TensorCash Core-Node: Operational Departures from Bitcoin

Imosuke Takakuni[1]

2 May 2026

ABSTRACT

TensorCash inherits Bitcoin Core's transaction graph, UTXO model, and signature machinery, but the node departs from Bitcoin in six operational ways spanning both consensus and peer-policy behavior: (1) the block header carries an additional commitment to an inference **proof blob** and an **adjusted-difficulty target**; (2) every block embeds a Chia-style **Wesolowski Verifiable Delay Function (VDF) proof** that ties the block to wall-clock work, which is verified at consensus from an activation height; (3) light clients and presync peers can request compact **sidecars** so they can validate the VDF and Merkle-prove the tick before downloading bodies; (4) per-block inference proof validation is split into **Quick**, **Smell**, and **Full** rungs, with Quick and Smell gating relay while Full runs outside the propagation path; (5) the presync layer applies policy gates that score chains by **proof-of-time before proof-of-work**, gate direct-fetch with a hysteresis margin, and add **ASN/onion diversity** and **body sampling** for deep reorgs; (6) when a deep reorg cannot be prevented, a **reorg advisory** computes a per-segment forensic report and can optionally **gate** the chain switch on an operator decision. We also describe the new **epoch-decay block subsidy**.

The proof blob is not a malleable side attachment: every new proof-payload item is cryptographically committed into `hashPoW`, and `hashPoW` is part of the canonical block identifier. TensorCash also expands the block envelope by 1 MB relative to Bitcoin so the proof payload has an explicit budget without stealing the familiar 4 MB transaction/witness budget. Implementation names quoted inline refer to protocol objects and public API surfaces.

I. Why Bitcoin Core needed to be modified

A vanilla Bitcoin Core node makes two implicit assumptions:

1. The *only* expensive thing a miner did was the SHA-256 grind. Everything else in the header is essentially free to fabricate.
2. The cumulative work `Σ 2^256 / target` is a faithful proxy for wall-clock effort, so chains can be ranked by chainwork alone.

TensorCash breaks both assumptions deliberately. Mining is an inference workload specified by the TensorCash Verification and Mining API papers: model forward passes drive entropy into the PoW search, and the heavyweight artefact is the **proof blob**. But a heavyweight blob alone is *not* sufficient for SPV / headers-presync: an adversary can grind cheap header-only forks (since the blob is not part of the short header digest), drown a syncing peer in shallow forks, and force it into the slow-path validation of every fake. The fix is to add a *second, orthogonal* proof dimension whose verification cost is small but whose production cost is real wall-clock time: a Wesolowski VDF.

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

The rest of this paper walks through the consequences.

i. API boundary

Three service surfaces sit around the consensus engine:

- **Core Node API.** A small authenticated REST surface exposes model registry metadata and miner metrics to the miner proxy, while the normal JSON-RPC server remains the chain-control surface.
- **Mining API.** The miner proxy fronts forked vLLM and llama.cpp engines, serves OpenAI-compatible inference routes, tracks the active model, captures proof-bearing token windows, and exposes proof retrieval by completion id. The mining engine produces the proof blob; it does not decide consensus.
- **Verification API.** The verifier accepts FlatBuffers validation requests over ZMQ or HTTP, exposes Quick, Quick+Smell, Full, Model, Challenge, and status routes, and returns the discrete proof states consumed by Core: `Quick_OK_Smell_OK`, `Quick_OK_Smell_Fail`, `Quick_Fail_Smell_Fail`, `Full_Green`, `Full_Amber`, and `Full_Red`.

Core treats that verifier as asynchronous infrastructure. It can run a non-ML Quick implementation in C++ locally, enqueue remote Quick+Smell or Full work through `ValidationAPI`, persist returned statuses, and resume block processing when a deferred Full result arrives. This is the key engineering split: Quick and Smell protect gossip; Full protects chain acceptance. Operationally, `ValidationAPI` owns separate request, response, and behavior threads: one dispatches queued verifier jobs, one records returned statuses, and one executes follow-up behaviors such as retrying `ProcessNewBlock` after a pending Full request resolves.

