Cowboy Storage: CBFS and Runner Attached Storage

Technical Whitepaper

Charles DePue, Martin Aceto

November 5, 2026 — v0.10.9

Contents

1 Abstract **3**

2 Introduction **3**

3 Architectural overview **4**
3.1 Terminology . . . 4
3.2 Design rationale . . . 5
3.3 Differences vs. on-chain state . . . 5

4 Volumes and the account model **6**

5 The CBFS data plane **7**
5.1 Object representation . . . 8
5.2 Encryption . . . 10
5.3 Manifest . . . 10
5.4 Placement . . . 11
5.5 Erasure coding . . . 13
5.6 Transport . . . 13

6 Writing and reading data **14**
6.1 Write (object transfer) . . . 14
6.2 Commit and publication . . . 15
6.3 Read (object transfer) . . . 16
6.4 FUSE mount . . . 17

7 Consistency **18**

8 Relay Nodes **19**
8.1 Registration and stake . . . 19
8.2 Repair . . . 20
8.2.1 Repair local shards . . . 20
8.2.2 Replace unavailable relays . . . 20
8.3 Drain and peer-push . . . 21

9 Proof of retrievability **21**
9.1 Challenge protocol . . . 21
9.2 Outcomes . . . 23

© Cowboy Labs, Inc. 2026

10 The CapToken authorization model **23**
10.1 Runner CapTokens (chain-issued) . . . 24
10.2 Owner CapTokens (client-signed) . . . 25
10.3 DelegationCert . . . 26
10.4 Direct-client grants . . . 26
10.5 Access modes . . . 27
10.6 Scoping and concurrency . . . 28

11 Volume lifecycle **29**
11.1 Create (control plane) . . . 29
11.2 Attach (control plane) . . . 29
11.3 Delete, grace and restore . . . 30

12 Billing, rent, and eviction **31**
12.1 Fee components . . . 31
12.2 Relay earnings . . . 32
12.3 Settlement . . . 33
12.4 Comparison with state rent . . . 35

13 On-chain control plane **35**
13.1 Storage Manager (`0x0A`) . . . 36
13.2 Relay Registry (`0x0B`) . . . 38
13.3 Cache freshness . . . 39

14 State transition function (storage subset) **39**

15 Security considerations **40**
15.1 Client-side encryption . . . 40
15.2 Shard ID opacity . . . 40
15.3 Durability and scoped writes . . . 40
15.4 Relay eclipse and withholding . . . 40
15.5 Sybil resistance . . . 40
15.6 Key compromise . . . 41
15.7 Key-delivery boundary . . . 41
15.8 Replay and nonce reuse . . . 41

16 Parameters (genesis defaults) **41**

17 Relationship to the Technical Whitepaper **44**

© Cowboy Labs, Inc. 2026

1 Abstract

This document specifies Cowboy's storage layer: the **Cowboy File System (CBFS)** and **Runner Attached Storage (RAS)**. CBFS is a decentralized, encrypted, erasure-coded storage system run by a permissionless network of **Relay Nodes**. RAS is the Cowboy protocol layer on top of CBFS. It groups data into account-scoped **Volumes**, records each volume's state on chain in a compact **StorageCommitment**, and controls access to volumes with cryptographic **capability tokens** (CapTokens).

CBFS moves object data off chain, so an object read or write needs no transaction. RAS keeps volume ownership, manifest publication, billing, challenges and eviction on chain. The client encrypts a private volume's contents before sending them to Relay Nodes. A **delegation keypair** keeps the wallet key, which can be held offline, apart from a separate storage key that the client holds online for everyday use. The owner's client signs data operations locally with the storage key, and Relay Nodes check them against cached chain state without a chain round trip.

This specification covers the data layout, the write and read paths, consistency, Relay Node registration and repair, Proof-of-Retrievability challenges, CapToken and delegation cryptography, the volume lifecycle, per-epoch storage rent, and storage parameters. It complements the **Cowboy Technical Whitepaper**, which is the normative reference for consensus, execution, economics and actor semantics. The Relationship to the Technical Whitepaper section lists which document governs what.

2 Introduction

Actors on Cowboy are autonomous Python programs. Their on-chain storage pays state rent on the bytes it holds and has protocol-wide limits of 8 MiB and 10,000 keys per actor (CIP-4 §12). Those limits fit small state. Autonomous agents also need to keep working memory, cache retrieval traces, hold model adapters, stage artifacts between jobs and serve large public assets. That data is too large, changes too often, or is too private to put in a consensus-critical key/value store.

Cowboy therefore keeps two kinds of data apart:

- **State** is small, consensus-critical key/value data that takes part directly in the state transition function. It is billed per cell, stored in QMDB, and subject to rent and to the per-actor 8 MiB / 10,000-key limits for non-reserved actors (CIP-4 §12).
- **Storage** is large, possibly private data. A permissionless Relay Node network keeps it durably, and the chain refers to it only through compact commitments. It is billed per

© Cowboy Labs, Inc. 2026

byte per epoch in a separate market, and capability tokens control access to it without a transaction per access.

CBFS stores the data. It runs on its own as a plain distributed filesystem and also as Cowboy's data plane, the part of the system that moves and stores the bytes. It uses the same binaries and wire protocol in both configurations. RAS is the control plane, the on-chain records and rules that decide who owns, may access and pays for that data. It defines how Cowboy accounts own volumes, grant runners access, pay storage rent, and register and slash Relay Nodes.

3 Architectural overview

This document uses **MUST/SHOULD/MAY** as defined in RFC 2119. Parameters marked **governance-tunable** can be changed through the on-chain governance mechanism in Technical Whitepaper §11. The Technical Whitepaper is the normative reference for consensus, execution and economic parameters outside the storage layer.

The detailed storage contract is CIP-9. Storage pricing and settlement follow CIP-31. The authorization rules below summarize CIP-9, including its direct-client grant contract.

3.1 Terminology

The data model, from largest to smallest:

- A *volume* is a named, account-scoped storage namespace. Each volume is private (encrypted) or public (plaintext) and holds objects identified by path keys.
- An *object* is a single blob within a volume, identified by its path. The client encrypts it for a private volume and stores it as plaintext for a public one.
- A *chunk* is a bounded piece of an object. The client splits every object into chunks (one chunk even for an empty object) and encodes each chunk separately. Chunk size depends on the erasure width; see Volumes and the Account Model.
- A *shard* is one erasure-coded fragment of an object chunk or manifest node. Any K of its K+M shards reconstruct that chunk or node.
- A *stripe* is all `w = K+M` shards of one object chunk or manifest node (leaf or interior). They share one `shard_id` and contain indexes 0 through `w−1`. The chain keeps a list of every volume's live stripes in its own state, the *native RAS inventory*, with one row per stripe.
- The *manifest* is the authoritative, Merkle-committed index of a volume's contents. It is encrypted for private volumes and plaintext for public ones, and Relay Nodes store it as content-addressed DAG nodes.

- The *manifest root* is the locator (BLAKE3 hash) of the manifest DAG's top node. The chain stores it as `StorageCommitment.manifest_root`, and every read starts from it.
- A *StorageCommitment* is the on-chain record of a volume's existence, owner, manifest root, erasure parameters, size, billing state and status.
- A *PlacementRecord* is the off-chain, mutable record that binds a shard ID to its assigned Relay Nodes. It is replicated across the assigned nodes and versioned with compare-and-swap.

The participants and credentials:

- A *Relay Node* is a network participant that stores erasure-coded shards of volume data, answers reads, takes part in repair and answers Proof-of-Retrievability challenges. Relay Nodes are a separate role from Validators and Runners.
- A *CapToken* is a bearer credential that authorizes volume operations. The chain issues runner tokens at job dispatch. A delegated client key signs owner-auth tokens, and a relay admits them through volume ownership or an explicit direct-client grant. Scope and validation depend on the authorization path.
- A *DelegationCert* is a wallet-signed certificate that binds an Ed25519 CBFS key to a Cowboy account for storage use only, with an expiry of at most 30 days.
- The *Storage Manager* (`0x0A`) is the Cowboy system actor that maintains StorageCommitments, mints runner CapTokens, runs per-epoch billing and governs volume status transitions.
- The *Relay Registry* (`0x0B`) is the Cowboy system actor that maintains Relay Node profiles, staking, health, and repair and challenge coordination.

3.2 Design rationale

Actor workloads need mutable working state, private data, shared artifacts and filesystem access from short-lived runners. CBFS meets these needs as follows:

- Clients exchange shards directly with Relay Nodes. Publication and lifecycle changes go through the chain.
- A job attachment binds storage access to one runner. Owners and explicitly authorized clients use locally signed tokens.
- An on-chain root identifies the authoritative volume contents that relays serve.
- Erasure shards and placement metadata let relays repair private data without holding the DEK.
- FUSE lets runner workloads use ordinary file operations on persistent volumes.

3.3 Differences vs. on-chain state

© Cowboy Labs, Inc. 2026

Aspect	On-chain State	CBFS / RAS
Typical size	Bytes to kilobytes	Kilobytes to hundreds of GiB
Privacy	Public to all validators	Encrypted per volume (private mode)
Fees	Cycles, Accesses, Cells, and state rent	Storage rent and transfer fees; chain mutations also pay execution fees
Durability	Consensus replication across all validators	Reed-Solomon: at least K of K+M shards per stripe at commit, with owed shards repaired
Access control	Actor and account authorization	Runner tokens, ownership, and direct-client grants
Data path	State mutations execute on-chain; read-only RPCs query committed state	Object transfers use relay RPCs; manifest publication uses a transaction
Latency	Mutations wait for consensus; read-only queries use RPC	Transfers depend on relay round trips; publication waits for consensus

4 Volumes and the account model

Each volume is identified by:

```
volume_id = keccak256(owner_address || utf8_bytes(volume_name))
```

`volume_name` MUST be UTF-8, NFC-normalized, 1–64 bytes long, and match `^[a-zA-Z0-9_\-.]+$`. It MUST NOT begin or end with `.` or `-`. Each volume has a **visibility**, set at creation and never changed:

- In a `PRIVATE` volume, objects are AES-256-GCM encrypted under a per-volume DEK. The manifest is encrypted. Reads and writes require CapTokens.
- In a `PUBLIC` volume, objects are plaintext. The manifest is plaintext. Any party may read without a CapToken; writes still require one.

An account MUST NOT own more than `MAX_VOLUMES_PER_OWNER = 1,024` volumes. Each volume MUST NOT exceed `MAX_OBJECTS_PER_VOLUME = 1,000,000` objects or `MAX_VOLUME_SIZE = 200 GiB` of stored (erasure-expanded) bytes, which holds at least 100 GiB of content on any erasure split with M ≤ K; each object MUST NOT exceed `MAX_OBJECT_SIZE = 1 GiB`.

