The Decentralized File Storage Protocol

Availability-Audited Capsule Hosting on DIG Layer 2

Michael Taylor

Version 1.0 (Full Specification) · May 2026

Abstract

The Decentralized File Storage Protocol (DFSP) is the storage layer of DIG Layer 2. It stores full-replica capsules and pays for audited availability, running on the Chia-anchored proof-of-stake consensus of Part 1 and serving the content-addressable WASM store of Part 3. A capsule is a Digstore store fixed at 100 megabytes, padded with deterministic noise so every capsule is identical in size and, through Digstore's blinding, indistinguishable at rest. An operator mirrors a capsule by posting a collateral bond and holding the whole module; redundancy is full replication, not erasure coding, so any one healthy mirror serves a request complete.

Availability is the reward event, and the audit is the whole enforcement mechanism. Each block's proposer, the single validator that Part 1 selects for that slot, audits a seed-forced sample of mirrors and proposes which of them that block's storage subsidy pays. The rest of the validator set does not re-probe: it recomputes the selection from the seed, re-verifies the cryptographic proofs the proposer collected, and finalizes the result under Part 1's ordinary attestation. One auditor per block, never the whole set at once, so the audit cannot become a denial-of-service attack on the mirrors. A mirror that passes is paid for the block. A mirror that fails is simply not paid, and is not slashed for a single miss; instead a decaying failure weight, a leaky bucket that ages isolated misses to nothing, slashes its bond in full only when sustained failure crosses a threshold. A separate, immediate reader fraud proof slashes a mirror that serves verifiably wrong or stale content, proven on a signed response, and is structurally impossible to fake against an honest mirror.

`$DIG` is minted on Part 1's perpetual block schedule and split across five reward roles; the storage role-share is the entire hosting emission and its own cap, so per-mirror reward dilutes automatically as the network grows and total hosting issuance stays bounded with no separate ceiling to maintain. Against that emission the protocol burns at every consumption point. The architecture draws one hard line: collateral and a capsule's update authority are coins, with puzzle-enforced exits, and everything else, the registries, the accrued reward balances, and the failure weights, is layer-2 state finalized through the checkpoint. This document specifies the full system: the capsule and its blinding, the coin and state model, the token, the mirror and its bond, the proposer audit and its seed, the reward and slashing mechanisms, the read path, the consensus integration, the economics and governance, and the security analysis.

Table of Contents

Part 1: Foundation

1. Introduction

Decentralized storage has a persistence problem and a payment problem, and they are the same problem seen from two sides. Networks that ask volunteers to pin content cannot guarantee that anyone still holds it tomorrow, because nothing is at stake when a node drops the data. Networks that put something at stake usually reward the wrong thing: sealed capacity, proven once and never read, or per-byte retrieval, which pays nothing for content that is rarely requested and lets the dormant long tail rot. The data that most needs durable hosting, a reference dataset, an archived site, the assets behind a contract, is exactly the data that earns least under a retrieval-weighted model.

DFSP takes a different reward event. It pays operators to prove, continuously and under audit, that they hold a piece of content and can serve it, whether or not anyone reads it. The proof is cheap, the holding is collateralized, and the reward flows for as long as the content is funded. A popular application and a dormant archive are hosted on identical terms, because the protocol pays for availability, not traffic. This single choice, availability as the reward event, determines the rest of the design.

DFSP is the storage layer of DIG Layer 2, a Chia-anchored proof-of-stake rollup specified in Part 1. It does not introduce a new chain, a new consensus, or a new trust anchor. It reuses the layer-2 validators as auditors, the layer-2 block subsidy as the hosting reward, the layer-2 checkpoint as the finalization path to Chia, and the Digstore store of Part 3 as the content format. What DFSP adds is a unit of hosting, the capsule; a mechanism for auditing availability inside ordinary block production; a reward rule that pays audited mirrors from the block subsidy; and a slashing rule that punishes sustained unavailability and any provably wrong service. The result is a storage market whose only counterparty is the protocol, whose security budget is the chain's own, and whose state is anchored to Chia alongside consensus.