II. Block format and the two header digests

The TensorCash block header extends the Bitcoin header with three fields and ships a *new* serialization order:

```
class CBlockHeader {
public:
    int32_t  nVersion;
    uint256  hashPrevBlock;
    uint256  hashMerkleRoot;
    uint32_t nTime;
    uint32_t nBits;       // base PoW target (network difficulty)
    uint32_t nNonce;
    uint32_t nAdjBits;    // adjusted target = f(nBits, model_difficulty)
    uint256  hashPoW;     // commitment to the proof blob (Merkle root)
    uint8_t  flags;

    SERIALIZE_METHODS(CBlockHeader, obj) {
        READWRITE(obj.nVersion, obj.hashPrevBlock, obj.hashMerkleRoot,
                  obj.nTime, obj.nAdjBits, obj.nNonce, obj.nBits,
                  obj.hashPoW, obj.flags);
    }
    ...
};
```

Two distinct hashes are derived from this header:

```
// "Long" hash — full block identity, includes proof commitment
uint256 CBlockHeader::GetHash() const {
    DataStream ss;
    ss << nVersion << hashPrevBlock << hashMerkleRoot
       << nTime << nAdjBits << nNonce
       << nBits << hashPoW;
    return Hash(ss);
}

// "Short" hash — pre-VDF identity, excludes hashPoW
uint256 CBlockHeader::GetShortHash() const {
    DataStream ss;
    ss << nVersion << hashPrevBlock << hashMerkleRoot
       << nTime << nAdjBits << nNonce;
    return Hash(ss);
}
```

The intuition:

- `GetShortHash()` is the **pre-image the miner grinds against** during the inference search. It is what the external miner / `bitcoin-util` PoW check hashes when scanning for a nonce. It is *not* the next block's VDF challenge: a child's VDF input is the child's `hashPrevBlock`, i.e. the parent block's canonical `GetHash()`.
- `GetHash()` is the **canonical block identifier** committed to by children. It folds in `hashPoW`, so once the block is published the proof blob is cryptographically part of the chain — children would be invalidated if someone tried to swap the blob.

The full block carries the heavyweight payload alongside:

```
class CBlock : public CBlockHeader {
public:
    std::vector<CTransactionRef> vtx;
    CProofBlob pow;            // inference + VDF artefact
    uint64_t cumulative_tick;  // running sum of pow.tick, genesis-anchored
    ...
};
```

`cumulative_tick` is the chain's "wall-clock score" — the genesis-anchored sum of every block's VDF iteration count.

The block-size envelope is intentionally larger than Bitcoin's. TensorCash sets both the serialized block limit and block-weight limit to 5,000,000 bytes/weight units and reserves a 1,000,000-byte proof-blob budget. The additional 1 MB accounts for the PoW payload while preserving the practical 4 MB transaction/witness budget operators already expect from Bitcoin-style blocks.

III. The proof blob and its Merkle commitment

`CProofBlob` holds three classes of payload bundled into one serialized container:

Group	Purpose
	wall-clock & PoW artefact

Group	Purpose
`version`, `tick`, `timestamp`, `vdf`, `target`, `hash`, `block_hash`, `header_prefix`, `is_solution`, `extra_flags`	
`model_identifier` (`name@commit`), `compute_precision`, `ipfs_cid`	model identity, looked up in `g_modeldb`
`temperature`, `top_p`, `top_k`, `repetition_penalty`, `chosen_tokens`, `chosen_probs`, `sampling_u`, `softmax_normalizers`, `prompt_tokens`, `topk_logits/_indices`, `logsumexp_stats`, `pad_mask`	inference trace, replayed by the external validator

Putting

all

of that directly into the header digest would make headers tens of KB each — fatal for SPV. Instead, the header only commits to a Merkle root over four canonical leaves:

```
struct CPowLeaves {
    uint256 l_tick;   // tag 0x01: u64 LE tick count
    uint256 l_vdf;    // tag 0x02: serialized VDF proof bytes
    uint256 l_meta;   // tag 0x03: version || SHA256("name@commit")
    uint256 l_rest;   // tag 0x04: full blob serialization
};

// Each leaf is hashed under a domain-separated tag:
//   leaf = SHA256d( 0xFF || "POW\0" || tag || u32le(len) || data )
// The tree is balanced (4 leaves):
//   h01 = H(L_tick, L_vdf)
//   h23 = H(L_meta, L_rest)
//   root = H(h01, h23)
uint256 CProofBlob::GetMerkleRoot() const {
    CPowLeaves l = BuildLeaves();
    return ComputeMerkleRoot({l.l_tick, l.l_vdf, l.l_meta, l.l_rest});
}
```

The header invariant is enforced at consensus:

```
const bool expect_merkle = chainman.GetConsensus().IsVdfSpvActive(nHeight);
const uint256 expected = block.pow.GetCommitment(expect_merkle);
if (block.hashPoW != expected) {
    return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS,
                         "bad-pow-commitment", "pow blob commitment mismatch");
}
```

The leaf layout is engineered for *partial reveals*: a sidecar (§5) that exposes only `tick` and `vdf` ships two short authentication paths

```
// Branch for L_tick: { l_vdf, h23 }
// Branch for L_vdf : { l_tick, h23 }
```

(2 × 32 B = 64 B per branch), so an SPV peer can verify both leaves against `hashPoW` *without* ever seeing the multi-KB inference trace. The branches themselves are produced by `CProof-Blob::BuildBranchForTick()` and `BuildBranchForVdf()`.

