirrors `aion-core/aion_rbac`.

17. Perpetual branch governance (v0.4 — specified, partial impl)

Normative design: docs/perpetual_branch_governance_v04.md. Mirrors intra-node sequential KL markets (aion-core/docs/architecture/market_math.md).

17.1 Goals

1. Code-proposal (and, by extension, society/norm) markets **do not settle**.
2. Canonical **git tree** at height `h+1` is the open `git_oid` with maximum **branch trust**, not the winner of a separate block-candidate auction alone.
3. **Newcomer damping:** author self-bets count at `min(mT_author, author_cap)`.
4. **Trust updates** use recurring observation `y_h = 1` iff `oid` is canonical at `h`, scored with sequential KL toward the live consensus distribution.

17.2 Branch trust

```
effective_mT(v, oid) = min(mT_v, author_cap)  if v == author(oid)
                     = mT_v                    otherwise

branch_trust(oid) = sum_v effective_mT(v, oid) * p_v(merged)

canonical_oid = argmax_oid branch_trust(oid)   // tie: smallest oid string
```

Default `author_cap_mtrust = 1000`. Reference: `pot_core::branch_trust`.

17.3 Perpetual proposals

From chain height `PERPETUAL_FROM_HEIGHT` (default **15** on the AGI network):

- `closes_at_height` MUST NOT trigger auto-expire.
- `ProposalStatus::Merged | Expired` are **deprecated** for new proposals.
- Status is `Open { on_head, head_since_height }` only.

Heights below `PERPETUAL_FROM_HEIGHT` retain v0.3 auto-resolve (grandfathered).

17.4 Scoring

Bet rows are trust-weighted probability vectors over `{merged, not_merged}`. Marginal value uses Bayesian sequential KL reduction (same rule as `aion-core/prediction_market/pm_sequential.rs`). Reference: `pot_core::sequential_kl::value_bayesian_sequential`.

At each finalize height `h`, for each open proposal:

```
y_h = merged   if oid == canonical_oid at h
    = not_merged otherwise

ΔmT_v +=   * (log q_v(y_h) - log p_market(y_h))   // per admissible bet window
```

17.5 Fork choice (supersedes §6 for v0.4 heights)

Block payload at `h+1` MUST include `canonical_oid` in `code_commits`. `pot advance` MUST reject candidates whose merge oid `argmax branch_trust` (unless grandfathered height). Block-level `W(B)` votes may remain as attestation in v0.4.0; branch trust is authoritative for the merged tree.

17.6 CLI (planned)

- `pot submit-proposal` — no `--closes-at`; sets `perpetual: true`.
- `pot propose --merge-best` — merge `argmax branch_trust` oid.
- `pot branch-rank` — print ranked open oids and `branch_trust` scores.
- Hub: **branch leaderboard** replaces **Resolved** market filter.

17.7 Implementation status

Component	Status
`pot-core::branch_trust`	implemented
`pot-core::sequential_kl`	implemented
`state.rs` perpetual mode + KL at finalize	implemented
`pot-cli` / hub UI	implemented
`deploy/sync-agi-into-chain.sh`	implemented (v0.4 from h15)

---

18. Core predictions (v0.5 — specified)

Normative design: `docs/ips_ontology.md` §6.

A **core prediction** is the society's **foundational hypothesis** — the falsifiable bet the chain exists to prove or disprove (e.g. "People want to stream movies at home", "KL-scored public forecasts allocate trust better than PoW"). It is **not** a generic society market: it is identity-defining world model content, curated through nomination and adoption.

Intra-node mirror: aion-core **root tasks** (`parent_id == ""`) — core tasks say what to **do**; core predictions say what must be **true**. See `aion-core/docs/architecture/core_tasks_priority.md`.

18.1 Semantics

Concept	Meaning
Hypothesis	Falsifiable claim about the external world (`hypothesis` field)
Rationale	Why this hypothesis defines the society (`rationale` field)
Phase A — adoption	Trust-weighted market: should this become a **core** prediction?

Concept	Meaning
Phase B — outcome	Trust-weighted society market: is the hypothesis **true**?
Side bet	Permissionless `market_decl` not in the core registry

While unresolved, a core prediction may be a working **hypothesis** or a **delusion**; finalization via log score is what separates them ($\mathcal{D}_{KL}$ in IPS vocabulary).