© Cowboy Labs, Inc. 2026

Object chunks are 32 MiB at widths up to 6, 64 MiB at widths 7–12, and 128 MiB at widths 13 and above. One commit can pay for only a bounded amount of on-chain work, its commit budget. For some changes the client checks the complete change against that budget: changes that deepen the manifest or split its nodes, changes made while storage epochs are queued for rent settlement, and inserts or deletes that repack every following manifest leaf in a volume of multi-chunk objects. Before signing, the client refuses a change whose commit the budget cannot pay for. An object replacement that the budget cannot pay for can be done as separate delete and recreate commits only if the preview of each commit fits the budget, and the object is absent between the two commits. CIP-9 §14 specifies these rules.

Signing funds the full Access reservation for finalizing a staged manifest, including the extra Accesses for queued rent epochs and native routing. That reservation may use at most **1,116** structural Accesses, within the **1,507** charged Accesses allowed for an ordinary System transaction.

Object paths are UTF-8 strings of up to `MAX_OBJECT_PATH_LENGTH = 512 bytes`. The full address of an object is:

`/{owner_address}/{volume_name}/{object_path}`

5 The CBFS data plane

CBFS stores and serves volume data. The implementation is divided into these crates:

Crate	Responsibility
`cbfs-types`	Shared protocol types: `VolumeId`, `ShardId`, `NodeId`, `PlacementRecord`, `ObjectDescriptor`, `RpcRequest/Response`, `Operation`.
`cbfs-crypto`	AES-256-GCM, BLAKE3, DEK generation and envelope wrapping.
`cbfs-erasure`	Reed-Solomon over GF(2^8) via `reed-solomon-erasure` v6 with SIMD.
`cbfs-transport`	QUIC client/server (`quinn`) with TLS 1.3 and bincode-framed RPCs.
`cbfs-store`	File-per-shard blob store (`FileBlobStore`) on Relay Nodes.

Crate	Responsibility
`cbfs-manifest`	Sorted client-side volume index and canonical content-chunked Merkle DAG.
`cbfs-placement`	Persistent `PlacementStore` for shard-to-node assignments with CAS and replica merge.
`cbfs-hooks`	Pluggable traits (`AuthProvider`, `ManifestRegistry`, `MeteringSink`) with standalone and Cowboy-mode implementations.
`cbfs-auth`	Cowboy-facing auth runtime: delegation loading, request signing, owner token minting, relay and chain-state caches.
`cbfs-sdk`	High-level Volume API: create, open, put, get, commit, with local manifest and placement synchronization.
`cbfs-fuse`	POSIX FUSE mount (inode table, write-back cache, sync daemon, extended attributes).
`cbfs-node`	The Relay Node daemon: QUIC server, garbage collection, heartbeats, repair, drain.
`cbfs-cli`	Operator CLI: `login`, `auth`, `volume`, `put`, `get`, `mount`, `commit`, `rebalance`.

5.1 Object representation

Each object is described by an **ObjectDescriptor** stored in the manifest. Its `chunks: Vec<ChunkDescriptor>` holds one descriptor per object chunk. Each chunk descriptor names its own stripe, with its own `shard_id`, `shard_hashes` and `shard_chunk_roots`. `object_chunk_shard_id(volume_id, write_id, chunk_index)` derives each ID from the domain, the volume, the write ID and the zero-based chunk position encoded as a little-endian `u32`. The derivation (below) does not include the path, so renaming an object leaves its shard IDs unchanged. The fields separate whole-object identity from per-chunk commitments:

Field	Type	Meaning
`object_path`	`String`	≤ 512 bytes, UTF-8
`content_type`	`Option<String>`	Optional ASCII content type
`write_id`	`[u8; 16]`	Fresh CSPRNG per write, isolates versions
`content_hash`	`[u8; 32]`	BLAKE3 of the complete plaintext in chunk order
`chunks`	`Vec<ChunkDescriptor>`	Ordered commitments; position bound into each chunk ID
`chunks[].content_hash`	`[u8; 32]`	BLAKE3 of this chunk's plaintext
`chunks[].ciphertext_hash`	`[u8; 32]`	BLAKE3 of stored bytes before erasure coding, for either visibility
`chunks[].encryption_nonce`	`[u8; 12]`	Fresh random per private chunk; all zero for public chunks
`size_bytes`	`u64`	Plaintext size
`chunks[].size_bytes`	`u64`	Stored length before erasure padding: plaintext + 16-byte tag for private chunks, plaintext length for public chunks
`chunks[].shard_id`	`[u8; 32]`	`BLAKE3("cbfs.object-chunk.v1" ‖ volume_id ‖ write_id ‖ u32_le(chunk_index))` — opaque to Relay Nodes
`chunks[].erasure_k`, `chunks[].erasure_m`	`u8`, `u8`	Per-chunk erasure parameters

Field	Type	Meaning
`chunks[].shard_hashes`	`[[u8; 32]; K+M]`	BLAKE3 of each shard in that chunk's stripe, used for integrity
`chunks[].shard_chunk_roots`	`[[u8; 32]; K+M]`	Per-shard chunk Merkle roots in that stripe, used for PoR

A Relay Node sees a `shard_id` hash and never a path. It cannot learn which object a shard belongs to or detect an overwrite. The Scoping and Concurrency section explains what this means for authorization.

Each write MUST use a fresh CSPRNG-generated `write_id` and record the aggregate `content_hash = BLAKE3(all_plaintext_bytes_in_order)`. Every chunk, public or private, records its own plaintext `content_hash`, stored-byte `ciphertext_hash`, and stored length before erasure padding. A private chunk's stored length includes a 16-byte authentication tag. A public chunk's stored bytes equal its plaintext, so its plaintext and stored-byte hashes are equal. There is no single object-level ciphertext hash. Descriptor validation checks lengths, the uniform chunk layout and grid, and the position-bound IDs before any allocation or data transfer. Readers check content hashes during reads.

5.2 Encryption

A private volume's DEK MUST be 32 bytes drawn from a CSPRNG. Each private object chunk MUST be encrypted independently with AES-256-GCM using a fresh random 12-byte nonce; deterministic nonces MUST NOT be used, and a (DEK, nonce) pair MUST NOT be reused across chunks or writes. The AAD MUST be the chunk's `shard_id`, which binds `volume_id`, `write_id` and the chunk index through the derivation above and does not change on rename. The authentication tag MUST be appended to the chunk's ciphertext per RFC 5116. Readers MUST verify each chunk's `ciphertext_hash` before decryption and `content_hash` afterward. Public volumes store plaintext objects and manifests.

5.3 Manifest

The client indexes file, symlink and directory entries by path. The committed representation MUST follow the canonical, content-chunked Merkle DAG in CIP-9 §3.1. Entries are ordered by UTF-8 path bytes, grouped into leaf nodes, and reduced through interior nodes

to one root. Each node's locator is the BLAKE3 hash of its canonical stored encoding, and `StorageCommitment.manifest_root` is the top node's locator.

The client encrypts each private-volume node before storage, so the root commits to ciphertext. Public-volume nodes are plaintext. Unchanged nodes keep their locators across commits, and clients publish only newly added nodes, addressed by `manifest_node_shard_id(volume_id, locator)`. Readers MUST verify nodes against their expected locators. CIP-9 §3.1 specifies the canonical encoding, chunk boundaries and encryption domains. Commit and Publication covers ordering and authority.

Each manifest DAG node MUST be encoded into K+M erasure-coded shards at the volume-bound address, with at least K placed at commit and any remainder owed (CIP-9 §5.4.1):

```
manifest_node_shard_id(volume_id, locator) =
    BLAKE3("cbfs.manifest-node.v2" || volume_id || locator)
```

The hash input MUST be the domain's literal UTF-8 bytes followed by the 32-byte volume ID and the 32-byte node locator, without delimiters or a terminating NUL. Readers MUST start from the authoritative `StorageCommitment.manifest_root`, verify fetched nodes against their expected locators, and traverse the child locators. Private-volume nodes MUST be decrypted with the DEK during traversal. There is no well-known single manifest blob address.

Public-volume readers MAY use `GET_MANIFEST`, supplying the authoritative root and erasure parameters. The response is an entry map in place of the stored DAG encoding; readers MUST rebuild the canonical DAG and verify its root against the authoritative root supplied in the request. Private volumes have no such alternative.

5.4 Placement

Each object chunk has one **PlacementRecord** that assigns its stripe's K+M shards to the volume's K+M distinct Relay Nodes, shard index `i` to `relay_nodes[i]`. Each manifest DAG node also has its own record:

Field	Type	Meaning
`volume_id`	`[u8; 32]`	Volume binding checked against locally stored shard metadata
`shard_id`	`[u8; 32]`	Opaque shard identifier

Field	Type	Meaning
`version`	`u64`	CAS version for atomic updates during repair
`assignments`	`[{shard_index: u8, node_id: [u8;32], addr: SocketAddr}]`	Node and network address per shard index
`erasure_k`, `erasure_m`	`u8`, `u8`	Duplicated from descriptor for repair without manifest
`ciphertext_size`	`u64`	Chunk's pre-padding stored length, equal to `ChunkDescriptor.size_bytes`
`shard_hashes`	`[[u8; 32]; K+M]`	Duplicated for integrity checks during repair
`shard_chunk_roots`	`[[u8; 32]; K+M]`	Duplicated per-shard Merkle roots over small PoR chunks

PlacementRecords MUST remain mutable, versioned records separate from the manifest. For object chunks, they MUST duplicate the corresponding ChunkDescriptor's erasure parameters, stored size, shard hashes and PoR roots so repair does not require decryption. They MUST be replicated across all assigned Relay Nodes; updates MUST use compare-and-swap through these RPCs:

- `PutPlacement(shard_id, record, expected_version)`: CAS store or update.
- `GetPlacement(shard_id)`: fetch.
- `ReplicatePlacement(shard_id, record)`: peer-to-peer replication, signed by the sending relay's identity key.

A PlacementRecord MUST carry its `volume_id`. When accepting an update or replication for a shard assigned to itself, a Relay Node MUST reject a record whose volume conflicts with that shard's existing local volume binding. A placement update cannot rebind stored bytes to another volume.

Placement lives outside the encrypted manifest. That is what lets repair workers act on private volumes without the right to decrypt them.

Writes are index-aligned. The volume's `StorageCommitment.relay_nodes` names exactly K+M distinct relays, fixed at creation and changed only by the chain's relay rotation. Shard index

`i` of every stripe goes to `relay_nodes[i]` and to no other relay. The writer does no selection, scoring or fallback.

5.5 Erasure coding

Cowboy uses Reed-Solomon over GF(2^8) through the `reed-solomon-erasure` v6 crate with the `simd-accel` feature. The defaults are K=4 data shards and M=2 parity shards, for 1.5× storage overhead.