This is the anti-malleability boundary. `L_tick`, `L_vdf`, and `L_meta` give small, separately provable commitments for SPV, while `L_rest` commits to the canonical serialization of the complete proof blob. Mutating any new proof payload item — version, tick, VDF, timestamp, target, proof hash, block hash, header prefix, model identifier, precision, CID, sampler parameters, chosen tokens, logits/probes, softmax normalizers, masks, or extra flags — changes `hashPoW`. Because `hashPoW` is included in `GetHash()`, the canonical block identifier changes too, and every descendant's `hashPrevBlock` commitment breaks. A proof blob therefore cannot be swapped, normalized, or rewritten after header propagation without invalidating the chain linkage.

IV. The Wesolowski VDF embedded in every block

The VDF integrates the open-source Chia VDF library. It is a standard Wesolowski proof over a class group of imaginary quadratic order:

```
bool VerifyAgainstPrevHash(const uint256& prev_hash,
                           std::span<const uint8_t> vdf_proof,
                           uint64_t iterations,
                           uint32_t discr_bits = 1024,
                           uint32_t recursion = 0)
{
    // 1. Challenge bytes are the child header's hashPrevBlock bytes:
    //    the parent block's canonical GetHash().
    std::vector<uint8_t> seed(prev_hash.begin(), prev_hash.end());

    // 2. Derive a fresh discriminant D of |D| = 1024 bits from the seed.
    integer D = CreateDiscriminant(seed, discr_bits);

    // 3. Initial form x = (a=2, b=1, c=(1-D)/8), reduced.
    form x = form::from_abd(integer(2), integer(1), D);
    x.reduce();

    // 4. Wesolowski check:    y = x^(2^T) ; verifier checks π = x^q · r
    return CheckProofOfTimeNWesolowski(D,
        x_bytes.data(), vdf_proof.data(), int32_t(vdf_proof.size()),
        iterations, real_bits, int32_t(recursion));
}
```

The math, written compactly:

> **VDF tuple** `(D, x, T) → (y, π)`
>
> - **Setup** `D ← H(prev_hash, |D|)` (1024-bit discriminant), `x = (2,1,D)`.
> - **Eval** `y = x^(2^T)` in the class group `Cl(D)`. This is sequential — no shortcut beyond the generic `T` squarings.
> - **Wesolowski proof** Verifier samples a 264-bit prime `ℓ`, prover returns `π = x^q` where `2^T = qℓ + r` (`r < ℓ`). Verifier accepts iff `π^ℓ · x^r = y`.

> • **Cost asymmetry** Eval costs `T` group squarings (~10^5–10^7 today on a single core), but verification costs only `O(log ℓ)` and finishes in a few milliseconds.

Three concrete bindings are critical:

1. **Challenge =** `prev_hash`**.** Here `prev_hash` is the child header's `hashPrevBlock`, i.e. the parent block's canonical `GetHash()`. A miner cannot pre-compute the VDF for block `n+1` until block `n` is final. Grinding the short hash to produce a shallow alternative does not help: the *child* would still need a fresh VDF for the new prev-hash.
2. `iterations = block.pow.tick`**.** The header-committed tick count *is* the VDF time parameter. A miner who lies about ticks fails verification.
3. `hashPoW` **commits to** `vdf` **via** `L_vdf`**.** Tampering with the blob after the fact breaks the Merkle commitment.

Consensus turns the VDF check on at `vdf_spv_vdfverify_height` and enforces it in contextual block validation:

```
if (chainman.GetConsensus().IsVdfVdfVerifyActive(nHeight)) {
    if (!vdf::VerifyAgainstPrevHash(block.hashPrevBlock,
                                    block.pow.vdf,
                                    block.pow.tick,
                                    /*discr_bits=*/1024, /*rec=*/0)) {
        return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS,
                             "bad-vdf-proof", "vdf proof verification failed");
    }
}
```

A second consensus rule keeps `cumulative_tick` honest:

```
const uint64_t expected = prev_block.cumulative_tick + block.pow.tick;
if (block.cumulative_tick != expected)
    return state.Invalid(..., "bad-cumulative-tick", ...);
```

Together these three rules — Merkle commitment, VDF verification, and cumulative-tick continuity — turn the VDF dimension into an honest, chain-anchored measure of wall-clock effort. In normal connected validation the parent block is available and the continuity check is enforced; if the implementation cannot read the previous block from disk, it logs and skips this specific check rather than inventing a value.

