

Cowboy Storage: CBFS and Runner Attached Storage

Technical Whitepaper

Charles DePue, Martin Aceto

September 2026 — v0.10.1

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 substrate operated by a permissionless network of **Relay Nodes**. RAS is the Cowboy protocol layer that mounts CBFS under account-scoped **Volumes**, anchors volume state on-chain via compact **StorageCommitments**, and mediates access to volumes through cryptographic **capability tokens** (CapTokens).

CBFS transfers object data off-chain without a transaction per object access. RAS anchors volume ownership, manifest publication, billing, challenges, and eviction on-chain. Private volumes encrypt their contents before distribution to Relay Nodes. A **delegation keypair** model separates cold wallet authority from hot storage authority, allowing owner-direct data operations to be authorized locally and verified offline by Relay Nodes against cached chain state.

This specification covers the volume lifecycle, CapToken and delegation cryptography, Reed-Solomon erasure coding, Relay Node registration and repair, Proof-of-Retrievability challenges, per-epoch storage rent, and storage parameters. It is a complement to the [Cowboy Technical Whitepaper](#), which is the normative reference for consensus, execution, economics, and actor semantics. The [Relationship to the Technical Whitepaper](#) section defines the governing references.

2 Introduction

Actors on Cowboy are autonomous Python programs with on-chain storage subject to state rent and a 1 MiB default quota (extendable to 8 MiB via a storage bond) — see the Technical Whitepaper §3.2, §4.4. That model is sized for state, not for data. Modern autonomous agents need to persist working memory, cache retrieval traces, hold model adapters, stage artifacts between jobs, and serve large public assets. None of those fit naturally into on-chain state: they are too large, too churn-heavy, or too privacy-sensitive to push into a consensus-critical key/value store.

Cowboy separates consensus state from bulk storage:

- **State** — small, consensus-critical key/value data that participates directly in the state transition function. Billed per cell, subject to rent, stored in QMDB, and bounded by per-actor quotas.
- **Storage** — large, potentially private data, durably kept across a permissionless Relay Node network and referenced on-chain only by compact commitments. Billed per byte per epoch in a separate market, mediated by capability tokens rather than transactions.

CBFS is the storage substrate. It is designed to be usable standalone (as a plain distributed filesystem) and as the data plane for Cowboy, with the same binaries and wire protocol in both configurations. RAS is the control plane: it defines the rules under which Cowboy accounts own volumes, grant runners access, settle storage rent, and register and slash Relay Nodes.

3 Architectural Overview

This overview introduces the components. Requirements and their rationale appear together in the sections that follow.

This document uses **MUST/SHOULD/MAY** as defined in RFC 2119. Parameters marked **governance-tunable** can be changed via the on-chain governance mechanism specified in Technical Whitepaper §11. In all cases, 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

- **Volume:** A named, account-scoped storage namespace. Each volume is a private (encrypted) or public (plaintext) container of objects identified by path keys.
- **Object:** A single blob within a volume, identified by its path. Encrypted client-side for private volumes; plaintext for public ones.
- **Relay Node:** A network participant that stores erasure-coded shards of volume data, answers reads, participates in repair, and serves Proof-of-Retrievability challenges. A distinct role from Validators and Runners.
- **Shard:** A fragment of an erasure-coded object. An object is split into K data + M parity shards; any K reconstruct the ciphertext.
- **Manifest:** The authoritative, Merkle-committed index of a volume's contents. Encrypted for private volumes, plaintext for public ones, stored as content-addressed DAG nodes on Relay Nodes.
- **StorageCommitment:** The on-chain record of a volume's existence, owner, manifest root, erasure parameters, size, billing state, and status.
- **PlacementRecord:** The off-chain, mutable record binding a shard ID to its assigned Relay Nodes. Replicated across assigned nodes; versioned with compare-and-swap semantics.
- **CapToken:** A bearer credential authorizing volume operations. Runner tokens are issued by the chain at job dispatch; owner-auth tokens are signed by a delegated client key and

admitted through volume ownership or an explicit direct-client grant. Scope and validation depend on the authorization path.

- **DelegationCert:** A wallet-signed certificate that binds an Ed25519 CBFS key to a Cowboy account, scoped to storage use only, with a short (30-day) expiry.
- **Storage Manager (0x0A):** The Cowboy system actor that maintains StorageCommitments, mints runner CapTokens, handles per-epoch billing, and governs volume status transitions.
- **Relay Registry (0x0B):** The Cowboy system actor that maintains Relay Node profiles, staking, health, and repair/challenge coordination.

3.2 Design Rationale

Actor workloads need mutable working state, private data, shared artifacts, and filesystem access across short-lived runners. CBFS addresses these requirements through:

- **Off-chain object transfers:** clients exchange shards directly with Relay Nodes; publication and lifecycle changes use the chain.
- **Scoped authorization:** job attachments bind storage access to a runner; owners and explicitly authorized clients use locally signed tokens.
- **Committed manifests:** an on-chain root identifies the authoritative volume contents served by relays.
- **Repair without decryption:** erasure shards and placement metadata let relays repair private data without holding the DEK.
- **Filesystem access:** FUSE lets runner workloads use ordinary file operations against persistent volumes.

The following sections specify these mechanisms and their limits.

3.3 Differences vs. On-chain State

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

Aspect	On-chain State	CBFS / RAS
Durability	Consensus replication across all validators	Reed-Solomon K+M across Relay Nodes
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 fixed **visibility** set at creation and never changed:

- **PRIVATE:** Objects are AES-256-GCM encrypted under a per-volume DEK. The manifest is encrypted. Reads and writes require CapTokens.
- **PUBLIC:** Objects are plaintext. The manifest is plaintext. Reads are open to any party without a CapToken; writes still require one.