Each object chunk (one stripe) is split into K+M shards. Under the default 4/6 scheme, its 6 shards go to 6 Relay Nodes, and any 4 reconstruct that chunk's ciphertext, so a fully placed stripe survives any 2 simultaneous Relay Node failures. A degraded write commits with fewer than K+M shards placed (see Write), and its stripe tolerates fewer failures until repair re-creates the owed shards. A one-chunk object is one stripe. Accounts are billed for `effective_size = raw_size × (K+M)/K`.

Implementations MUST support `MIN_K = 2` ≤ `K` ≤ `MAX_K = 16` and `MIN_M = 1` ≤ `M` ≤ `MAX_M = 8`. Accounts MAY choose higher redundancy within this grid at creation time. Creation MUST refuse parameters outside the grid (`K+M` ∈ `3..=24`). Width 32 is the bound for Cells metering and row decoding of stored rows. It does not widen the range of widths a volume can be created with.

Each chunk's stored bytes (ciphertext for private volumes, plaintext for public volumes) MUST be padded for Reed-Solomon encoding; readers MUST truncate reconstruction to the chunk's recorded stored length. Each shard MUST be hashed with BLAKE3, with the hash stored in its ChunkDescriptor and duplicated in its PlacementRecord. The small PoR chunks within each shard are separate from object chunks. Readers MUST verify shard hashes before reconstruction; repair workers MUST verify reconstructed shards before persistence.

5.6 Transport

Relay Nodes MUST use QUIC over TLS 1.3 on a single stream per RPC. Each frame is a `[version: u8][length: u32 big-endian][bincode payload]` tuple capped at `MAX_FRAME_SIZE = MAX_SHARD_TRANSFER_SIZE + 64 KiB`, with `MAX_SHARD_TRANSFER_SIZE = 256 MiB`. The RPC protocol is:

```
{ request_id: [u8;16], op: Operation, shard_id: [u8;32], shard_index: u8,
auth_token: Vec<u8>, payload: Vec<u8> }
```

`Operation` is bincode-encoded by ordinal, so its order is part of the wire format. The values, in order, are: `PutShard`, `GetShard`, `DeleteShard`, `ProveShard`, `GetPlacement`, `PutPlacement`,

`ReplicatePlacement`, `Ping`, `RelayHandshake`, `SyncPeers`, `PeerRelayPushShard`, `PeerPlacementUpdate`, `AuditGetShard`, `GetManifest`, `GetPaidShard`, `CompletePaidRead`, `GetChannelShard`, `CompleteChannelRead`, `SubmitRootUpdate`, `GetRootLeaf`, `AnswerOmission`. Responses carry a matching `request_id` and a status code, also by ordinal: `Ok`, `ErrAuth`, `ErrNotFound`, `ErrCapacity`, `ErrInvalid`, `ErrConflict`, `ErrBusyUntilNextAnchor`.

Client-to-relay operations requiring authorization MUST validate a CapToken; public-volume reads remain open. Peer-relay authentication follows Drain and Peer-Push.

`RelayHandshake` MUST sign `keccak256("cbfs:relay-handshake:v2:" || node_id || client_nonce || timestamp_ms_be || channel_cert_sha256)` with the relay's Ed25519 identity key. `node_id` is 32 bytes, `client_nonce` 16 bytes and `timestamp_ms_be` a big-endian `u64`. `channel_cert_sha256` is the SHA-256 of the DER certificate the relay serves on its TLS channel. The client sends the hash of the certificate it observed on its own TLS channel, so a TLS-terminating proxy cannot obtain a signature over its own certificate. Clients MUST verify against the cached on-chain `relay_identity_pubkey` and the certificate hash they observed, and reject timestamps outside ±30 seconds.

6 Writing and reading data

A write has three phases. The client first uploads each chunk's shards to Relay Nodes. It then publishes the new manifest DAG nodes and their placement records. Last, it commits the new manifest root on chain. Only the third phase is a transaction, and other readers see the write only after the chain accepts that commit.

A read runs the other way. The reader resolves the volume's current root from chain state, fetches and verifies the manifest from that root, fetches K valid shards for each chunk of the object, and decodes and verifies the chunks in order.

The subsections below give each step. They assume the volume already exists (Create) and the writer holds a CapToken (The CapToken Authorization Model).

6.1 Write (object transfer)

An authorized runner or client:

1. Splits the plaintext into the volume grid's bounded object chunks (one chunk even for an empty object), generates one fresh 16-byte `write_id`, and hashes the full plaintext in order. For each zero-based position, it derives `shard_id = BLAKE3("cbfs.object-chunk.v1" || volume_id || write_id || u32_le(chunk_index))`.

2. Encrypts each private chunk independently under the Encryption rules. Public chunks stay plaintext with a zero nonce. It records each chunk's plaintext hash, stored-byte hash and pre-padding stored length, which includes one 16-byte tag for a private chunk.
3. Reed-Solomon encodes each chunk's stored bytes into its own K+M shards, and computes each whole-shard hash and its small-chunk PoR Merkle root.
4. For each stripe, sends shard `i` to `relay_nodes[i]` with an authenticated `PutShard` RPC. Uploads run concurrently up to `max(1, floor(8 MiB / shard size))` at a time, capped at the stripe width. `ErrCapacity` or an unreachable relay leaves that index *owed*. Any other refusal fails the write whatever the other indexes did, after the attempts already started finish. The stripe succeeds when at least K shards are placed; with fewer the write fails.
5. Records one `ObjectDescriptor` with the whole-object plaintext hash and length and the ordered `ChunkDescriptor` vector, and builds one `PlacementRecord` per chunk that names `relay_nodes[i]` at every index, owed or not.

All of these steps are local or off-chain. The client submits no transaction.

A stripe with owed indexes is a degraded write. The commit lists every owed index in `owed_shards`, and the chain accepts at most M owed indexes per stripe, so each committed stripe has at least K placed shards. An owed index is credited to no relay, earns no rent and is not challenged by PoR. The relay at that index reconstructs the shard from K peers during repair and then settles it with a `SettleOwedShards` transaction signed by its identity key, after which the chain credits it and admits it to PoR. Until then the stripe tolerates the loss of M minus the number of owed indexes. CIP-9 §5.4.1 specifies the rule.

6.2 Commit and publication

Manifest publication follows CIP-9 §12.2 and has two steps:

1. The client stores the manifest DAG's added nodes at shard IDs derived from the volume ID and each node's locator, as specified in Manifest, and publishes their PlacementRecords to every assigned Relay Node via `PutPlacement`; manifest-node placement publication MUST succeed on every assignee that holds its shard before step 2; an owed index's relay is skipped and receives the record through repair. Nodes reachable from the previous root MUST remain intact while that root is authoritative.
2. The client submits a transaction carrying the new root, `prev_root` (the root this change was built on), and the lists of added and removed shards, through the authorized control-plane path. The chain MUST compare `prev_root` with the current root before advancing `StorageCommitment.manifest_root`; in the same state transition, the added and removed

shards update the native RAS inventory and `live_shard_count`. Each added and removed shard reference carries its real nonzero `chunk_root`. Each added or removed stripe must contain all `w` indexes exactly once, with a common size, kind and path tag, and a removal must match the existing stripe row's root at every index and its size. PoR snapshots the selected stripe index's chunk-root at challenge issuance. Control-plane writes are account transactions that the client signs and submits itself, carrying the owner authorization where the instruction needs one (CIP-9 §12.2.1). Scoped runner commits follow Scoping and Concurrency. A mount-grant holder publishes a root only through its grant row, as the transaction sender (see Direct-Client Grants).

The client publishes new nodes first and advances the on-chain root second. If the chain commit fails, the previous root and its nodes remain usable.

A relay deletes stored bytes only under these rules. Deletion MUST protect both the current committed shard set and newly published bytes that no commit references yet. Every accepted publication or republication, including identical bytes at an existing node address, MUST establish a bounded tentative hold: a marker with a deadline that stops the relay deleting those bytes while they wait for a commit. A commit event predating that publication MUST NOT delete its bytes or clear its hold. The hold ends when a commit that includes the publication puts the bytes into the committed shard set, or when its deadline expires. Local publication and deletion MUST coordinate atomically so an older deletion decision cannot remove a concurrent replacement. Publishers MUST keep their holds valid through commitment and re-establish publication after a hold lapses.

Even after a hold expires, relays MUST preserve shards in the chain's current native RAS inventory for a volume that has not entered `GARBAGE_COLLECTING`, including private manifest nodes. A relay does not need to decrypt the DAG to tell which shards are live. A stale removal event alone cannot authorize deletion, and if the relay cannot tell whether a shard may be deleted, it MUST wait. Finalized `GARBAGE_COLLECTING` status authorizes the relay to delete all of the volume's shards, even while the chain is still removing the volume's inventory rows a bounded number at a time, except for shards still referenced by another live volume. CIP-9 §12.2 specifies the chain-state signal and the publication rules. Garbage collection of orphan shards, which no committed manifest references, may delete abandoned uploads only under those rules.

6.3 Read (object transfer)

1. Fetch the manifest root from the authoritative Storage Manager state (or a cached value whose age is bounded).
2. Fetch and verify the manifest DAG from its committed root, using the shard addresses derived from the volume ID and each node's locator, as specified in Manifest. Decrypt

private-volume nodes with the DEK. Public readers MAY use the verified `GET_MANIFEST` alternative described there.

3. Resolve `object_path` to an `ObjectDescriptor` and validate its path, lengths, uniform chunk layout and grid, and position-bound IDs. Get the placement for every chunk's `shard_id`.
4. For each chunk in vector order, request K valid shards, verify their hashes against that chunk's manifest commitments, and fail over to another shard on timeout or corruption.
5. Reconstruct that chunk's stored bytes and remove the padding using `ChunkDescriptor.size_bytes` (duplicated as `PlacementRecord.ciphertext_size`). Verify `ciphertext_hash` for either visibility. Decrypt a private chunk with its recorded nonce and `shard_id` AAD, and verify each chunk's plaintext `content_hash`.
6. Deliver verified chunks in order, and at the end of the full read verify the aggregate plaintext hash against `ObjectDescriptor.content_hash`. A read that fails partway may already have written the first part of the object to the caller's output; callers MUST discard failed output.

6.4 FUSE mount

CBFS ships a FUSE filesystem (`cbfs-fuse`) that exposes a volume as a POSIX directory with write-back caching. A write goes at once into an in-memory cache of changes not yet uploaded (the dirty cache). A background sync daemon encrypts, shards and pushes the data to Relay Nodes every `DEFAULT_SYNC_INTERVAL = 5 seconds` ($\geq$ `MIN_SYNC_INTERVAL = 1 second`). The local cache is bounded by `MAX_LOCAL_CACHE_SIZE = 10 GiB` per mount.