V. The sidecar protocol — `getheadext` / `headers_ext`

A new service flag `NODE_VDFSPV` advertises that a peer will answer two new p2p verbs:

- `GETHEADERS_EXT`: vector of `VdfExtQueryItem { header_hash, prev_hash }`
- `HEADERS_EXT`: vector of `VdfExtSidecar` (max 1 KiB per element).

The sidecar wire format:

```
struct VdfExtSidecar {
    uint256 header_hash;
    uint256 prev_hash;
    uint64_t tick;
    std::vector<unsigned char> vdf;       // ~200 B Wesolowski proof
    std::vector<uint256> merkle_branch_tick; // sibling list for L_tick
    std::vector<uint256> merkle_branch_vdf;  // sibling list for L_vdf
    uint8_t  leaf_scheme_version{1};
    uint32_t n_leaves{4};
};
```

A

valid

sidecar therefore lets a peer answer three questions cheaply:

1. **Are these the leaves the header committed to?** Recompute `L_tick` and `L_vdf` from the sidecar fields, walk both Merkle paths up to the root, compare with `block.hashPoW`.
2. **Does the VDF actually verify against the parent's hash with the advertised tick count?** Run `vdf::VerifyAgainstPrevHash`.
3. **What is the chain's accumulated wall-clock score so far?** Add `tick` to the parent's cumulative tick.

All three together cost ≲10 ms of CPU; the malicious-grind cost (forging a header that *also* has a fresh, valid VDF over its prev-hash) is on the order of `T` sequential class-group squarings — minutes of single-thread time per attempt, with no parallel shortcut.

i. Resource caps

Every parameter is intentionally tight so a misbehaving peer cannot move the node off its budget:

Parameter	Value	Comment
`VDF_MAX_PROOF_SIZE`	1 024 B	a sidecar that exceeds this is dropped at receipt
`VDF_MAX_SIDECAR_BYTES`	1 024 B	server-side budget per response element
`VDF_MAX_BUCKETS_PER_PARENT`	16	max distinct proofs per `prev_hash`
`VDF_MAX_BUCKETS_TOTAL`	50 000	global eviction cap, oldest-first
`VDF_BUCKET_TTL_SEC`	60	unverified buckets expire fast
`VDF_HEADERS_MAX_ENTRIES`	60 000	header-info LRU
`VDF_HEADER_TTL_SEC`	1 800	non-pinned headers expire after 30 min
`VDF_VERIFY_CACHE_MAX`	64 000	dedup verified `(prev_hash, tick, H(vdf))`
`VDF_VERIFY_CACHE_TTL_SEC`	600	10 min
`VDF_MAX_VERIFIES_PER_SEC`	200	global VDF worker throttle
`MAX_HEADERS_EXT_PER_MIN`	2 000	per-peer rate cap on incoming `HEADERS_EXT`
Active-chain headers	pinned	never evicted, even after long sleeps

Verification runs on a single dedicated worker thread that batches up to 8 tasks at a time, dedups against the cache, and only accepts the proof in the bucket on success. Malformed, oversized, stale, or rate-limited sidecars can cost the sender a `Misbehaving` increment with a specific reason string (`headers_ext-rate-limit`, etc.). A VDF proof that is well-formed but simply fails verification is not accepted into the bucket; the worker drops that verification result and continues.

VI. Propagation and verification gates

Bitcoin's headers-first sync ranks competing peers by `nChainWork` and only falls back to body validation after picking a tip. TensorCash inserts two earlier gates:

1. A **relay gate** for newly received blocks. The node runs Quick+Smell before announcing the header or compact block to peers.
2. A **presync gate** for candidate chains. The node verifies sidecars and scores cumulative VDF ticks before fetching full bodies.