An account MUST NOT exceed `MAX_VOLUMES_PER_ACCOUNT`. Each volume MUST NOT exceed `MAX_OBJECTS_PER_VOLUME = 1,000,000` objects or `MAX_VOLUME_SIZE = 100 GiB`; each object MUST NOT exceed `MAX_OBJECT_SIZE = 1 GiB`. Object paths are UTF-8 strings of up to `MAX_OBJECT_PATH_LENGTH = 512` bytes. The full addressing scheme for an object is:

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

5 The CBFS Data Plane

CBFS is the substrate that stores and serves volume data. Its implementation is divided into the following crates:

Crate	Responsibility
<code>cbfs-types</code>	Shared protocol types: <code>VolumeId</code> , <code>ShardId</code> , <code>NodeId</code> , <code>PlacementRecord</code> , <code>ObjectDescriptor</code> , <code>RpcRequest/Response</code> , <code>Operation</code> .
<code>cbfs-crypto</code>	AES-256-GCM, BLAKE3, DEK generation and envelope wrapping.
<code>cbfs-erasure</code>	Reed-Solomon over $GF(2^8)$ via <code>reed-solomon-erasure</code> v6 with SIMD.
<code>cbfs-transport</code>	QUIC client/server (<code>quinn</code>) with TLS 1.3 and bincode-framed RPCs.
<code>cbfs-store</code>	Sled-backed blob store on Relay Nodes.
<code>cbfs-manifest</code>	Sorted client-side volume index and canonical content-chunked Merkle DAG.
<code>cbfs-placement</code>	Persistent <code>PlacementStore</code> for shard-to-node assignments with CAS and replica merge.
<code>cbfs-hooks</code>	Pluggable traits (<code>AuthProvider</code> , <code>ManifestRegistry</code> , <code>NodeSelector</code> , <code>MeteringSink</code>) with standalone and Cowboy-mode implementations.
<code>cbfs-auth</code>	Cowboy-facing auth runtime: delegation loading, request signing, owner token minting, relay and chain-state caches.
<code>cbfs-sdk</code>	High-level Volume API: create, open, put, get, commit, with local manifest and placement synchronization.
<code>cbfs-fuse</code>	POSIX FUSE mount (inode table, write-back cache, sync daemon, extended attributes).
<code>cbfs-node</code>	The Relay Node daemon: QUIC server, garbage collection, heartbeats, repair, drain.
<code>cbfs-cli</code>	Operator CLI: <code>login</code> , <code>auth</code> , <code>volume</code> , <code>put</code> , <code>get</code> , <code>mount</code> , <code>commit</code> , <code>rebalance</code> .

5.1 Object Representation

Each object is described by an **ObjectDescriptor** stored in the manifest:

Field	Type	Meaning
<code>object_path</code>	<code>String</code>	≤ 512 bytes, UTF-8
<code>write_id</code>	<code>[u8; 16]</code>	Fresh CSPRNG per write, isolates versions
<code>content_hash</code>	<code>[u8; 32]</code>	BLAKE3 of plaintext
<code>ciphertext_hash</code>	<code>[u8; 32]</code>	BLAKE3 of ciphertext before erasure coding
<code>encryption_nonce</code>	<code>[u8; 12]</code>	Fresh random per write (AES-GCM invariant)
<code>size_bytes</code>	<code>u64</code>	Plaintext size
<code>ciphertext_size</code>	<code>u64</code>	Pre-padding ciphertext size (required for erasure truncation)
<code>shard_id</code>	<code>[u8; 32]</code>	<code>BLAKE3(volume_id object_path write_id)</code> — opaque to Relay Nodes
<code>erasure_k, erasure_m</code>	<code>u8, u8</code>	Per-object erasure parameters
<code>shard_hashes</code>	<code>[[u8; 32]; K+M]</code>	BLAKE3 of each shard, used for integrity and PoR

The opacity of `shard_id` — Relay Nodes see a hash, not a path — preserves privacy: they cannot learn which object a shard belongs to, nor detect overwrites. The [Scoping and Concurrency](#) section explains the resulting authorization limits.

Each write MUST use a fresh CSPRNG-generated `write_id` and record `content_hash = BLAKE3(plaintext)`. Private objects MUST also record `ciphertext_hash = BLAKE3(ciphertext)` and the size and padding metadata needed for reconstruction without decryption.

5.2 Encryption

A private volume's DEK MUST be 32 bytes drawn from a CSPRNG. Each private object MUST be encrypted 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. The AAD MUST bind the ciphertext to its descriptor's `volume_id`, `object_path`, and `write_id`; the authentication tag MUST be appended to the ciphertext per RFC 5116. Readers MUST verify `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 (`cbfs/manifest/src/dag.rs`): 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; `StorageCommitment.manifest_root` is the top node's locator.

Private-volume nodes are individually encrypted before storage, so the root commits to ciphertext. Public-volume nodes are plaintext. Unchanged nodes retain their locators across commits; clients publish only newly added nodes, addressed by `manifest_node_shard_id(locator)`. Readers MUST verify nodes against their expected locators. The canonical encoding, chunk boundaries, and encryption domains are specified in CIP-9 §3.1. See [Commit and Publication](#) for ordering and authority.