Extended attributes under the `cbfs.*` prefix expose shard and encryption metadata, read-only:

Attribute	Value
`cbfs.status`	`committed` (in the manifest, no local change), `dirty` (local change not yet pushed) or `clean` (not in the manifest, no local change)
`cbfs.volume_id`	Volume ID, hex
`cbfs.content_hash`	Full 32-byte BLAKE3 of the plaintext, hex
`cbfs.size`	Plaintext size in bytes
`cbfs.ciphertext_size`	Stored size in bytes
`cbfs.encrypted`	`aes-256-gcm` on a private volume, `none` on a public volume

Attribute	Value
`cbfs.chunk_nonces`	Per-chunk nonces, hex, comma-separated
`cbfs.chunk_count`	Number of object chunks
`cbfs.erasure`	`K+M` (e.g., `4+2`)
`cbfs.chunk_ids`	Per-chunk shard IDs, hex, comma-separated
`cbfs.write_id`	Write ID, hex

A path that is not yet in the manifest exposes only `cbfs.status` and `cbfs.volume_id`.

Cowboy-mode mounts refresh owner tokens before expiry and keep the requested access mode, so a session can run for a long time without widening token scope.

7 Consistency

The authoritative manifest is the one matching the root last accepted by the chain (READ_COMMITTED: a reader sees only manifests whose root the chain has accepted). Readers MUST verify that match. A committed root guarantees integrity and says nothing about availability: it cannot supply bytes when too few relays serve its shards.

A mount reads its own writes immediately and never moves to an older root. It switches to a new root only when a proof from finalized chain state shows that root, and only if the proof is no older than the newest proof it has already verified. Other mounts see a change once the chain accepts its commit and their next refresh succeeds. A change written on one healthy mount is visible on every other healthy mount within `V = 2·S + W + 2·C + A + R`. Here `S` is the sync interval (5 s). `W` is the commit window (5 s), the longest a change waits before its batch is submitted. `C` is the commit latency (about 1.5 s at 1 s blocks). `A` is the proof age (10 s), how far the mount's view may trail the newest finalized block. `R` is the time the refresh itself takes. By default `V` is about 28 s plus `R`, and `A + S + R` after a successful `fsync()`.

A mount sends its pending changes as one batch per commit. If the chain refuses the commit because another commit changed the root first, the writer re-applies its batch on top of the new root and tries again, so commits to disjoint paths are all accepted. For one path, the first commit the chain accepts wins: the loser's mount shows the winner's version and its next `fsync()` fails with `EIO`. Two mounts creating one directory merge. A path created as different kinds keeps the committed entry, and the loser's entry moves to `<name>.conflict-<mount id>`. CIP-9 §12.1.5 specifies the full model.

© Cowboy Labs, Inc. 2026

8 Relay Nodes

A Relay Node is a storage participant separate from validators and runners. It persists erasure-coded shards, serves reads and writes to authorized clients, takes part in relay-to-relay repair, answers Proof-of-Retrievability challenges, and reports capacity and health to the chain.

8.1 Registration and stake

A Relay Node registers by submitting a `RegisterRelay` transaction to the Relay Registry (`0x0B`) carrying its owner address, `node_id`, Ed25519 `relay_identity_pubkey`, owner and relay authorizations, endpoint, `capacity_bytes`, an optional region and an optional status URL. Registration requires a stake of at least `MIN_RELAY_STAKE = 33,000 CBY` (governance-tunable). It produces a **RelayNodeProfile**, which includes these fields:

Field	Type	Meaning
`address`	`[u8; 20]`	Relay account address; distinct from the 32-byte node ID
`stake_amount`	`u128`	Self-bonded stake in base units (1 CBY = 10^9)
`capacity_bytes`	`u64`	Advertised capacity
`used_bytes`	`u64`	Consensus-derived committed shard storage
`status`	`RelayStatus`	`ACTIVE`, `DRAINING`, `DISABLED`, or `INACTIVE`
`updated_epoch`	`u64`	Storage epoch of the last profile update
`shards_held`	`u32`	Active shard count
`shards_lost`	`u32`	Historical loss counter; no assignment reads it
`region`	`Option<String>`	Operator-declared region code

`UpdateRelay` changes a relay's endpoint, capacity, region or status URL and refreshes `updated_epoch`. There is no heartbeat instruction, and liveness is the `status` field. Auto-assignment and rotation consider only `ACTIVE` relays with free capacity. The governance-controlled

auto-drain policy moves an over-capacity or under-staked relay out of `ACTIVE` without any action by its operator. `DRAINING`, `DISABLED` and `INACTIVE` relays receive no new assignments.

A Relay Node MAY unstake after a `RELAY_UNSTAKE_DELAY_EPOCHS = 1` storage-epoch (~24 h on production genesis) starting after full drain completion (see Drain and Peer-Push).

8.2 Repair

Relay Nodes run a repair loop every `REPAIR_CHECK_INTERVAL_SECS = 300` seconds. Each pass has two parts.

8.2.1 Repair local shards

For each PlacementRecord that names this node as an assignee, the node verifies its local shard against the expected `shard_hash`. If the shard is missing or corrupt, the node:

1. Fetches K healthy shards from peers listed in the PlacementRecord.
2. Reconstructs the ciphertext with Reed-Solomon and truncates it to `ciphertext_size`. The PlacementRecord duplicates `ciphertext_size` for this step; without it, the padding would corrupt the reconstruction.
3. Re-encodes, extracts the expected shard, verifies its hash and persists it locally.

Any number of nodes can run this local repair (self-heal) at the same time, because it writes only to local storage and never to the PlacementRecord.

8.2.2 Replace unavailable relays

For each PlacementRecord, the node probes all assignees. If a node stays unreachable past a timeout:

1. Among the live assignees, the node with the lowest `shard_index` is the unique leader for this PlacementRecord. Indexes are distinct, so there are no ties.
2. The leader selects a replacement Relay Node from its peer list, excluding current assignees, taking the peer with the most free capacity.
3. The leader reconstructs the missing shards from the surviving shards and pushes them to the replacement with `PutShard` or the signed `PeerRelayPushShard` RPC.
4. The leader writes a new PlacementRecord with the updated assignment and `version + 1` through a CAS `PutPlacement(shard_id, record, expected_version)`. The CAS fails if another node concurrently replaced a different dead node in the same record; the leader then retries in the next cycle.

© Cowboy Labs, Inc. 2026

5. The leader replicates the updated PlacementRecord to all current assignees with `ReplicatePlacement`.

Local repair also re-creates an owed shard (see Write): the relay at the owed index rebuilds it from K peers and then settles it on chain. Repair needs at least K healthy shards. Under the default K=4, M=2 scheme, three permanent shard losses before repair completes make the stripe unrecoverable. Orphan shards without a referencing manifest MUST be garbage-collected after `ORPHAN_SHARD_TTL_SECS = 86,400` seconds.

8.3 Drain and peer-push

A relay may leave voluntarily, or governance may order it to leave, by moving to `Draining` status. A drain worker goes through the relay's assigned shards and sends each one directly to a replacement relay with these signed RPCs:

- `PeerRelayPushShard(sender_node_id, timestamp, shard_id, shard_index, old_record_hash, shard_bytes_hash, shard_bytes, signature)`
- `PeerPlacementUpdate(sender_node_id, timestamp, old_record_hash, new_record_hash, new_record, signature)`

The sending relay signs both with its identity key over domain-tagged BLAKE3 digests (`cbfs:peer_push_shard:v1`, `cbfs:peer_placement_update:v1`). The receiver rejects a message whose timestamp is outside a 60 s window. When `shards_remaining == 0`, the draining node writes a signed `RelaySelfDisableRequest` and leaves the active set.

9 Proof of retrievability

A Relay Node earns rent for holding shards, so it has an economic reason to discard shards quietly and keep claiming rent. RAS counters this with periodic **Proof of Retrievability (PoR)** challenges. A challenge is cheap to verify, and a relay that fails one loses stake.

9.1 Challenge protocol

An on-chain timer triggers a challenge every `POR_CHALLENGE_INTERVAL = 7,200 blocks` (~2 h at 1 s/block). The challenge selects a random `(shard_id, shard_index, byte_offset, byte_length)` tuple on an active relay. Consensus reads the selected stripe's row in the native RAS inventory and copies that index's `chunk_root`, the stripe's common `shard_size` and the sequence number of the event that published the stripe into the challenge. The PoR weight

page, the paged table that PoR draws its random targets from, already records the stripe width, so choosing a target at block start takes five state reads. The relay MUST respond within `POR_RESPONSE_WINDOW = 50 blocks` with:

- The requested byte range (ciphertext bytes) from the named shard.
- A `byte_range_proof` that folds the challenged ciphertext chunks up to the shard's `chunk_root` (`cbfs_erasure::chunk_root`).

The fixed congestion grace, extra time in case a congested network delays the response, is `POR_RESPONSE_GRACE = 20` additional blocks. A valid response is accepted through `deadline_block + 20` inclusive; an unanswered challenge may settle only at a strictly greater height. CIP-9 §5.6 specifies the exact `u64` saturating arithmetic and boundary (normally issue height +70 inclusive for responses, +71 onward for settlement).

The on-chain verifier:

1. Recomputes the chunk leaves from the submitted ciphertext bytes and verifies that `byte_range_proof` resolves to a chunk-root.
2. Requires that chunk-root to equal the selected stripe index's `snapshot_chunk_root` recorded at challenge issuance.

Every verification input is public: `snapshot_chunk_root`, `shard_id`, `shard_index` and the relay's ciphertext bytes. The client encrypts data before erasure coding and hashing, so shard commitments do not depend on the DEK, and the chain can verify PoR for a private volume without the DEK.

A relay that stored the shard answers from local disk. A passed challenge shows that the shard was retrievable at challenge time, backed by stake and repair. It does not show that the relay stored the shard continuously: within the response window a relay could rebuild the challenged range from the other relays that hold shards of the same stripe. PoR therefore deters silent discard through economics, since a relay that keeps missing can be evicted and slashed. It gives no cryptographic proof of dedicated storage.

Other inventory updates during the response window do not change this challenge's `snapshot_chunk_root`. Removing or replacing the challenged shard voids an unanswered challenge, and adding the shard back later cannot rewrite its snapshot. If the volume enters `GARBAGE_COLLECTING` after issue, the unanswered challenge also voids at settlement, even while its inventory row still waits for the chain's bounded cleanup. A voided challenge refunds the full bond, frees the slot the challenge occupied among open challenges, and causes no miss, fee, slash or bounty. Reversible `DELETED` status does not void a challenge.