Full verification is deliberately not a relay prerequisite. Let `N` be the reachable node count, `f` the effective fanout, `R = Θ(log_f N)` the number of epidemic gossip rounds, `δ` the per-round network latency, `T_qs` the Quick+Smell time, and `T_full` the Full time. Randomized block gossip reaches the network in roughly:

```
relay_with_quick_smell  ≈  R · (δ + T_qs)
relay_after_full        ≈  R · (δ + T_full)
```

Because `T_full` is measured in seconds and `R` grows logarithmically with the network, putting Full on every propagation hop would multiply a seconds-scale model replay by the epidemic diameter. TensorCash therefore broadcasts the header / compact-block announcement after Quick+Smell passes, while Full runs asynchronously and gates local block acceptance and chain-tip update. Economically validating nodes can still require `Full_Green` before accepting the block; they just do not make every relay hop wait for that result.

The full verification ladder for an incoming chain is:

```
                    ┌─ rung 0 ── headers received (CheckHeadersPoW, anti-DoS, min chainwork)
                    |
                    ├─ rung 1 ── sidecar requested over GETHEADERS_EXT
                    |             • Merkle branch into hashPoW    (≪1 ms)
                    |             • Wesolowski VDF verify          (~few ms)
                    |             → header bucket marked VALID, cum_tick set
                    |
                    ├─ rung 2 ── SPV scoring + hysteresis gate
                    |             • Compute cum_tick(candidate) from genesis
                    |             • Require margin over current best (§6.1)
                    |
                    ├─ rung 3 ── block body received
                    |             • Check proof-blob commitment and size envelope
                    |             • Run local C++ Quick or Verification API Quick+Smell
                    |             • If Quick+Smell passes, announce header / compact block
                    |
                    ├─ rung 4 ── ASN / onion diversity check  (only if D > 3)
                    |             • ≥ 2 distinct ASNs / vetted onion vanity peers
                    |
                    ├─ rung 5 ── deep-reorg body sampling      (only if D > 6)
                    |             • Fetch M = ⌈N/3⌉ of N = min(12, 2D) bodies
                    |
                    └─ rung 6 ── full ConnectBlock validation (UTXO, scripts,
                                  hashPoW commitment, VDF, cumulative_tick,
                                  external validator Full)
```

Each rung is more expensive than the previous one and is only attempted if the prior rung passes. A peer that produced a fork without a real VDF is ejected at rung 1; one that pretends to have a longer chain via cheap sub-tree work is rejected at rung 2; one that ships an impossible proof blob is stopped before relay at rung 3.

The DoS policy follows the same split. Malformed headers, malformed sidecars, oversized sidecars, bad Merkle branches, bad commitments, and other cheap deterministic failures can increment peer misbehavior immediately. A `Full_Red` result is handled differently: it rejects local acceptance and can cache the block as failed, but it is not automatically treated as proof that the announcing peer was malicious, because the network intentionally permits relay before Full completes. This avoids turning slow model verification into a remote peer-banning oracle.

i. SPV hysteresis

Once sidecars accumulate, every header has a verified `tick` and the node can compute the wall-clock score along any branch:

```
// cum_tick(tip) = Σ tick(h)  for h on path tip → genesis,
// provided every header on the path has a VALID sidecar.
std::pair<bool, arith_uint256>
ComputeCumFromGenesis(const CBlockIndex* tip, int64_t now);
```

The **hysteresis gate** for direct-fetching a new tip is then:

```
D := depth_current  =  best_tip.height − LCA.height
E := EMA( tick per block ) on the active chain     // initialized to 1e6, α=0.02
margin = ⌊ E · base_frac ⌋ + D · E                 // base_frac = 0.5

accept_candidate ⇐⇒ cum_tick(candidate) ≥ cum_tick(best) + margin
```

In words: to reorg `D` blocks deep, the candidate must beat us by **half a block-worth of wall-clock evidence plus one block-worth per block of depth**. If sidecar coverage is incomplete and the score is unavailable, we fall back to chainwork — but never *replace* it with the cheaper short-hash work.

ii. ASN / onion diversity for deep reorgs

A single attacker (or a single hosting region) can run dozens of nodes. To prevent that from triggering a deep reorg, every tip announcement is attributed to either the peer's mapped autonomous system number, or — for onion peers — to the peer's onion vanity tag. Reorgs of depth `D > 3` are only allowed once **at least two distinct ASNs or vetted onion-vanity peers** corroborate the tip. Inbound onion peers receive *no* diversity credit — diversity must come from outbound, freshness-vetted vanity onions or clearnet ASNs.

iii. Deep-reorg body sampling

For `D > m_spv_reorg_sampling_threshold` (default 6) the node refuses to bulk-fetch the candidate's bodies until a representative sample passes:

```
K       = min(144, D)                    // window size
N       = min(12, 2·D)                   // # samples
M       = ⌈N / 3⌉                        // success quorum
targets = roughly evenly spaced K-window block hashes
```

Bodies are fetched piecewise; bulk fetch only proceeds when at least `M` of `N` sampled bodies arrive and validate. This converts a "sync-attack" into a "supply-`M`-real-blocks" attack — the latter being indistinguishable from a real fork.