18.2 Genesis (`ChainParams` extension)

The following fields extend §2.2. They use `#[serde(default)]` and are **excluded from `chain_id`** (same rule as `perpetual_from_height` in `pot-core::state::ChainParams::chain_id`).

```
ChainParams {
  // ... existing fields ...
  max_core_predictions:          u32,   // default 3
  min_core_nom_stake_mtrust:     u64,   // default 1000
  core_adopt_bonus_base_mtrust:  u64,   // default 500
  core_adopt_threshold_ppm:      u32,   // default 700_000 (70% ADOPT)
  min_core_adopt_voters:         u32,   // default 3
  initial_core_predictions:      Vec<GenesisCorePrediction>,  // default []
}

GenesisCorePrediction {
  hypothesis:               String,
  rationale:                String,
  outcomes:                 Vec<String>,  // 2..=64 distinct labels
  resolver:                 PubKey,
  alpha_mtrust_per_nat:     u64,
  outcome_closes_at_height: u64,
}
```

At genesis initialization (before block 1):

1. For each entry in `initial_core_predictions`, validate shape.
2. If `count(active) < max_core_predictions`, spawn an **active** core prediction (skip phase A) with an outcome market opening at height `0` and closing at `outcome_closes_at_height`.
3. Record in `core_predictions` registry (§18.4).

18.3 Types

```
CorePredictionNomBody {
  chain_id:                 Hash,
  proposer:                 PubKey,
  hypothesis:               String,       // 1..=4096 bytes
  rationale:                String,       // 1..=8192 bytes
  outcomes:                 Vec<String>,  // phase-B labels; 2..=64 distinct
  resolver:                 PubKey,       // phase-B resolver
  alpha_mtrust_per_nat:     u64,
  adopt_opens_at_height:    u64,
  adopt_closes_at_height:   u64,
  outcome_opens_at_height:  u64,
  outcome_closes_at_height: u64,
  stake_mtrust:             u64,          // locked until adoption resolves
  author_adopt_probs_ppm:   Vec<u32>,     // [ADOPT, REJECT]; sums to PPM_TOTAL
  nonce:                    u64,
}
SignedCorePredictionNom { body: CorePredictionNomBody, sig: Signature }
```

```
CorePredictionAdoptVoteBody {
  nom_id:    Hash,   // = NomId (below)
  voter:     PubKey,
  height:    u64,
  probs_ppm: Vec<u32>,  // [ADOPT, REJECT]; sums to PPM_TOTAL
}
SignedCorePredictionAdoptVote { body: CorePredictionAdoptVoteBody, sig: Signature }
```

Identifiers and signature domains:

- `NomId := blake3(CBOR(("pot/core-prediction/nom/v1", CorePredictionNomBody)))`.
- Nom signature verified against `proposer` on `NomId` bytes.
- Adopt vote: `blake3(CBOR(("pot/core-prediction/adopt-vote/v1", chain_id, CorePredictionAdoptVoteBo`

Outcome constants for phase A: `ADOPT = 0`, `REJECT = 1`.

Reference types: `pot-core::core_prediction`.

18.4 State

```
CorePredictionNomState {
  nom:         SignedCorePredictionNom,
  adopt_votes: Map<PubKey, SignedCorePredictionAdoptVote>,
  status:      enum {
                 PendingAdoption,
                 Adopted { in_block_height: u64, market_id: MarketId },
                 Rejected { in_block_height: u64 },
               },
}

CorePredictionMeta {
  nom_id:            Hash,
  hypothesis:        String,
  rationale:         String,
  proposer:          PubKey,
  market_id:         MarketId,
  adopted_at_height: u64,
}
```

`ChainState` gains:

```
core_prediction_noms:  Map<NomId, CorePredictionNomState>,
core_predictions:      Map<MarketId, CorePredictionMeta>,   // active registry
stake_escrow:          Map<NomId, u64>,                     // locked stake_mtrust
```

Invariant: `|core_predictions|` MUST NOT exceed `params.max_core_predictions`.

18.5 Payload extension (v0.5)

`Payload` gains optional vectors (default `[]`):

```
core_prediction_noms:        Vec<SignedCorePredictionNom>,
core_prediction_adopt_votes: Vec<SignedCorePredictionAdoptVote>,
```

Git ref prefixes:

```
refs/pot/core-predictions/noms/<nom_id_hex>
refs/pot/core-predictions/adopt-votes/<nom_id_hex>/<voter_fp>
refs/pot/core-predictions/active/<market_id_hex>
```

Adopted hypotheses are also recorded under `refs/pot/core-predictions/active/` as CBOR `CorePredictionMeta` for hub and agent discovery.