© Cowboy Labs, Inc. 2026

9.2 Outcomes

Outcome	Consequence
Valid response within window or congestion grace	Challenge passed; bond returned less `POR_CHALLENGE_FEE`; no slash
Invalid proof	Response transaction rejected; challenge remains open for a valid response or expiry settlement
Unanswered challenge after window and congestion grace	Miss alarm without stake slash; with slashing enabled the per-shard consecutive-miss count advances; bond returned less `POR_CHALLENGE_FEE`
Third consecutive miss with slashing enabled	Relay disabled, `shards_lost` increments once, and `RELAY_EVICTION_PENALTY` is slashed up to available stake; challenger gets full bond refund and may receive a bounded pool-funded bounty
Shard incarnation removed/replaced or volume enters `GARBAGE_COLLECTING`	Challenge voided; full bond refund and open-slot release, with no miss, fee, slash, or bounty

A governance-tunable 1% share of storage rent plus the retained `POR_CHALLENGE_FEE` funds challenge issuance. A confirmed third consecutive miss may pay a challenger bounty from the funded pool. With slashing disabled, an unanswered challenge emits an alarm and does not increment the consecutive-miss count or touch stake. The current settlement path does not charge `POR_MISS_PENALTY` or `POR_FRAUD_PENALTY` for a single miss or a rejected invalid proof.

10 The CapToken authorization model

Storage authorization combines chain-issued runner tokens with client-signed owner-auth tokens. CapTokens MUST be encoded as `[version: u8][bincode payload]`, with `0` for runner tokens and `1` for owner-auth tokens. Signing inputs that cover a token MUST include its version byte. Relays validate credentials against authoritative, freshness-bounded chain state as described under Cache Freshness.

© Cowboy Labs, Inc. 2026

10.1 Runner CapTokens (chain-issued)

A **Runner CapToken** MUST be issued through the Storage Manager (`0x0A`) at job dispatch and stored on-chain. It MUST include the following fields:

Field	Type	Meaning
`version`	`u8`	`0`
`token_id`	`[u8; 16]`	Deterministically derived
`token_secret`	`bytes`	Entropy; validation requires secret match
`volume_id`	`[u8; 32]`	Target volume
`volume_name`	`String`	For logging, not validation
`runner_address`	`Address`	Restricts to a single runner
`access_mode`	`AccessMode`	`ReadOnly`, `WriteOnly`, `ReadWrite`
`path_prefix`	`String`	Canonical attachment prefix (e.g., `workspace/`); `""` means whole volume
`max_bytes`	`u64`	Per-token quota
`valid_from`, `valid_until`	`u64`, `u64`	Block heights
`mount`, `sync_interval`	`bool`, `u32`	Mount parameters

The `token_id` and `token_secret` are derived deterministically from job metadata, runner address, block height, attachment index and VRF seed, which prevents pre-image attacks. To validate a presented runner CapToken, a Relay Node looks up the stored copy at `StorageManager[token_id]` and checks for a byte-for-byte match, TTL, runner address, volume status and access mode. The chain enforces prefix restrictions when it accepts a volume attachment, and clients and coordinators check them. Opaque shard RPCs carry no paths, so a relay cannot check containment.

Runner CapTokens MUST have a validity window of at least `MIN_VOLUME_CAP_TOKEN_BLOCKS = 600,000` blocks (~7 days at 1 s blocks), which covers typical job durations.

© Cowboy Labs, Inc. 2026

10.2 Owner CapTokens (client-signed)

A delegated CBFS key signs an **Owner CapToken** (`OwnerCapTokenV1`) locally. The signer MUST be admitted either as the volume owner or through the Direct-Client Grants contract. A delegation cert alone gives no authority over another owner's volume.

Field	Type	Meaning
`version`	`u8`	`1`
`volume_id`	`[u8; 32]`	Target volume
`volume_name`	`String`	For logging, not validation
`owner_address`	`Address`	Volume owner; MUST match the authoritative commitment owner
`access_mode`	`AccessMode`	`ReadOnly`, `WriteOnly`, `ReadWrite`
`path_prefix`	`String`	Requested scope; MUST be empty for a grant-scoped token
`valid_from_ms`, `valid_until_ms`	`u64`, `u64`	Unix milliseconds (wall-clock, short TTL)
`mount`, `sync_interval`	`bool`, `u32`	Mount parameters
`delegation_cert`	`DelegationCert`	Embedded wallet delegation cert
`owner_signature`	`bytes`	Ed25519 over all other fields, including the version byte

An owner-auth token MUST have a maximum TTL of **15 minutes** for reads and puts (`OWNER_TOKEN_OPEN_TTL_SECS = 900`), and **24 hours** for mounts (`OWNER_TOKEN_MOUNT_TTL_SECS = 86,400`). Long-lived operations refresh the token before expiry and keep its scope. Relays validate owner-auth tokens without storing each token on-chain. They MUST check the wallet's secp256k1 signature on the embedded DelegationCert, the token's Ed25519 signature under the cert's key, cert expiry, `chain_id`, `aud == "cowboy:cbfs"`, and `scope == "cbfs"`. The cert has no `network` field: its signature is computed over the CIP-40 `ChainIdentity` signing preimage, which binds the chain instance and its genesis network. The cert MUST be

registered and unrevoked for its wallet. Validation MUST also check the authoritative volume identity, owner and status, token TTL (±30 seconds clock skew), and the requested operation against the access mode. Ownership admission requires `delegation_cert.wallet_address == commitment.owner`; otherwise the direct-client grant rules apply. Path scope MUST NOT be treated as a relay-enforced confidentiality boundary for opaque shards.

The token signature MUST cover the canonical `OwnerCapTokenSigningPayload`, including the version byte and embedded cert. `cowboy-protocol-codec` owns the RAS wire types and signing-byte helpers; clients and validators MUST use that canonical encoding, as specified in CIP-9 §12.2.1.

10.3 DelegationCert

A **DelegationCert** binds an Ed25519 CBFS key to a Cowboy wallet. It MUST be signed with the wallet's secp256k1 key over the Keccak-256 digest of the canonical `DelegationCertSigningPayload`, and MUST expire no more than 30 days after issuance. It binds the wallet, delegated key, scope, service class, chain ID, audience and expiry, and its signing preimage binds the running `ChainIdentity`. The signing client registers the delegation on-chain, and the cert can then accompany many owner-auth tokens.

The signed service class limits mount authority: a non-mount service delegation MUST NOT authorize a mount token. Relays MUST verify that the registered delegation matches the cert, including its service class and scope. `cowboy-protocol-codec` defines the canonical fields and signing bytes, and the CBFS authorization layer uses them.

The client stores the delegated CBFS key encrypted at `~/.cbfs/cbfs_key.enc` under AES-256-GCM with a user-supplied passphrase (or an environment variable for headless use). It decrypts the key on demand into locked memory and zeroes it after signing. `cbfs auth rotate` replaces the key, and `cbfs auth revoke` posts a revocation to the Storage Manager. Cache Freshness bounds how long a revocation takes to reach relays.

10.4 Direct-client grants

CIP-9 §7.7 and §7.7.1 define a single on-chain mount allowlist for access to another owner's volume. A grant without `grantee_signing_key` authorizes job dispatch only. A direct-client grant also binds the grantee's delegated Ed25519 signing key. Its canonical path prefix MUST be empty, because opaque shard addressing and one DEK per private volume cannot enforce a narrower direct-client scope.

A grantee signs an `OwnerCapTokenV1` with its own DelegationCert and key, and `owner_address` names the volume's owner. In addition to ordinary token validation, a relay MUST require:

- The grant's grantee address matches the cert wallet by bare address, and `grantee_signing_key` matches the cert's delegated key.
- The token's mode is a subset of the grant's allowed operations, and its path prefix is empty.
- The grant is unexpired at the relay's block-height view. An expiring grant MUST be refused if that view is unavailable.
- `tee_required` is false; a direct client does not satisfy a runner TEE requirement.

A read grant permits volume-open metadata, committed-manifest discovery, shard and placement reads, and read-only proofs only when the request's volume is resolved from authoritative stored state. A shard the relay holds but cannot tie to a volume, and a proof request not tied to a volume, MUST be refused. A write grant permits shard and placement uploads; a read-write grant permits both. A grant-scoped token MUST NOT authorize replication, staging, finalization, manifest publication, deletion, or administrative operations at a relay. On chain, the grant row itself is commit authority when the grantee's account sends the transaction: a whole-volume, non-`tee_required`, unexpired grant (with its DEK package on a private volume) lets a `ReadWrite` grantee stage, commit directly and finalize, and a `WriteOnly` grantee stage (CIP-9 §7.7). Volume-owner actions, including deposits, transfer and allowlist changes, still require owner authorization. Other uploaded data becomes visible only when a full-volume holder publishes a manifest that references it.

For a private volume, direct access requires the grantee's separate X25519 encryption key and an HPKE-wrapped DEK bound to that key and the current `dek_version`, following CIP-9 §9.2.1. The encryption key and package MUST be supplied together, and the package MUST satisfy CIP-9's exact size and binding checks; public-volume grants MUST omit both. A private-volume write grantee holds the whole DEK and can decrypt the volume. `WriteOnly` limits which operations the grantee is authorized to perform; it places no limit on what that key can decrypt.

A grantee opens a volume by explicit owner and volume name. The response MUST supply the grantee's DEK package, never the owner's wrap. Owner-wide enumeration still lists only owned volumes. Access ends within the cache freshness bound after the grant is removed, the delegation is revoked, the grant expires or ownership is transferred. Transfer MUST clear the allowlist. Revocation does not erase plaintext or DEKs already obtained, and issued tokens remain bearer credentials until invalidated or expired.

10.5 Access modes

Mode	Read	Write	List	Delete
`ReadOnly`	Yes	No	Yes	No

Mode	Read	Write	List	Delete
`WriteOnly`	No	Yes	No	No
`ReadWrite`	Yes	Yes	Yes	No

Runners never receive `Delete`. It is an owner-only control-plane operation.

10.6 Scoping and concurrency

Runner CapTokens are scoped by **volume**, **path prefix**, **byte quota**, and **time window**. CIP-9 §7.3.1 defines three enforcement boundaries:

1. At issuance, the Dispatcher MUST reject a write prefix that overlaps an already-valid write token on the same volume. Read-only tokens may overlap. Prefixes use the canonical attachment form: `""` for the whole volume, otherwise no leading slash and one trailing slash.
2. Every commit MUST name the `prev_root` it extends, and the chain rejects a commit whose `prev_root` differs from the current root. A commit carries the new root and its shard changes but not the changed paths, so the chain cannot check prefix confinement. It therefore rejects a direct commit from a writer scoped to a non-empty prefix, in any access mode and on either visibility. Such a writer only stages its delta: it records its manifest change on chain as pending, without publishing a new root. A full-volume holder (the owner, an owner delegation, or an empty-prefix grant or CapToken) confirms the delta stays inside the prefix and finalizes it, which publishes the root; on a private volume that check needs the DEK. A job whose CapToken covers the whole volume publishes its own writes. When such a job stages, the stage records finalize authority for that job, and the Runner that staged it may finalize it while the job's write access remains valid (CIP-9 §7.3.1).
3. At shard transfer, Relay Nodes cannot infer object paths from opaque shard IDs. They check token validity, access mode and byte quota, and they cannot prove from a shard ID that a write stays inside its path. Quota and orphan garbage collection bound out-of-prefix uploads. Reaching a relay gives an upload no authority to publish a manifest.