VII. The reorg advisory

Even with §4–§6 in place, a sufficiently endowed adversary or a true network split can produce a deep reorg whose VDF and chainwork all check out. At that point the question is not *"is this allowed?"* but *"is this plausible, and does the operator want it?"*. That is what the reorg advisory answers.

i. What it computes

The reorg advisory subsystem exposes `GenerateReorgAdvisory(current_tip, fork_tip, lca)`. The output structure is:

```
struct ReorgAdvisory {
    int    lca_height;
    int    depth_current;     // current_tip.height − lca.height
    int    depth_fork;        // fork_tip.height − lca.height
    double tx_overlap_pct;    // Jaccard(seg_current.txids, seg_fork.txids) × 100
    int64_t first_block_delay_secs;
    double hashrate_current_pct;
    double hashrate_fork_pct;
    TickTimeCalibration calibration;  // sec_per_tick, baseline_hashrate
    SegmentStats seg_current, seg_fork;
    int64_t since_last_block_secs;
    bool is_valid;
    std::string Summary() const;
};
```

The two interesting derivations:

(a) Tick-to-time calibration. For the 2 000 blocks ending at LCA we compute

```
sec_per_tick = (nTime[LCA] − nTime[LCA-W]) / (cumulative_tick[LCA] − cumulative_tick[LCA-W])
baseline_hashrate = (chainwork[LCA] − chainwork[LCA-W]) / (nTime[LCA] − nTime[LCA-W])
```

This converts the chain's wall-clock score back into seconds, calibrated on

pre-fork

blocks both branches share. It is the backbone of every other metric.

(b) First-block delay. Two estimates, take the max:

```
Δ₁ = first_seen(first_fork_block) − miner_nTime(first_fork_block)
Δ₂ = first_seen(first_fork_block) − ( nTime(LCA) +
                                     ticks_since_LCA · sec_per_tick )
delay = max(Δ₁, Δ₂)
```

A long delay says the fork sat undisclosed before being released — the classic withhold-and-replace attack signature. The VDF is what makes `Δ₂` trustworthy: without it, an attacker can lie about miner timestamps and escape detection.

(c) Hashrate% per segment. Using `min(clock_time, tick_time)` for the current segment and `clock_time` (or miner-time fallback) for the fork segment, both divided by `baseline_hashrate`. Anomalously low fork hashrate means the fork was produced by a small subset of the network; anomalously high means rented capacity.

(d) Tx overlap (Jaccard). Sets of non-coinbase txids for both segments (capped at 100 blocks per segment to bound disk I/O); a high overlap value indicates an honest temporary fork (same mempool, just different ordering), a low value indicates a hostile rewrite (different mempool, e.g. censored or double-spent transactions).

ii. When it triggers and what it does

`ShouldTriggerAdvisory` requires:

- `depth_current > 3` (default `ADVISORY_DEPTH_THRESHOLD`),

- `since_last_block ≤ 6 h` (default `ADVISORY_OFFLINE_THRESHOLD_SECS`),
- and the system to be enabled (`-reorgadvisory=1`, default on).

The 6-hour clamp is critical: a node that has been offline for a day and boots into a deep reorg should *not* spam advisories — that is normal IBD, not adversarial activity.

Two operational modes:

1. **Advisory-only (default).** `AsyncLogReorgAdvisory(...)` computes the advisory synchronously at the reorg decision point, then queues the logging, notification, and store update on a single-thread worker pool (`AdvisoryWorkerPool`, max 100 entries). The operator gets a structured log line and the advisory is retained in `GetReorgAdvisoryStore()` (last 100, RPC-readable). The chain switch proceeds normally.
2. **Gating mode** (`-reorgadvisorygating=1`, off by default). The reorg is *blocked*: `state.Invalid(BLOCK_REORG_GATING, ...)` returns up to `ActivateBestChain`, which releases `cs_main` and waits on `ReorgGatingManager::WaitForDecision()` for an operator decision via the `submitreorgdecision` RPC. Defaults: 30 min timeout, default action on timeout = REJECT (stay on the current chain).