5.4 Placement

Shards are assigned to K+M distinct Relay Nodes via a **PlacementRecord**:

Field	Type	Meaning
<code>shard_id</code>	[u8; 32]	Opaque shard identifier
<code>version</code>	u64	CAS version for atomic updates during repair
<code>assignments</code>	[{shard_index: u8, node_id: [u8;32]}]	Node per shard index

Field	Type	Meaning
<code>erasure_k, erasure_m</code>	<code>u8, u8</code>	Duplicated from descriptor for repair without manifest
<code>ciphertext_size</code>	<code>u64</code>	Duplicated to enable reconstruction without decryption
<code>shard_hashes</code>	<code>[[u8; 32]; K+M]</code>	Duplicated for integrity checks during repair

PlacementRecords MUST remain mutable, versioned records separate from the manifest. They MUST duplicate the descriptor's erasure parameters, ciphertext size, and shard hashes 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/update.
- `GetPlacement(shard_id)` — fetch.
- `ReplicatePlacement(shard_id, record)` — peer-to-peer replication, signed by the sending relay's identity key.

This externalization of placement — as opposed to embedding assignments in the (encrypted) manifest — is what allows repair workers to operate on private volumes without decryption rights.

Node selection is pluggable via the `NodeSelector` trait. The selector scores eligible relays by available capacity, geographic and operator diversity, health, reputation, and observed latency, then chooses $K+M$ distinct destinations.

5.5 Erasure Coding

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

For the default 4/6 scheme, an object is split into 6 shards distributed across 6 Relay Nodes; any 4 reconstruct the ciphertext, so the system tolerates any 2 simultaneous Relay Node failures per object. Accounts billed for `effective_size = raw_size * (K+M)/K`.

Implementations MUST support $2 \leq K \leq \text{MAX_ERASURE_K} = 16$ and $1 \leq M \leq \text{MAX_ERASURE_M} = 8$. Accounts MAY choose higher redundancy at creation time.