Direct-client grants follow Direct-Client Grants: an eligible whole-volume `ReadWrite` grantee may stage, commit directly and finalize, and a `WriteOnly` grantee may only stage.

© Cowboy Labs, Inc. 2026

11 Volume lifecycle

This section covers the on-chain (control-plane) steps of a volume's life. The data-plane steps are in Writing and Reading Data.

11.1 Create (control plane)

The owner submits a `create_volume` transaction to the Storage Manager (`0x0A`) with `volume_name`, `erasure_k`, `erasure_m`, `visibility`, and an initial CBY reserve. The actor:

1. Derives `volume_id = keccak256(owner_address || utf8_bytes(volume_name))`. If the volume already exists, it reverts with `VolumeAlreadyExists`.
2. Debits the refundable creation bond, `VOLUME_CREATION_DEPOSIT_WEI` = 1 CBY (governed at `ras.volume_creation_deposit_wei`), from the owner in addition to the initial reserve, and holds it in the volume's escrow. The bond pays no rent, gas or withdrawal and returns to the owner's storage reserve with the remaining escrow when the volume is garbage-collected.
3. For `PRIVATE` volumes, the client MUST generate the DEK off-chain and wrap it to the current CBSS threshold-IBE committee key. The request MUST include non-zero `cbss_committee_epoch` and `cbss_release_key_material_hash`; the Storage Manager MUST validate them against Secrets Manager (`0x04`) before recording the wrapped key. Plaintext DEK material never enters consensus.
4. Writes a `StorageCommitment` record with status `ACTIVE`, empty `manifest_root`, `effective_size_bytes = 0`.
5. Records the volume's `relay_nodes`: an explicit list of exactly K+M distinct registered relays, or, when none is given, the first K+M `ACTIVE` relays with free capacity from the Relay Registry active list. Creation fails when fewer exist.

11.2 Attach (control plane)

When a job is dispatched with volume attachments, the Storage Manager mints a runner CapToken per attachment. For private volumes, Secrets Manager (`0x04`) coordinates release of the DEK sealed directly to the runner's ephemeral X25519 key. The Dispatcher carries only sealed material and never sees plaintext. The runner decrypts the sealed DEK when the job starts and zeroes it on completion.

11.3 Delete, grace and restore

The owner MAY soft-delete a volume via `delete_volume`. The StorageCommitment transitions to `DELETED` for `VOLUME_DELETE_GRACE_EPOCHS = 1` storage epoch (genesis parameter `storage_epoch_blocks`, 86,400 blocks or ~24 h on production genesis), during which:

- All active CapTokens are revoked.
- New CapTokens cannot be issued.
- Storage rent continues to accrue.
- The owner MAY call `undelete_volume` to restore status to `ACTIVE`.

If the grace window expires without restoration, the volume transitions to `GARBAGE_COLLECTING`. Relay Nodes learn of the transition through heartbeat acknowledgments or direct RPCs and purge that terminal volume's shards asynchronously, and storage rent stops. The same terminal state is reached when per-epoch billing detects a balance shortfall (see Settlement).

The chain then cleans up the volume's inventory. Relays sign acknowledgements (ACKs) of the volume's events, and the chain prunes events once they are acknowledged. After that acknowledged event history is pruned, the native GC lane, a separate cleanup step that runs in every block with a bounded amount of work, removes the volume's inventory rows. Each item removes one stripe row and its dense stripe ordinal (the stripe's number in a gap-free numbering of the volume's stripes) and decrements the live shard-ref count by `w`. The lane records its progress durably and shares its work among eligible terminal volumes. It keeps the `StorageCommitment` while rows remain, and removes the volume from the live index only after `live_shard_count` reaches zero and the event-prune gate, the check that the volume's acknowledged event history has been pruned, is satisfied. Healthy native-row cleanup runs independently of daily rent epochs. Faults in committed data are isolated within the EpochSweep phase, the block phase after transactions that processes bounded settlement and cleanup work, under CIP-31's native inventory GC rules, which specify quarantine, stalls, retained frozen state and recovery limits. Both PoR sampling builders exclude `GARBAGE_COLLECTING` volumes before drawing, even while native inventory rows remain.

The current event-prune pass stops at 1,024 acknowledged events per volume per daily storage epoch. A large history can therefore delay the start of native GC for many epochs, and more than 1,024 new events per day can grow the backlog without limit. Three open problems follow:

- A separate, durable, fair per-block event-prune queue is required. It cannot by itself bound the backlog, because valid manifest commits can create event-prune work faster than the current sweep's shared 32-item-per-block ceiling.
- Indefinite production operation also requires a deterministic bound on outstanding prune work, enforced by refusing or slowing new commits when the bound is reached. It also

requires prune capacity reserved when a commit is staged, so an accepted staged commit is never blocked at finalize solely because pruning is at capacity, and automatic client retry when a commit is temporarily refused.

- A permanently unavailable relay that was assigned when an event happened blocks pruning. The chain still requires that relay's signed event ACK before pruning, even after eviction, assignment rotation or terminal volume status. A safe way to prune without the ACK of such a lost relay is also required.

The queue, admission, retry and lost-relay recovery changes are proposed. The native inventory and its GC lane do not implement them.

12 Billing, rent, and eviction

Storage rent charges continuously for the Relay Node capacity a volume occupies, and any account can fund it.

12.1 Fee components

Fee	When Charged	Amount
Volume creation bond	`create_volume`	`VOLUME_CREATION_DEPOSIT_WEI = 1 CBY`, refundable at garbage collection
Storage rent	Per epoch	Rounded-up MiB of effective size at the storage rate
Read transfer	Bounded funded-channel checkpoint	Newly acknowledged logical-shard MiB units × execution-time rate, within signed caps; genesis rate 66,700 nano-CBY/MiB. Writes are outside this read mechanism.

Storage rent and read-transfer charges are separate from the Cycles, Accesses and Cells execution fee markets specified in Technical Whitepaper §4 and §17. RAS adds no on-chain meter of its own. On-chain operations consume Cycles, Accesses and Cells as usual: creating StorageCommitments,

writing manifest roots, posting revocations, and opening, claiming or refunding read channels. The Storage Manager keeps the per-epoch storage rent in a separate ledger, charged to the volume's escrow and then the owner's account storage reserve. Dedicated read-channel principal is another separate ledger. Transaction gas is paid separately from both.

CBFS commits charge an upper bound of 1 Cell per logical persisted key+value byte for all state growth, plus full rewrites of rows sized by the transaction inputs. This includes inventory, rent, event, snapshot, funding and deferred writes. Direct and staged paths pay the same application Cells for the same delta and settlement context. Staging a commit separately reserves full stage-row writes and global-index growth. Only the in-place remainder of StorageCommitment, the global stage index and existing PoR rows is covered by the flat base charge, as specified in CIP-43 §5.5.

12.2 Relay earnings

Relay Nodes earn from two sources:

1. Relays share the per-epoch storage rent. The total per-epoch rent on a volume is split among the platform, the PoR challenge pool and relays:

   ```
   billable_mib = ceil(effective_size / 2^20)
   total_storage_fee = billable_mib
                       × STORAGE_FEE_PER_MIB_PER_EPOCH
   relay_pool = total_storage_fee
                × STORAGE_FEE_RELAY_BPS / 10,000
   relay_weight_i = Σ_{p ∈ counted_i} min(age_p,
                         MAX_SHARD_AGE_FOR_WEIGHTING)
   ```

   The relay pool is distributed in proportion to `relay_weight_i` under CIP-31 §5. `counted_i` contains the unique `(shard_id, shard_index)` pairs served by relay `i` with counted valid PoR responses in the prior epoch. `age_p` is each pair's assignment age at settlement, capped separately at 90 epochs. Multiplying the count by a capped age gives the same result only when all the ages are equal.

2. Relays earn payer-authorized read transfer fees. A fixed serving relay claims newly acknowledged logical-shard MiB units from a dedicated funded channel, within cumulative payer caps. Governance controls the execution-time rate. `TRANSFER_FEE_PER_MIB = 66,700 nano-CBY` is the genesis rate (see Parameters); small reads at that rate may not cover the gas to open, checkpoint and refund a channel. No unsolicited relay report debits a reader's account.

© Cowboy Labs, Inc. 2026

`STORAGE_FEE_PLATFORM_BPS = 10%` of gross storage rent is credited to the Platform Fee Account (`0x18`, CIP-31 §4); `STORAGE_FEE_CHALLENGE_POOL_BPS = 1%` funds PoR challenges; and `STORAGE_FEE_RELAY_BPS = 89%` goes to relays in proportion to each relay's per-pair capped-age weight (CIP-31 §5).

12.3 Settlement

At each epoch boundary, the Storage Manager runs `settle_volume_epoch()`:

1. Fund the volume's escrow toward the elapsed rent plus `E` epochs ahead, where `E` is `max(min_storage_balance_epochs, auto top-up target)`. Funding draws from the owner's account storage reserve, then (with auto top-up on) from the owner's spendable balance (CIP-9 §10.3).
2. Charge the elapsed rent from escrow, then from the reserve. The first unsettled epoch costs `ceil(epoch_peak_used_bytes / 2^20) × STORAGE_FEE_PER_MIB_PER_EPOCH`, where `epoch_peak_used_bytes` is the largest `used_bytes` since the last settlement. Each later epoch costs the same formula over the current `used_bytes`. The fee is the one in the fee snapshot the sweep recorded when it queued the epoch. Split it 10% to the Platform Fee Account, 1% to the challenge pool, and 89% to relays in proportion to each relay's per-pair capped-age weight (CIP-31 §5).
3. If escrow plus reserve is below the volume's rent floor, mark the volume as entering `GRACE_PERIOD`, with reads permitted and writes denied. The rent floor is `ceil(used_bytes / 2^20) × floor_rate_per_mib_epoch × floor_epochs`, using the rate and epoch count stamped on the volume, never the live rate (CIP-31 §3).
4. A volume already in `GRACE_PERIOD` moves to `GARBAGE_COLLECTING`, and holding relays get a purge signal, only when escrow plus reserve cannot pay the elapsed rent and at least `STORAGE_GRACE_EPOCHS = 1` epoch (`storage_epoch_blocks`, 86,400 blocks or ≈24 h at 1 s/block on production genesis) remains unpaid. A settlement that pays all elapsed rent keeps the volume in `GRACE_PERIOD` while escrow plus reserve stays below the rent floor, and returns it to `ACTIVE` once they reach the floor.