18.6 Block application (v0.5 steps)

After §13 norm steps (or §4.3 markets if norms absent), **in order**:

18.6.1 Nominations For each `SignedCorePredictionNom cn` in `payload.core_prediction_noms`:

- Verify `cn.verify()`.
- Require `NomId` not already in `core_prediction_noms`.
- Require `|core_predictions| + pending_adoptions| < max_core_predictions` (count noms with `PendingAdoption` toward the cap).
- Require `cn.body.stake_mtrust >= params.min_core_nom_stake_mtrust`.
- Require `accounts[proposer].trust_mtrust >= stake_mtrust`.
- Lock stake: subtract from proposer, add to `stake_escrow[NomId]`.
- Insert `CorePredictionNomState::pending(cn)`.
- Treat `author_adopt_probs_ppm` as the proposer's first adopt vote at `B.header.height` (same rules as §18.6.2).

18.6.2 Adoption votes For each `SignedCorePredictionAdoptVote av` in `payload.core_prediction_adopt_votes`:

- Verify signature.
- Require nom exists and `status == PendingAdoption`.
- Require `adopt_opens_at_height <= av.body.height <= adopt_closes_at_height`.
- Require `B.header.height >= av.body.height`.
- Require no prior vote by `av.body.voter` on this nom.
- Insert into `adopt_votes`.

18.6.3 Adoption auto-resolution After 18.6.1–18.6.2, for each nom with `status == PendingAdoption` where `B.header.height > adopt_closes_at_height`:

Compute trust-weighted adoption market p_m over `{ADOPT, REJECT}` from `adopt_votes` using balances in `trust_snapshots[B.header.height - 1]`.

Let n = count of distinct voters excluding proposer.

ADOPT iff:

- $p_m(\text{ADOPT}) \cdot \text{PPM\_TOTAL} \geq \text{core\_adopt\_threshold\_ppm}$, **and**
- $n \geq \text{min\_core\_adopt\_voters}$, **and**
- $|core_predictions| < max_core_predictions$.

Otherwise **REJECT**.

On ADOPT:

1. Spawn `MarketState::open_from(SignedMarketDecl)` where the decl is derived:
   - `question := hypothesis`
   - `outcomes`, `resolver`, `alpha`, `opens_at_height := outcome_opens_at_height`, `closes_at_height := outcome_closes_at_height`
   - `proposer := nom.proposer`
2. Insert `CorePredictionMeta` into `core_predictions`.
3. Return `stake_mtrust` from escrow to proposer.
4. Credit `core_adopt_bonus_base_mtrust` to proposer (saturating).
5. KL-score adoption votes against outcome ADOPT (§5.5 / norm scoring).
6. Set nom status `Adopted { market_id, in_block_height }`.

On REJECT:

1. **Slash** escrowed stake (default: burn; MAY credit REJECT-side bettors proportionally to positive KL margin on REJECT — implementation choice).
2. KL-score adoption votes against outcome REJECT.
3. Set nom status `Rejected`.

Phase B (outcome market) follows §4.3 and §5.5 unchanged. The linked market MUST appear in `core_predictions`; hub MUST badge it as **Core**.

18.7 Economics (informal)

Event	Proposer	Society
Nom rejected	Loses `stake_mtrust`	Gains information; spam filtered
Nom adopted	Stake back + `core_adopt_bonus_base_mtrust`	Gains curated hypothesis registry
Outcome market	Scored like any society market	$\mathcal{D}_{KL}$ reduced on identity question

Proposer reward is for **successfully elevating** a hypothesis to core status, not for being right about the outcome. Outcome accuracy is scored in phase B like any other market.

18.8 CLI (planned)

```
pot core nominate --as ALICE \
  --hypothesis "People want to stream movies at home" \
  --rationale "Netflix-shaped product thesis" \
  --outcomes yes,no \
  --resolver RESOLVER_FP \
  --adopt-closes-at-height 100 \
  --outcome-closes-at-height 50000 \
  --stake 5000 \
  --p-adopt 0.65

pot core list              # pending noms + active core predictions
pot core bet-adopt --nom ID --probs 0.8,0.2
pot core bet --market ID --probs 0.7,0.3   # phase B (alias of market bet)
```

18.9 Hub and agents

- **Core predictions** section at top of Markets tab; distinct badge from generic society markets.
- `pot agent next` SHOULD prioritize open core prediction markets when the agent holds mT.
- Chain registration MAY include `core_hypotheses_summary` in hub manifest (off-chain metadata pointing at on-chain registry).

18.10 Implementation status

Component	Status
`pot-core::core_prediction` types	implemented
`Payload.core_prediction_*` fields	implemented (serde default)
`ChainParams` v0.5 fields	implemented (serde default)
`state.rs` application + escrow	**not implemented**
`pot-cli` / hub UI	**not implemented**
Genesis `initial_core_predictions` spawn	**not implemented**