Ciphertext **MUST** be padded to a multiple of K before splitting; readers **MUST** truncate reconstruction to the recorded `ciphertext_size`. Each shard **MUST** be hashed with BLAKE3, with the hash stored in both the `ObjectDescriptor` and `PlacementRecord`. 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_SHARD_TRANSFER_SIZE + 64 KiB` (default ~1 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> }
```

Implementations **MUST** support the following `Operation` values: `PutShard`, `GetShard`, `DeleteShard`, `GetPlacement`, `PutPlacement`, `ReplicatePlacement`, `Ping`, `RelayHandshake`, `SyncPeers`, `PeerRelayPushShard`, `PeerPlacementUpdate`. Responses carry a matching `request_id` and a status code (`Ok`, `ErrAuth`, `ErrNotFound`, `ErrCapacity`, `ErrInvalid`, `ErrConflict`).

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:" || node_id || client_nonce || timestamp_ms_be)` with the relay's Ed25519 identity key. Clients **MUST** verify against the cached on-chain `relay_identity_pubkey` and reject timestamps outside ± 30 seconds.

5.7 FUSE Mount

CBFS ships a FUSE filesystem (`cbfs-fuse`) that exposes a volume as a POSIX directory with write-back caching. Writes land instantly in an in-memory dirty cache; a background sync daemon encrypts, shards, and pushes 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
<code>cbfs.status</code>	<code>committed</code> / <code>dirty</code> / <code>syncing</code>
<code>cbfs.encrypted</code>	<code>aes-256-gcm</code>
<code>cbfs.erasure</code>	K+M (e.g., 4+2)
<code>cbfs.shards</code>	<code>shard_index@node_addr,...</code>
<code>cbfs.content_hash</code>	First 8 bytes of BLAKE3, hex

Cowboy-mode mounts refresh owner tokens before expiry while preserving the requested access mode, allowing long-lived sessions without weakening token scope.

6 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](#).

6.1 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
<code>version</code>	<code>u8</code>	0
<code>token_id</code>	<code>[u8; 16]</code>	Deterministically derived
<code>token_secret</code>	<code>bytes</code>	Entropy; validation requires secret match
<code>volume_id</code>	<code>[u8; 32]</code>	Target volume
<code>volume_name</code>	<code>String</code>	For logging, not validation
<code>runner_address</code>	<code>Address</code>	Restricts to a single runner

Field	Type	Meaning
<code>access_mode</code>	<code>AccessMode</code>	<code>ReadOnly</code> , <code>WriteOnly</code> , <code>ReadWrite</code>
<code>path_prefix</code>	<code>String</code>	Canonical attachment prefix (e.g., <code>workspace/</code>); <code>""</code> means whole volume
<code>max_bytes</code>	<code>u64</code>	Per-token quota
<code>valid_from</code> , <code>valid_until</code>	<code>u64</code> , <code>u64</code>	Block heights
<code>mount</code> , <code>sync_interval</code>	<code>bool</code> , <code>u32</code>	Mount parameters

The `token_id` and `token_secret` are derived deterministically from job metadata, runner address, block height, attachment index, and VRF seed, preventing pre-image attacks. At runtime, a Relay Node validates a presented runner `CapToken` by looking up the stored copy at `StorageManager[token_id]` and checking byte-for-byte match plus TTL, runner address, volume status, and access mode. Prefix restrictions are enforced at attachment admission and checked by clients/coordinators; opaque shard RPCs do not expose paths for relay-side containment checks.

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

6.2 Owner CapTokens (client-signed)

An **Owner CapToken** (`OwnerCapTokenV1`) is signed locally by a delegated CBFS key. The signer MUST be admitted either as the volume owner or through the [Direct-Client Grants](#) contract; a delegation cert alone conveys no cross-owner volume authority.

Field	Type	Meaning
<code>version</code>	<code>u8</code>	1
<code>volume_id</code>	<code>[u8; 32]</code>	Target volume
<code>volume_name</code>	<code>String</code>	For logging, not validation

Field	Type	Meaning
<code>owner_address</code>	<code>Address</code>	Volume owner; MUST match the authoritative commitment owner
<code>access_mode</code>	<code>AccessMode</code>	<code>ReadOnly</code> , <code>WriteOnly</code> , <code>ReadWrite</code>
<code>path_prefix</code>	<code>String</code>	Requested scope; MUST be empty for a grant-scoped token
<code>valid_from_ms</code> , <code>valid_until_ms</code>	<code>u64</code> , <code>u64</code>	Unix milliseconds (wall-clock, short TTL)
<code>mount</code> , <code>sync_interval</code>	<code>bool</code> , <code>u32</code>	Mount parameters
<code>delegation_cert</code>	<code>DelegationCert</code>	Embedded wallet delegation cert
<code>owner_signature</code>	<code>bytes</code>	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 while preserving 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`, `network`, `aud` == "cowboy:cbfs", and `scope` == "cbfs". 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. RAS wire types and signing-byte helpers are owned by `cowboy-protocol-codec`; clients and validators MUST use that canonical encoding, as specified in CIP-9 §12.2.1.

6.3 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, network, audience, and expiry. The signing client registers the delegation on-chain; 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. Canonical fields and signing bytes are defined by `cowboy-protocol-codec` and consumed by the CBFS authorization layer.

The delegated CBFS key is stored encrypted at `~/.cbfs/cbfs_key.enc` under AES-256-GCM with a user-supplied passphrase (or environment variable for headless use), decrypted on demand into locked memory, and zeroed after signing. `cbfs auth rotate` replaces the key; `cbfs auth revoke` posts revocation to the Storage Manager. [Cache Freshness](#) bounds propagation to relays.

6.4 Direct-Client Grants

CIP-9 §7.7 and §7.7.1 define a single on-chain mount allowlist for cross-owner access. A grant without `grantee_signing_key` authorizes job dispatch only. A direct-client grant additionally 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, while `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. Unattributed shards and unscoped proofs MUST be refused. A write grant

permits shard and placement uploads; a read-write grant permits both. Grants **MUST NOT** authorize replication, staging, finalization, manifest publication, deletion, or administrative operations. Volume-owner actions, including deposits, transfer, and allowlist changes, still require owner authorization. Uploaded data becomes visible only when the owner publishes a manifest referencing 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 holding the whole DEK has decryption capability: `WriteOnly` limits authorized operations, not what that key can decrypt.

A grantee opens 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. Removing the grant, revoking the delegation, expiry, or ownership transfer ends access within the cache freshness bound. Transfer **MUST** clear the allowlist. Revocation does not erase plaintext or DEKs already obtained, and issued tokens remain bearer credentials until invalidated or expired.