Optional shell hook: `-reorgadvisorynotify="<cmd>"` substitutes `%d` (depth_current), `%f` (depth_fork), `%h`, and `%o` (overlap%). In gating mode `%h` is the fork-tip hash and the command is launched from the reorg decision path; in advisory-only mode the worker path currently uses the LCA height for `%h` before calling `runCommand`. Typical use is to page the operator (PagerDuty, Slack, e-mail).

iii. Why an advisory and not auto-acceptance

A few principled reasons:

- **A deep reorg in Bitcoin is, today, a 1-in-many-years event;** in a young inference-PoW chain it is comparatively cheaper for a state-actor or a large-fleet adversary, especially during low-hashrate dawn periods.
- **The economic damage is operational, not cryptographic.** Once you have a >6-block reorg, exchanges have already credited deposits, custodians have already settled — silently switching the chain causes more financial harm than refusing it.
- **The VDF + ASN + sampling stack already makes naive deep reorgs hard.** If one *does* succeed, it is by definition exceptional and worth a human in the loop.
- **Keeping it off by default** preserves Nakamoto consensus for ordinary operators, while letting institutional nodes flip on gating without forking the protocol.

VIII. Block subsidy: the new issuance schedule

TensorCash replaces Bitcoin's pure halving with an **epoch-decay** schedule when `consensus.tensor_subsidy = true`. The flag is set on TensorMain, TensorTest, and TensorRegtest. The full rule is:

```
static constexpr CAmount kInitialReward{715 * COIN};     // 715 TC at genesis
static constexpr int     kEpochStartLen{715};            // first epoch length
static constexpr int     kEpochCapLen{715 * (1 << 10)};  // 732 160 blocks
static constexpr int     kP{3}, kQ{5};                   // reward × 3/5 per epoch

if (nHeight == 0) return kInitialReward;

CAmount reward    = kInitialReward;
int     epoch_len = kEpochStartLen;
int64_t remaining = nHeight;
while (remaining >= epoch_len && reward > 0) {
    remaining -= epoch_len;
    reward     = (reward * kP) / kQ;            // integer ×3/5
    epoch_len  = std::min(epoch_len * 2, kEpochCapLen);
}
return reward;
```

Mathematically:

$$\begin{aligned} \text{reward(epoch } k) &= \lfloor\, 715 \cdot \text{COIN} \cdot (3/5)^k \,\rfloor \\ \text{epoch\_len}(k) &= \min(\, 715 \cdot 2^k,\ 715 \cdot 2^{10} \,) \end{aligned}$$

So the first epochs are short (715 blocks ≈ 5 days at 10-min target spacing), the curve decelerates as epochs lengthen, and integer truncation eventually drives the reward to 0. The 3/5 ratio (vs. Bitcoin's 1/2) gives a smoother emission and front-loads issuance when the network most needs to bootstrap hashrate / inference capacity, while the doubling epoch length stretches the tail decades into the future. Total supply is bounded but soft — implementation-defined by the truncation and the cap, not by an explicit constant.

A second consensus rule preserves Bitcoin's familiar relationship between header difficulty and the block reward. The header carries both `nBits` (network target) and `nAdjBits` (adjusted target, a function of the mined-against model's catalogued difficulty). Consensus enforces

$$\text{adj\_target} \;\le\; \lfloor\, \text{base\_target} \cdot \text{ModelDifficultyNormalizer} \,/\, \text{model.difficulty} \,\rfloor$$

(saturated against powLimit, computed in 256-bit then 128-bit arithmetic to avoid overflow)

so that easier models don't get cheaper PoW: the normalizer is fixed in consensus params, the difficulty is the model registry's catalogued value stored by Core, and the resulting `adj_target` is the actual target the miner's hash must beat.

IX. End-to-end summary

Putting all the pieces together, a TensorCash full node treats an incoming chain as follows:

1. **Header arrives.** Cheap PoW sanity check on `nBits`. Anti-DoS minimum-chain-work gate. If the peer advertises `NODE_VDFSPV`, the node queues a `GETHEADERS_EXT` for the new headers.
2. **Sidecar arrives.** Validate Merkle branches into `hashPoW`, run Wesolowski VDF over `(prev_hash, tick)`, dedupe in the verify cache, mark the bucket VALID, increment `cum_tick`.
3. **Score the candidate.** Compare cumulative tick to current best with the `base_frac · E + D · E` hysteresis margin. Fall back to chainwork only if sidecar coverage is incomplete.