Transfer principal comes from payer-funded cumulative read channels. Relay Node attestations alone cannot charge a payer, and nothing debits a Runner or Actor account implicitly. A direct payer-signed opening reserves a fixed deposit for a fixed relay wallet and node and a fixed volume. Each complete logical shard adds `ceil(bytes / 2^20)` units only after the payer verifies the independently trusted manifest length and hash and durably acknowledges it. The signed request to serve bytes gives no spending authority, and payment never replaces the separate data capability.

© Cowboy Labs, Inc. 2026

Claims for one relay and epoch are emitted in atomic pages of 1..32 sorted unique channels, with one latest cumulative payer ACK per selected channel. At execution, a claim pays `(cumulative_units - settled_units) × current_TRANSFER_FEE_PER_MIB`, bounded by the fixed maximum rate, the remaining signed cumulative fee ceiling, and dedicated escrow. The fixed relay receives 100% of that principal, and the transaction submitter cannot redirect it. Sequence numbers and counters are consumed once, even at zero price. One invalid entry rejects the entire page. Changed registry ownership, volume status, or expired eligibility for new reads cannot revoke an existing funded claim before its claim deadline.

The channel's terms fix its read-stop height and its claim-deadline height. Strictly after the claim deadline, a gas-paying submitter may return the remaining principal, and only to the fixed original payer. The channel record stays as a permanent closed record (a tombstone), so its network, payer and funding-nonce identity cannot be reused. `escrow_remaining + paid + refunded = original_deposit` must always hold. Rent escrow, storage reward pools and challenge funds do not fund these reads.

Opening, checkpoint and refund gas is paid on top of the principal. Its Cycle, Cell and Access components all debit the same native CBY balance. Four mechanisms stop a restart or retry from creating fresh spending authority: one-shot logical data requests, durable reservations of the full deposit and opening gas, a bound on bytes delivered but not yet acknowledged by a signed ACK, and immutable lifetime budgets. Exact signed transactions must be journaled before submission. After an unknown outcome, only an exact-byte resubmission is allowed; a new nonce or a repacked page cannot stand in for finality. Releasing a pending transaction lane requires trusted finalized evidence of the exact transaction and of its committed nonce consumption. Saved signed ACKs may be resent without rereading data or signing again. Bytes the payer verified but had not yet signed an ACK for when it stopped require explicit recovery. Unverified lost bytes cannot be acknowledged.

Claiming a small remaining balance may cost more in gas than it pays, even with batching. An evaluation of a channel's whole lifetime cost counts opening, multiple checkpoint partitions, charged failures, refunds, locked capital and delivery risk. Each settlement runs under the three explicit operator thresholds in CIP-9 §10.4.2 (minimum revenue, maximum gas share and maximum gas per settlement), which must be set before signing. The CIP-31 §2.1 measurement discussion separates actual execution measurements from production quotes, and approves no default gas-share threshold or lock window.

This consensus rule is active from genesis. Operators opt in to the operator services. The capability is `funded_payer_cumulative_read_channel_v1`, with RAS kinds 28/29/30. Kind 27, its signatures and its journals keep their own meaning and are not reinterpreted. Full wire, authorization and recovery requirements are in CIP-9 §10.4.1–§10.4.2. Enabling the operator

© Cowboy Labs, Inc. 2026

service does not move existing pools. Disabling it must preserve the claim and refund paths of channels already funded.

The mounted SDK keeps one immutable, one-shot selection per object read. Its payment journal does not keep payloads, so a second paid read does not repair cache loss or an incomplete response. There is no free fallback, automatic relay replacement, journal reset or guarantee of fair exchange for the last shard. State and finality JSON from a trusted node is an observation. It is neither a cryptographic state proof nor a relay's evidence of delivery.

12.4 Comparison with state rent

Storage rent follows the same pattern as the state rent system in CIP-4 §12: unpaid rent, then eviction, then restoration. The details differ:

Property	State rent (on-chain state)	Storage rent (CBFS volumes)
Target resource	QMDB key/value state	Relay Node shard capacity
Billed unit	Stored key/value bytes, at least 1 KiB per actor	Bytes of `effective_size` (post-erasure)
Window before eviction	Unpaid debt above 30 rent-epochs of rent	1 storage epoch (~24 h)
Eviction	Storage pruned, code + root preserved	Shards purged; the `StorageCommitment` remains as a terminal volume-ID tombstone after GC
Restoration	Anyone may restore with original data + back-rent + 10% catch-up premium	Owner may `undelete_volume` during grace; not after GC

13 On-chain control plane

Two system actors hold RAS's on-chain state: the **Storage Manager** at `0x0A` and the **Relay Registry** at `0x0B`. Both are registered in the system-actor table (Technical Whitepaper §9). Their state and instructions are below.

© Cowboy Labs, Inc. 2026

13.1 Storage Manager (0x0A)

The Storage Manager is the on-chain authority for volume ownership, CapToken issuance, revocation and billing. Its state is:

- `ras:volume:{volume_id}`

  `StorageCommitment` (owner, manifest root, size, erasure K/M, relays, status).

- `ras:owner-name:{owner_address}:{keccak256(volume_name)}`

  `volume_id` (index from name to ID).

- `ras:token:{token_id}`

  `CapToken` (runner tokens, stored for validation).

- `ras:relay-assignments:{volume_id}`

  List of assigned relay addresses.

- `ras:account-summary:{owner_address}`

  `AccountStorageSummary` (total volumes, effective bytes, reserved balance).

- `ras:volume-billing:{volume_id}`

  `VolumeBillingState` (last_billed_epoch, delete_requested_epoch, accrued_fees).

- `ras:revocation:{wallet_address}:{cbfs_public_key}`

  `DelegationRevocation` (timestamp).

The `StorageCommitment` records volume identity, ownership, roots, erasure settings, billing, visibility, access grants and key-release bindings. CIP-9 §11.1 defines its wire schema. The fields are:

```
volume_id:            bytes32     // keccak256(owner || volume_name)
owner:                Address
volume_name:          String      // UTF-8, ≤64 bytes
visibility:           PRIVATE | PUBLIC
created_at:           u64         // block height
wrapped_dek:          bytes       // wrapped DEK; empty for PUBLIC
manifest_root:        bytes32     // manifest DAG root
live_shard_count:    u64         // live shard-ref count; w refs per stripe row
mount_allowed_actors: MountGrant[]
dek_version:          u32         // binds direct-client key packages
```

```
raw_size_bytes:       u64
effective_size_bytes: u64         // raw × (K+M)/K
erasure_k:            u8
erasure_m:            u8
last_updated:         u64         // block height of last manifest commit
degraded_shards:      u16
status:               PENDING_MPK_SETUP | MPK_SETUP_FAILED | ACTIVE | GRACE_PERIOD | DELETED
```

The authoritative live shard set is in the native RAS consensus inventory, outside this serialized commitment. One stripe row keyed by `(volume_id, shard_id)` holds all `w = K+M` per-index `chunk_root` values, the common `shard_size`, the publication event sequence and a dense stripe ordinal. An ordinal row identifies the stripe for bounded selection and paging, and removal keeps stripe ordinals dense. The shards RPC expands stripe ordinal `c` and index `i` into the computed shard number `c·w+i`, so the RPC keeps its request and response shape. `live_shard_count` and `MAX_LIVE_SHARDS_PER_VOLUME = 1,048,576` count shard refs, with `w` refs per stripe, and `MAX_SHARDS_PER_COMMIT = 4,096` caps combined added and removed refs across a complete commit. Direct commits and each Stage page have a stricter 512-ref cap. The commit must also be able to fund its Access reservation, which is a further limit (CIP-9 §11.1).

Rent evidence stores one history row per stripe inventory incarnation, with per-index assignment data. An incarnation is one stay of a stripe in the inventory, from the event that adds it until its removal. Its stripe-wide `generation` is the stripe's publication event sequence (`RasStripeRow.added_at_event_seq`), which PoR challenges carry as `snapshot_added_at_event_seq`. There is no separate `ras:rent:v1:assignment:` row. The `ras:rent:v1:history:<volume><shard><incarnation_be>` key uses that sequence as a big-endian u64. Other rent keys, including `rrp:`, `rrw:` and `rrf:`, keep their formats. Rent rows use strict big-endian binary encoding. Identifiers implied by keys, stripe headers or index positions are omitted and reconstructed on decode (CIP-9 §11.1). The `/fees/cbfs-rent/epochs/{epoch}` endpoint keeps its JSON response shape and its `assignment_generation` field name, and the field's value identifies the inventory incarnation event sequence.

The Storage Manager exposes `/ras/*` RPC endpoints on Cowboy validator nodes, including:

Endpoint	Purpose
`GET /ras/auth/me`	Verify delegation, return account summary
`POST /ras/auth/revoke`	Revoke a DelegationCert
`GET /ras/accounts/balance`	Check `balance_reserved` and current usage

Endpoint	Purpose
`POST /ras/volumes`	Register a new volume
`GET /ras/volumes`	List owned volumes
`GET /ras/volumes/{id}`	Volume details, relay endpoints, status
`GET /ras/volumes/{name}/open`	Open metadata; cross-owner clients specify `?owner={address}` and require a matching direct-client grant
`POST /ras/volumes/{id}/delete`	Soft-delete (enter grace)
`POST /ras/volumes/{id}/undelete`	Cancel soft-delete within grace

Owner control-plane mutations are client-signed transactions carrying the canonical owner authorization in CIP-9 §12.2.1. HTTP delegation envelopes use `X-Cbfs-Delegation`, `X-Cbfs-Timestamp`, `X-Cbfs-Nonce` and `X-Cbfs-Signature`; where this envelope is used, timestamps MUST be within ±30 seconds and replay checks MUST reject reused nonces. A data-plane grant cannot substitute for owner control-plane authorization.

13.2 Relay Registry (`0x0B`)

The Relay Registry manages Relay Node membership, health and settlement:

Key	Value
`ras:relay:{relay_address}`	`RelayNodeProfile` (by address)
`ras:relay-node:{node_id}`	`RelayNodeProfile` (by node_id)
`ras:active-relays:0`, `ras:active-relays:1`	Active relay addresses, two fixed pages of raw 20-byte addresses (2,048 per page)
`ras:settlement:{relay_address}`	`RelaySettlementState` (claimable rewards)

Relay Nodes submit `RegisterRelay`, `UpdateRelay`, `SetRelayStatus`, `SettleOwedShards` and the PoR and drain instructions to this actor. There are no heartbeat, capacity-advertisement, shard-report or settlement-claim calls. It is also the target of PoR challenge resolution and slashing.