6.5 Access Modes

Mode	Read	Write	List	Delete
<code>ReadOnly</code>	Yes	No	Yes	No
<code>WriteOnly</code>	No	Yes	No	No
<code>ReadWrite</code>	Yes	Yes	Yes	No

`Delete` is never granted to runners; it is an owner-only control-plane operation.

6.6 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. **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. **Commit authority:** every commit MUST name the `prev_root` it extends; the chain rejects a mismatch with the current root. CIP-9 requires public-volume write-only commits to prove prefix confinement against the plaintext manifest delta. For private volumes, a write-only sub-agent stages its delta; a DEK-holding coordinator or owner verifies confinement before finalizing the root. The sub-agent cannot unilaterally publish it.
3. **Shard transfer:** Relay Nodes cannot infer object paths from opaque shard IDs. They check token validity, access mode, and byte quota, but cannot prove a write's path containment from its shard ID. Out-of-prefix uploads remain bounded by quota and orphan garbage collection; they do not acquire manifest publication authority merely by reaching a relay.

Direct-client grants follow the separate whole-volume and owner-publication restrictions in [Direct-Client Grants](#).

7 Volume Lifecycle

This section specifies the end-to-end lifecycle of a volume, distinguishing control-plane (on-chain) and data-plane (CBFS) steps.

7.1 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, revert with `VolumeAlreadyExists`.
2. Charges `VOLUME_CREATION_FEE = 1,000 CBY` from the owner's account.
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. Assigns initial Relay Nodes via the [Relay Registry](#).

7.2 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. Dispatcher carries only sealed material and never sees plaintext. The runner decrypts the sealed DEK on job start and zeroes it on completion.

7.3 Write (object transfer)

An authorized runner or client:

1. Generates a fresh 16-byte `write_id` and computes `shard_id = BLAKE3(volume_id || object_path || write_id)`.
2. For a private volume, encrypts the object under the [Encryption](#) rules; public-volume objects remain plaintext.
3. Reed-Solomon encodes the stored object bytes (ciphertext for private volumes, plaintext for public volumes) to $K+M$ shards, padding to a multiple of K .
4. Selects $K+M$ Relay Nodes from the cached Relay Registry and issues `PutShard(shard_id, shard_index, shard_bytes, metadata)` RPCs in parallel, each authenticated by the CapToken.
5. Records the new `ObjectDescriptor` in the local manifest and constructs the `PlacementRecord`.

Every step is local or off-chain. No transaction is submitted.

7.4 Read (object transfer)

1. Fetch the manifest root from the authoritative Storage Manager state (or a cached, freshness-bounded value).
2. Fetch and verify the manifest DAG from its committed root, using each node's locator to address its stored shards; decrypt private-volume nodes with the DEK.
3. Resolve `object_path` to an `ObjectDescriptor`; fetch its `PlacementRecord` via `GetPlacement`.
4. Request any K of $K+M$ shards in parallel (prefer low-latency, fail over on timeout or corruption).
5. Verify shard hashes, reconstruct the stored bytes via Reed-Solomon, and truncate padding to the recorded size. For private volumes, verify `ciphertext_hash` before decryption.
6. For private volumes, decrypt; for all volumes, verify the plaintext `content_hash`.

7.5 Commit and Publication

Manifest publication follows CIP-9 §12.2:

1. **Publish data:** store the manifest DAG's added nodes under their content-addressed locators and publish the corresponding PlacementRecords. Nodes reachable from the previous root **MUST** remain intact while that root is authoritative.
2. **Commit authority:** submit the new root, `prev_root`, and added/removed shard deltas through the authorized control-plane path. The chain **MUST** compare `prev_root` with the current root before advancing `StorageCommitment.manifest_root`; shard deltas update `shard_root` for PoR. Owner control-plane writes use the write-relayer transport specified in CIP-9 §12.2.1. Scoped runner commits follow [Scoping and Concurrency](#); direct-client grantees cannot publish a root.

The required order is to publish new nodes, then advance the on-chain root. If the chain commit fails, the previous root and its nodes remain usable. Implementations **MUST** reclaim nodes made obsolete by a successful commit only after the root advances; uploads left unreferenced by a failed commit are reclaimed through orphan GC.

The authoritative manifest is the one matching the root last accepted by the chain (`READ_COMMITTED`). Readers **MUST** verify that match. Root integrity and availability are separate: a committed root cannot supply bytes when too few relays serve its shards.

7.6 Delete, Grace, Restore

The owner **MAY** soft-delete a volume via `delete_volume`. The `StorageCommitment` transitions to `DELETED` for `VOLUME_DELETE_GRACE_EPOCHS = 1` storage epoch (`STORAGE_EPOCH_BLOCKS = 86,400`, ~24 h), 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 via heartbeat acknowledgments or direct RPCs, purge their shards asynchronously, and storage rent ceases. The same terminal state is reached when per-epoch billing detects a balance shortfall (see [Settlement](#)).

8 Relay Nodes

A Relay Node is a dedicated storage participant distinct from validators and runners. It persists erasure-coded shards, serves reads and writes to authorized clients, participates 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 `register_relay(capacity_bytes)` transaction to the Relay Registry (0x0B) with a stake of at least `MIN_RELAY_STAKE = 33,000 CBY` (governance-tunable) and publishing its Ed25519 `relay_identity_pubkey`. The registration produces a **RelayNodeProfile**:

Field	Type	Meaning
<code>address</code>	<code>[u8; 32]</code>	Relay Node address
<code>stake_amount</code>	<code>u256</code>	Self-bonded CBY
<code>capacity_bytes</code>	<code>u64</code>	Advertised capacity
<code>used_bytes</code>	<code>u64</code>	Consensus-derived committed shard storage
<code>status</code>	<code>RelayStatus</code>	<code>ACTIVE</code> , <code>DRAINING</code> , <code>DISABLED</code> , or <code>INACTIVE</code>
<code>updated_epoch</code>	<code>u64</code>	Last profile mutation and heartbeat freshness
<code>shards_held</code>	<code>u32</code>	Active shard count
<code>shards_lost</code>	<code>u32</code>	Historical losses (reputation)
<code>region_hint</code>	<code>[u8; 4]</code>	Geographic hint for placement diversity

Relay Nodes **MUST** refresh `updated_epoch` via `heartbeat()`. Placement considers only `ACTIVE` relays whose heartbeat remains fresh under the governance-controlled auto-drain policy. A stale heartbeat makes a relay eligible for auto-drain; `DRAINING`, `DISABLED`, and `INACTIVE` relays receive no new assignments.