4. **Block body arrives.** Check the 5 MB block envelope, the 1 MB proof-blob cap, and the `hashPoW` commitment. Run Quick+Smell through local C++ Quick or the Verification API. If Quick+Smell passes, relay the header / compact-block announcement immediately; enqueue Full asynchronously.
5. **Diversity gate.** If `D > 3`, require ≥ 2 distinct ASNs / vetted onion announcers for the candidate tip.
6. **Body sampling.** If `D > 6`, fetch and validate `M = ⌈N/3⌉` of `N = min(12, 2D)` evenly-spaced bodies before bulk fetch.
7. **Full validation.** `ConnectBlock` checks transactions, witness commitment, the `hashPoW = MerkleRoot(L_tick, L_vdf, L_meta, L_rest)` commitment, the Wesolowski VDF, the `cumulative_tick = prev + tick` continuity, and the `nAdjBits ≤ ⌊base · norm / difficulty⌋` ratio. Subsidy is computed by the epoch-decay schedule (§8). Model-registration transactions are dispatched to the external validator through `ValidationAPI`.
8. **If a deep reorg is happening anyway**, a `ReorgAdvisory` is computed and either logged (default) or used to *block the switch* until the operator decides (`-reorgadvisorygating=1`). The advisory carries a tick-calibrated first-block delay, per-segment hashrate%, and Jaccard tx overlap — i.e. the forensic primitives needed to tell *legitimate network split* from *adversarial rewrite*.

Together, these changes preserve the property a Bitcoin operator cares about most — **chain finality grows as more wall-clock effort accumulates** — while replacing pure SHA-256 grinding with two orthogonal, useful proofs: inference work (which produces commercial value) and Chia-style VDF time (which is what an SPV light client can actually verify on a phone).

Appendix A. Notable implementation surfaces

Concern	Surface
Block header & block primitives	`CBlockHeader`, `CBlock`, `GetHash()`, `GetShortHash()`
Proof blob, Merkle leaves & branches	`CProofBlob`, `BuildLeaves()`, `BuildBranchForTick()`, `BuildBranchForVdf()`
VDF generation & verification	Chia VDF glue, `VerifyAgainstPrevHash()`
Sidecar wire format, buckets, worker, hysteresis, ASN, sampling	`NODE_VDFSPV`, `GETHEADERS_EXT`, `HEADERS_EXT`, `VdfExtSidecar`, VDF worker, tip-announcer policy
Mining API	OpenAI-compatible inference proxy, active-model selection, proof cache, `/v1/proof/{completion_id}`
Verification API	ZMQ and HTTP verifier bridge, `/v1/verify/quick`, `/v1/verify/quick-smell`, `/v1/verify/full/request`, `/v1/verify/status/*`, `/v1/public/status/{hash_id}`
Core Node API	`/api/v1/models`, `/api/v1/models/{model_hash}`, `/api/v1/miner/metrics`
Consensus enforcement	`hashPoW`, VDF, cumulative tick, adjusted difficulty, subsidy, and block-size limits
Reorg advisory	advisory generation, advisory store, optional operator gating
Service flag negotiation	`NODE_VDFSPV` advertisement

Appendix B. Default knobs

Knob	Default	Scope
Headers-presync-side service flag	`NODE_VDFSPV` advertised	service negotiation
`GETHEADERS_EXT` per-peer rate	2 000 / minute	sidecar DoS budget
VDF verifications	200 / second	VDF worker budget
VDF verify cache	64 000 entries / 10 min	VDF dedupe
Sidecar wire size	1 KiB	sidecar DoS budget
Block serialized size	5,000,000 bytes	consensus block envelope
Block weight	5,000,000 weight units	consensus block envelope
PoW blob size	1,000,000 bytes	consensus proof payload envelope
Hysteresis base fraction	$0.5 \cdot E$	SPV direct-fetch policy
Tick EMA α	0.02	SPV direct-fetch policy
Default E	10^6 ticks/block	SPV direct-fetch policy
ASN corroboration	enabled, 2 distinct sources	deep-reorg policy
Deep-reorg sampling threshold	$D > 6$	deep-reorg policy
Body samples	$N = \min(12, 2D)$, $M = \lceil N/3 \rceil$	deep-reorg policy
Reorg advisory depth threshold	$D > 3$	advisory trigger
Reorg advisory offline clamp	6 h	advisory trigger
Reorg gating timeout	30 min, default REJECT	advisory gating
Subsidy: initial reward	$715 \cdot \mathrm{COIN}$	epoch-decay subsidy
Subsidy: epoch start length	715 blocks	epoch-decay subsidy
Subsidy: per-epoch decay	× 3 / 5	epoch-decay subsidy
Subsidy: epoch-length cap	$715 \cdot 2^{10}$ blocks	epoch-decay subsidy