© Cowboy Labs, Inc. 2026

13.3 Cache freshness

Relays read owner authority (the volume's commitment with its grants, the delegation and its revocation) together, from one finalized-state proof whose block is at most **10 seconds** old, and reuse a read for at most 250 ms. Authorization MUST fail closed when required commitment, delegation, revocation, or grant state is stale or unavailable. A revoked DelegationCert, a removed grant or an ownership transfer therefore takes effect at relays within about 10 s of the block that finalizes it. Relay profiles are cached for **300 seconds**, and the peer-drain authorization path refreshes them every 30 seconds. A relay missing from a local cache is fetched synchronously on first use, so a newly registered relay does not cause a latency spike.

14 State transition function (storage subset)

This section specifies the storage-relevant part of the protocol state transition function defined in the Technical Whitepaper (see *The State Transition Function*). Within each block:

1. The chain processes RAS system instructions. Volume creation, deletion, undelete, commit manifest, revoke delegation, and settlement apply to Storage Manager state; relay registration, status changes, drain, and settlement apply to Relay Registry state.
2. The chain issues runner CapTokens. For each runner job assigned in the block with volume attachments, it derives and persists the runner CapToken and sealed DEK copies.
3. The chain applies PoR instructions in the block. `RespondPorChallenge` verifies a submitted proof. There is no automatic settlement when a window closes: an unanswered challenge settles only when someone submits `SettlePorChallenge` after the response window and grace, which applies the miss and any slashing.
4. At each epoch boundary (every `rent_epoch_length`), the chain iterates all active volumes, computes storage rent, transitions statuses (ACTIVE to GRACE_PERIOD to GARBAGE_COLLECTING, with restoration to ACTIVE permitted from GRACE_PERIOD), and distributes relay earnings.

Storage mutations pay execution fees under the Technical Whitepaper. Per-epoch rent is accounted for separately as described in Billing, Rent, and Eviction.

© Cowboy Labs, Inc. 2026

15 Security considerations

15.1 Client-side encryption

For private volumes, Relay Nodes store ciphertext and see structural metadata such as sizes and object counts. Decryption requires the DEK, which is delivered to authorized runners or direct clients under the applicable key-release contract. Public-volume objects and manifests are plaintext.

15.2 Shard ID opacity

`shard_id = BLAKE3("cbfs.object-chunk.v1" || volume_id || write_id || u32_le(chunk_index))` is a one-way hash. Relay Nodes cannot learn object paths, detect overwrites, or enumerate a volume from shard IDs alone. This protects privacy, and it also means relays can do little to enforce path scope at the shard-transfer boundary (boundary 3 in Scoping and Concurrency); see also Direct-Client Grants.

15.3 Durability and scoped writes

The Erasure Coding section defines the reconstruction threshold, and Repair defines recovery. The Scoping and Concurrency section separates the authority to upload shards from the authority to publish a manifest. Prefix enforcement for a private volume depends on the DEK-holding finalizer checking the staged delta before publication.

15.4 Relay eclipse and withholding

An object stays retrievable while at least K of its assigned shards are available. Losing more than M shards of a stripe, counting owed shards not yet repaired, prevents reconstruction. Withholding means a relay keeps shards but does not serve them. PoR challenges detect silent withholding across the full relay set on a rolling basis. Readers detect a substituted or stale manifest by comparing its root with the on-chain `manifest_root`. Withholding prevents retrieval, and a root commitment alone does not establish availability.

15.5 Sybil resistance

Relay registration requires a non-trivial `MIN_RELAY_STAKE`. Auto-assignment takes the first `ACTIVE` relays with free capacity from the registry's active list, so a sybil attacker must stake for every relay it registers, and each sybil faces its own PoR slashing risk.

© Cowboy Labs, Inc. 2026

15.6 Key compromise

An attacker who compromises the cold wallet key gains account authority, including the ability to authorize storage access. Confidentiality therefore depends on protecting both wallet authority and DEK material. If the hot CBFS key is lost, the owner revokes the DelegationCert on-chain (effective at relays within about 10 s of finalizing) and issues a new one.

15.7 Key-delivery boundary

Runner key delivery follows Attach; cross-owner client key delivery follows Direct-Client Grants. Dispatcher MUST NOT receive a plaintext DEK. Revocation ends authorized access and cannot retract key material already delivered.

15.8 Replay and nonce reuse

DelegationCert signatures commit to `expires_at_ms`, `chain_id` and `aud`, and to the network through the `ChainIdentity` preimage. Owner-token signatures commit to the version byte and the full payload. Control-plane requests carry `X-Cbfs-Nonce` (LRU replay cache, 60 s TTL). Peer-relay RPCs carry domain-tagged digests with a 60 s timestamp window.

16 Parameters (genesis defaults)

Storage pricing follows CIP-31. Billing and PoR economic values, minimum relay stake, and auto-drain thresholds are governance-tunable. Approximate block durations below assume one second per block.

Group	Parameter and default
Volumes	`MAX_VOLUMES_PER_OWNER` = 1,024
Volumes	`MAX_VOLUME_SIZE` = 200 GiB stored (at least 100 GiB of content when M $\leq$ K)
Volumes	`MAX_OBJECTS_PER_VOLUME` = 1,000,000
Volumes	`MAX_OBJECT_SIZE` = 1 GiB; atomic mutation depends on chunk tier, manifest depth, repartitioning and pending-epoch headroom (CIP-9 §14)

© Cowboy Labs, Inc. 2026

Group	Parameter and default
Volumes	`MAX_VOLUME_NAME_LENGTH` = 64 bytes
Volumes	`MAX_OBJECT_PATH_LENGTH` = 512 bytes
Erasure coding	`DEFAULT_K` = 4
Erasure coding	`DEFAULT_M` = 2
Erasure coding	`MIN_K` = 2 / `MAX_K` = 16
Erasure coding	`MIN_M` = 1 / `MAX_M` = 8
CapTokens	`MIN_VOLUME_CAP_TOKEN_BLOCKS` = 600,000 (~7 days)
CapTokens	owner read/put TTL ≤ 900 s (`OWNER_TOKEN_OPEN_TTL_SECS`)
CapTokens	owner mount TTL ≤ 86,400 s / 24 h (`OWNER_TOKEN_MOUNT_TTL_SECS`)
CapTokens	owner-token clock skew ±30 s (`OWNER_TOKEN_CLOCK_SKEW_MS`)
CapTokens	DelegationCert expiry ≤ 30 days
CapTokens	`MAX_CAP_TOKEN_BYTES` = 16 KiB
Relay Nodes	`MIN_RELAY_STAKE` = 33,000 CBY (governance-tunable)
Relay Nodes	relay status = `ACTIVE`, `DRAINING`, `DISABLED`, `INACTIVE`
Relay Nodes	auto-drain thresholds = governance-tunable
Relay Nodes	`RELAY_UNSTAKE_DELAY_EPOCHS` = 1 storage epoch (~24 h)
Relay Nodes	`REPAIR_CHECK_INTERVAL_SECS` = 300 seconds
Relay Nodes	`ORPHAN_SHARD_TTL_SECS` = 86,400 seconds (~24 h)
Proof of Retrievability	`POR_CHALLENGE_INTERVAL` = 7,200 blocks (~2 h at 1 s/block)
Proof of Retrievability	`POR_RESPONSE_WINDOW` = 50 blocks (~50 s)

© Cowboy Labs, Inc. 2026

Group	Parameter and default
Proof of Retrievability	`POR_MISS_PENALTY` = 330 CBY
Proof of Retrievability	`POR_FRAUD_PENALTY` = 3,300 CBY
Proof of Retrievability	`RELAY_EVICTION_PENALTY` = 13,200 CBY
Proof of Retrievability	`RELAY_CHALLENGE_BOND` = 66 CBY
Proof of Retrievability	`POR_CHALLENGE_FEE` = 7 CBY
Proof of Retrievability	`CHALLENGER_BOUNTY` = 33 CBY
Proof of Retrievability	`MAX_SHARD_AGE_FOR_WEIGHTING` = 90 epochs
Billing	`VOLUME_CREATION_DEPOSIT_WEI` = 1 CBY, refundable creation bond
Billing	`STORAGE_FEE_PER_MIB_PER_EPOCH` = 2,905 nano-CBY
Billing	`TRANSFER_FEE_PER_MIB` = 66,700 nano-CBY
Billing	Rent floor = `min_storage_balance_epochs` (genesis 1) epochs of fees at the rate stamped on the volume
Billing	fee split = 10% platform / 1% challenge pool / 89% relays
Billing	`STORAGE_GRACE_EPOCHS` = 1 storage epoch
Billing	`VOLUME_DELETE_GRACE_EPOCHS` = 1 storage epoch
Billing	`storage_epoch_blocks` = 86,400 blocks (~24 h), genesis parameter fixed for the chain's life
Filesystem mount	`DEFAULT_SYNC_INTERVAL` = 5 s
Filesystem mount	`MIN_SYNC_INTERVAL` = 1 s
Filesystem mount	`MAX_LOCAL_CACHE_SIZE` = 10 GiB
Cache freshness (relays)	owner-authority proof age $\leq$ 10 s
Cache freshness (relays)	relay profile cache = 300 s

© Cowboy Labs, Inc. 2026

17 Relationship to the Technical Whitepaper

This document builds on the Technical Whitepaper and does not override it:

- Actors, messages, timers, reentrancy and per-handler resource limits follow Technical Whitepaper §3.
- Fee markets (Cycles, Accesses, Cells) and per-block basefee adjustment follow Technical Whitepaper §4 and §17. Storage operations that touch the chain consume these meters as usual.
- State rent for **on-chain actor storage** follows Technical Whitepaper §4.4 and §17.5. Storage rent in this document applies only to **off-chain CBFS volumes**.
- The Runner Marketplace (Technical Whitepaper §5) consumes runner CapTokens. Job dispatch issues volume attachments through the Storage Manager.
- Consensus, randomness and networking follow Technical Whitepaper §6. Relay Nodes use the consensus randomness beacon for PoR challenge targeting.
- Governance, upgrades and hot-code replacement of system actors follow Technical Whitepaper §11.
- Entitlements (Technical Whitepaper §15) MAY gate storage capabilities at the runner and actor manifest level (for example, whether an actor may open a volume at all). RAS CapTokens are a second, finer layer within an entitlement's allowed scope.

Where this document and the Technical Whitepaper disagree on values or procedures, the Technical Whitepaper is the normative reference.

End of specification.

© Cowboy Labs, Inc. 2026