A Relay Node **MAY** unstake after a `RELAY_UNSTAKE_DELAY = 86,400 blocks` (~24 h) cooldown starting after full drain completion (see [Drain and Peer-Push](#)).

8.2 Repair

Relay Nodes run a recurring repair loop every `REPAIR_CHECK_INTERVAL_SECS = 300` seconds:

8.2.1 Repair Local Shards

For each `PlacementRecord` where this node is an assignee, verify the local shard against its expected `shard_hash`. On missing or corrupt:

1. Fetch `K` healthy shards from peers listed in the `PlacementRecord`.
2. Reconstruct the ciphertext via Reed-Solomon, truncating to `ciphertext_size` (this step is why `ciphertext_size` is duplicated into the `PlacementRecord` — without it, padding would corrupt the reconstruction).
3. Re-encode, extract the expected shard, verify its hash, persist locally.

Self-heal is safe to run concurrently on any number of nodes: it only writes to local storage, never to the `PlacementRecord`.

8.2.2 Replace Unavailable Relays

For each `PlacementRecord`, probe all assignees. If a node is unreachable past a timeout:

1. **Leader election.** Among the live assignees, the node with the **lowest** `shard_index` is the unique leader for this `PlacementRecord`. Ties cannot occur (indices are distinct).
2. The leader selects a replacement Relay Node from the peer list (excluding current assignees) using the `NodeSelector`.
3. The leader reconstructs the missing shard(s) from surviving shards and pushes them to the replacement via `PutShard` or the signed `PeerRelayPushShard` RPC.
4. The leader writes a new `PlacementRecord` with the updated assignment and `version + 1` via a `CAS PutPlacement(shard_id, record, expected_version)`. If CAS fails (because another node replaced a different dead node in the same record concurrently), the leader retries in the next cycle.
5. Updated `PlacementRecord` is replicated to all current assignees via `ReplicatePlacement`.

Repair requires at least `K` healthy shards. Under the default `K=4, M=2` scheme, three permanent shard losses before repair completes prevent reconstruction. 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 voluntarily exit (or be commanded to exit by governance) by transitioning to **Draining** status. A drain worker iterates assigned shards and peer-pushes each via the 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)`

Both are signed by the sending relay's identity key over domain-tagged BLAKE3 digests (`cbfs:peer_push_shard:v1`, `cbfs:peer_placement_update:v1`) and reject messages whose timestamp is outside a 60 s window. On successful drain, the node writes a signed `RelaySelfDisableRequest` when `shards_remaining == 0` and exits the active set.

9 Proof of Retrievability

Because Relay Nodes are economically incentivized to silently discard shards (to claim rent without serving), RAS implements periodic **Proof of Retrievability (PoR)** challenges. Challenges are cheap to verify, expensive to fake, and cost Relay Nodes measurable stake when they fail.

9.1 Challenge Protocol

A challenge is triggered by an on-chain timer every `POR_CHALLENGE_INTERVAL = 7,200 blocks` (~2 h at 1 s/block). It selects a random `(shard_id, shard_index, byte_offset, byte_length)` tuple targeting an active relay. The relay **MUST** respond within `POR_RESPONSE_WINDOW = 600 blocks` (~10 min at 1 s/block) with:

- The requested byte range (ciphertext bytes) from the named shard.
- A two-level Merkle proof chain:
 1. `byte_range_proof` — folds the challenged chunk of ciphertext up to the shard's `chunk_root` (`cbfs_eraser::chunk_root`).
 2. `shard_inclusion_proof` — folds the shard leaf `BLAKE3(0x00 || shard_id || shard_index || chunk_root)` up to the `StorageCommitment`'s `shard_root` accumulator (a keyed sparse Merkle root over all ciphertext shard commitments; `cowboy_ras::shard_root`).

The on-chain verifier:

1. Recomputes the chunk leaves from the submitted ciphertext bytes and verifies `byte_range_proof` resolves to the shard's `chunk_root`.
2. Verifies `shard_inclusion_proof` resolves to `challenge.snapshot_shard_root`, snapshotted from `StorageCommitment.shard_root` at challenge issuance.

PoR anchors on the `StorageCommitment.shard_root`, a sparse Merkle root over ciphertext shard commitments. The challenge snapshots this root at issuance. Every verification input — `snapshot_shard_root`, `shard_id`, `shard_index`, `chunk_root`, and the relay's ciphertext bytes — is public. Because data is encrypted before erasure coding and hashing, shard commitments are DEK-independent, so PoR for a private volume is verifiable on-chain without the DEK.

A relay that stored the shard answers from local disk. PoR proves the shard is *retrievable at challenge time*, backed by stake and repair — not that the relay stores it continuously: within the response window a relay could reconstruct the challenged range from its erasure peers, so PoR deters silent discard economically (a caught relay is slashed and its shards are re-replicated by repair) rather than proving dedicated storage cryptographically. Commitment updates during the response window are handled by evaluating proofs against the `shard_root` snapshotted at challenge issuance; shards removed by a legitimate write during the window void the challenge without penalty.

9.2 Outcomes

Outcome	Consequence
Valid response within window	No action
No response	<code>shards_lost++</code> , shard flagged for repair, <code>POR_MISS_PENALTY</code> slashed
Invalid proof	<code>shards_lost++</code> , shard flagged for repair, <code>POR_FRAUD_PENALTY</code> (> miss) slashed
3+ consecutive failures	Removed from active set, all shards flagged for repair, <code>RELAY_EVICTION_PENALTY</code> slashed

A governance-tunable 1% share of storage rent plus `POR_CHALLENGE_FEE` funds challenge issuance; challengers (who may be any on-chain actor triggering challenges via timers — see Technical Whitepaper §3.3) receive a finder's fee from slashed stake.

10 Billing, Rent, and Eviction

Storage rent is CBFS's primary economic mechanism: it creates a continuous cost for occupying Relay Node capacity, directly fundable by any account.

10.1 Fee Components

Fee	When Charged	Amount
Volume creation	<code>create_volume</code>	<code>VOLUME_CREATION_FEE</code> = 1,000 CBY
Attachment	<code>submit_task</code> with volume attachment	<code>BASE_ATTACHMENT_FEE</code> = 100 CBY per volume per job
Storage rent	Per epoch	Rounded-up MiB of effective size at the storage rate
Transfer	Per served MiB	Rounded-up MiB served at the transfer rate

Storage rent and transfer charges are separate from the Cycles, Accesses, and Cells execution fee markets specified in the Technical Whitepaper §4 and §17. On-chain operations (creating StorageCommitments, writing manifest roots, posting revocations) consume Cycles, Accesses and Cells normally. The per-epoch storage rent is a separate ledger maintained by the Storage Manager, debited from a dedicated `balance_reserved` on the account.

10.2 Relay Earnings

Relay Nodes earn from two streams:

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

```
billable_mib = ceil(effective_size / 220)
total_storage_fee = billable_mib
                    × STORAGE_FEE_PER_MIB_PER_EPOCH
relay_pool = total_storage_fee
             × STORAGE_FEE_RELAY_BPS / 10,000
age_i = min(shard_age_in_epochs_i,
```

```

MAX_SHARD_AGE_FOR_WEIGHTING)
relay_weight_i = shard_count_i * age_i

```

The relay pool is distributed in proportion to `relay_weight_i` under CIP-31 §5.

2. **Metered transfer fees.** Each serving relay receives `ceil(bytes_served / 220) × TRANSFER_FEE_PER_MIB` from the reader’s account at read time. The rate is governance-tunable; see [Parameters](#).

`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% flows to relays, weighted by shard count and shard age. Relays with high `shards_lost` values receive fewer assignments over time via the `NodeSelector` reputation score.

10.3 Settlement

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

1. Compute `epoch_fee = ceil(effective_size_bytes / 220) × STORAGE_FEE_PER_MIB_PER_EPOCH`.
2. If `balance_reserved ≥ epoch_fee`: charge, split 10% to the Platform Fee Account, 1% to the challenge pool, and 89% to relays weighted by shard count and shard age, then increment `last_billed_epoch`.
3. If `balance_reserved < epoch_fee`: mark the volume as entering `GRACE_PERIOD`, with reads permitted and writes denied.
4. If the volume has been in `GRACE_PERIOD` for `STORAGE_GRACE_EPOCHS = 1` epoch (`STORAGE_EPOCH_BLOCKS = 86,400` blocks, ≈ 24 h at 1 s/block) without top-up, transition to `GARBAGE_COLLECTING` and broadcast a purge signal to holding relays.

Transfer fees are settled per read via Relay Node attestations, batched and submitted as a single transaction per relay per epoch to minimize on-chain overhead.

10.4 Comparison with State Rent

The mechanism intentionally mirrors the state rent system specified in Technical Whitepaper §4.4 and §17.5 — grace period, then warning, then eviction — but differs in kind:

Property	State rent (on-chain state)	Storage rent (CBFS volumes)
Target resource	QMDB key/value state	Relay Node shard capacity

Property	State rent (on-chain state)	Storage rent (CBFS volumes)
Billed unit	Cells (bytes of consensus state)	Bytes of <code>effective_size</code> (post-erasure)
Grace period	7 rent-epochs	1 storage epoch (~24 h)
Eviction	Storage pruned, code + root preserved	Shards purged, <code>StorageCommitment</code> retained until GC completes
Restoration	Anyone may restore with original data + back-rent	Owner may <code>undelete_volume</code> during grace; not after GC

11 On-Chain Control Plane

Two system actors anchor RAS on Cowboy: the **Storage Manager** at 0x0A and the **Relay Registry** at 0x0B (both registered in the system-actor table — Technical Whitepaper §9). Their state and instruction surfaces are detailed below.

11.1 Storage Manager (0x0A)

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

- `ras:volume:{volume_id}`
StorageCommitment (owner, manifest root, size, erasure K/M, relays, status).
- `ras:owner-name:{owner_address}:{keccak256(volume_name)}`
volume_id (name→id index).
- `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:challenge:{challenge_id}:{wallet_address}`
ChallengeRecord (single-use nonce for high-impact ops, 30 s TTL).
- `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. Its wire schema is defined by CIP-9 §11.1; the following is a field summary:

```

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
wrapping_key_hash: bytes32
manifest_root:  bytes32      // manifest DAG root
shard_root:     bytes32      // PoR shard commitment root
mount_allowed_actors: MountGrant[]
dek_version:    u64          // 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:         ACTIVE | GRACE_PERIOD | DELETED | GARBAGE_COLLECTING

```

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

Endpoint	Purpose
POST <code>/ras/challenge</code>	Mint a single-use challenge (TTL 30 s) for high-impact mutations

Endpoint	Purpose
GET /ras/auth/me	Verify delegation, return account summary
POST /ras/auth/revoke	Revoke a DelegationCert
GET /ras/accounts/balance	Check <code>balance_reserved</code> and current usage
POST /ras/volumes	Register a new volume (requires challenge)
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 <code>?owner={address}</code> 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 require the canonical owner authorization and write-relayer transport in CIP-9 §12.2.1. HTTP delegation envelopes use `X-Cbfs-Delegation`, `X-Cbfs-Timestamp`, `X-Cbfs-Nonce` or `X-Cbfs-Challenge-Id`, and `X-Cbfs-Signature`; where this envelope is used, timestamps **MUST** be within ± 30 seconds and replay checks **MUST** reject reused nonces or challenges. A data-plane grant cannot substitute for owner control-plane authorization.

11.2 Relay Registry (0x0B)

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

Key	Value
<code>ras:relay:{relay_address}</code>	<code>RelayNodeProfile</code> (by address)
<code>ras:relay-node:{node_id}</code>	<code>RelayNodeProfile</code> (by node_id)
<code>ras:active-relays</code>	List of active relay addresses
<code>ras:settlement:{relay_address}</code>	<code>RelaySettlementState</code> (claimable rewards)

Relay Nodes perform `heartbeat()`, `advertise_capacity()`, `report_shards()`, and `claim_settlement()` against this actor. It is also the target of PoR challenge resolution and slashing.

11.3 Cache Freshness

Relay Nodes maintain local caches of Storage Manager and Relay Registry state — ownership, delegations, grants, revocations, relay profiles — to avoid a chain round-trip on every CapToken validation. Caches refresh every **15 seconds** with a maximum staleness bound of **60 seconds**. Authorization **MUST** fail closed when required commitment, delegation, revocation, or grant state is stale or unavailable. A revoked DelegationCert is therefore effective across the network within 60 s of the on-chain posting. Relays missing from a local cache are fetched synchronously on first use, preventing a latency cliff for newly registered relays.

12 State Transition Function (Storage Subset)

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

1. **Process RAS system instructions.** Volume creation, deletion, undelete, commit manifest, revoke delegation, and settlement apply to Storage Manager state; relay registration, heartbeat, drain, and settlement apply to Relay Registry state.
2. **Issue runner CapTokens.** For each runner job assigned in the block with volume attachments, derive and persist the runner CapToken and sealed DEK copies.
3. **Resolve PoR challenges.** For challenges whose response window closes this block, verify submitted proofs and apply slashing and reputation updates.
4. **Epoch boundary (every `rent_epoch_length`).** Iterate all active volumes, compute storage rent, transition statuses (ACTIVE → GRACE_PERIOD → GARBAGE_COLLECTING, with restoration to ACTIVE permitted from GRACE_PERIOD), and distribute 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](#).

13 Security Considerations

13.1 Client-side encryption

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

13.2 Shard ID opacity

`shard_id = BLAKE3(volume_id || object_path || write_id)` is a one-way hash. Relay Nodes cannot learn object paths, detect overwrites, or enumerate a volume from shard IDs alone. This enables privacy but forces Layer-3 path enforcement to be weak; see [Scoping and Concurrency](#) and [Direct-Client Grants](#).

13.3 Durability and Scoped Writes

The [Erasure Coding](#) section defines the reconstruction threshold, and [Repair](#) defines recovery. The [Scoping and Concurrency](#) section separates shard upload authority from manifest publication authority. Private-volume prefix enforcement depends on the DEK-holding finalizer checking the staged delta before publication.

13.4 Relay eclipse and withholding

An object remains retrievable while at least K of its assigned shards are available; exceeding the erasure budget prevents reconstruction. 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 instead prevents retrieval; a root commitment alone does not establish availability.

13.5 Sybil resistance

Relay registration requires a non-trivial `MIN_RELAY_STAKE`, and `NodeSelector` rewards diverse, healthy, reputationally clean relays. A sybil attacker must stake proportionally to the fraction of placements desired, and each sybil faces independent PoR slashing risk.

13.6 Key compromise

Compromise of the cold wallet key gives the attacker account authority, including the ability to authorize storage access. Confidentiality therefore depends on protecting both wallet authority and DEK material. Loss of the hot CBFS key requires revoking the `DelegationCert` on-chain (effective within the 60 s cache bound) and issuing a new one.

13.7 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 but cannot retract key material already delivered.

13.8 Replay and nonce reuse

DelegationCert signatures commit to `expires_at_ms`, `chain_id`, `network`, and `aud`; owner-token signatures commit to the version byte and full payload. Control-plane requests carry `X-Cbfs-Nonce` (LRU replay cache, 60 s TTL) or `X-Cbfs-Challenge-Id` (single-use, 30 s TTL). Peer-relay RPCs carry domain-tagged digests with a 60 s timestamp window.

14 Parameters (Genesis Defaults)

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

Group	Parameter and default
Volumes	<code>MAX_VOLUMES_PER_ACCOUNT</code> = 256
Volumes	<code>MAX_VOLUME_SIZE</code> = 100 GiB
Volumes	<code>MAX_OBJECTS_PER_VOLUME</code> = 1,000,000
Volumes	<code>MAX_OBJECT_SIZE</code> = 1 GiB
Volumes	<code>MAX_VOLUME_NAME_LENGTH</code> = 64 bytes
Volumes	<code>MAX_OBJECT_PATH_LENGTH</code> = 512 bytes
Erasure coding	<code>DEFAULT_ERASURE_K</code> = 4
Erasure coding	<code>DEFAULT_ERASURE_M</code> = 2
Erasure coding	<code>MAX_ERASURE_K</code> = 16
Erasure coding	<code>MAX_ERASURE_M</code> = 8
CapTokens	<code>MIN_VOLUME_CAP_TOKEN_BLOCKS</code> = 600,000 (~7 days)

Group	Parameter and default
CapTokens	owner read/put TTL ≤ 900 s (OWNER_TOKEN_OPEN_TTL_SECS)
CapTokens	owner mount TTL $\leq 86,400$ 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	heartbeat freshness and auto-drain thresholds = governance-tunable
Relay Nodes	RELAY_UNSTAKE_DELAY = 86,400 blocks (~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 = 600 blocks (~10 min)
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

Group	Parameter and default
Billing	VOLUME_CREATION_FEE = 1,000 CBY
Billing	BASE_ATTACHMENT_FEE = 100 CBY
Billing	STORAGE_FEE_PER_MIB_PER_EPOCH = 2,905 nano-CBY
Billing	TRANSFER_FEE_PER_MIB = 66,700 nano-CBY
Billing	MIN_STORAGE_BALANCE = one epoch of fees at the current rate
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)
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)	Refresh interval = 15 s
Cache freshness (relays)	max staleness = 60 s

15 Relationship to the Technical Whitepaper

This document layers on, but never supersedes, the Technical Whitepaper. Specifically:

- 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 normally.
- 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) is the consumer of 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 system actor hot-code replacement follow Technical Whitepaper §11.
- Entitlements (Technical Whitepaper §15) MAY gate storage capabilities at the runner and actor manifest level (for example, whether an actor is permitted to open a volume at all); RAS CapTokens are the second, fine-grained 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.