This document is the full specification. It states every mechanism, the reasoning behind each design choice, the algorithms that consensus runs, the structures it maintains, the edge cases that shaped the design, and a security analysis of the result. A condensed ten-page treatment exists for readers who want the summary; this is the reference.

2. Design Principles

Five principles carry the design. Each is a deliberate choice with consequences that propagate through the rest of the protocol.

2.1 The capsule is the unit

Hosting is denominated in a single fixed unit: a Digstore store padded to exactly 100 megabytes. Every capsule is the same size, so there is no size term in any reward, any audit challenge, any collateral calculation, or any circuit. A smaller application pays for one capsule and wastes the remainder on padding; a larger application buys more capsules and composes them. Uniformity costs storage and buys simplicity everywhere else, and as the next principle shows, the padding is not wasted: it is a privacy cover set.

2.2 Availability is the reward event

A mirror earns by holding a capsule and passing an audit, not by serving traffic. Reward is decoupled from read demand, so a capsule keeps its mirrors paid and stays replicated whether it is requested a million times a day or never. This is what makes DFSP a durable archive rather than a cache. It also simplifies the economic model: there is no retrieval accounting, no payment channel, no per-request settlement, and no incentive to fabricate read volume.

2.3 The proposer audits, the set verifies

Auditing is folded into ordinary block production. Each block's proposer, the one validator Part 1 selects for that slot, audits a forced sample of mirrors and proposes which of them the block's storage subsidy pays. The rest of the validator set does not re-probe the mirrors; it recomputes the sample from a public seed and re-verifies the cryptographic proofs the proposer collected. One auditor connects to mirrors per block, never the whole validator set at once, which is the difference between an audit and a distributed denial-of-service attack on popular capsules.

2.4 The chain mints and burns

Hosting is funded by emission, not by user payment. The storage role-share of Part 1's block subsidy is the whole hosting reward and its own ceiling: a fixed fraction of each block's issuance, split among the mirrors that block audits. Against that emission the protocol burns at every consumption point, naming, gas, capsule creation, version updates, and the remainder of any slash. Emission is scheduled and burns scale with use, so supply behavior is set by activity rather than by a target.

2.5 State is consensus; collateral is a coin

The protocol draws one hard line between what must be a coin and what is consensus state. A coin guards value or authority that must survive a non-protocol exit: slashable collateral, and a capsule's update authority. Everything else, the registries, the mirrors' accrued reward balances, and their failure weights, is layer-2 state, validated by every node and finalized through the checkpoint. This keeps the on-chain coin footprint minimal, an entry is not a coin, a reward is not a coin until withdrawn, and concentrates the puzzle logic where value actually changes hands.

3. Positioning

Decentralized storage has failed in recognizable ways. DFSP's choices are answers to specific, observed failure modes in the systems that came before it.

System	Failure mode	DFSP's answer
IPFS	Voluntary pinning; persistence is unfunded and unreliable	Collateral keeps data alive; an audit proves it is held

Filecoin	Rewards sealed capacity; proven-empty sectors dominate the market	Rewards availability of real content; every mirror serves it complete
Arweave	One-time endowment, a single durability tier, no renewal lever	Renewable funding; a capsule is hosted exactly as long as it is funded
Storj, Sia	Central coordinator or illiquid bilateral host contracts	On-chain registries; the protocol is the only counterparty

The throughline is that earlier designs either left persistence unpriced or priced the wrong quantity. Voluntary pinning prices nothing, so persistence is a favor. Capacity proofs price disk that may hold no real content. Retrieval payment prices bandwidth, which starves the dormant content that most needs durable hosting. One-time endowments price durability once and cannot adjust as costs or demand change.

DFSP prices the thing it wants: a named object, reachable, genuine, and current, replicated across mirrors, for as long as it is funded. Because the reward is audited availability, the incentive is identical for a rarely-read site and a popular application, and a capsule is hosted because it is funded, not because it is read. The protocol is the only counterparty an operator or a publisher faces; there are no bilateral contracts to negotiate, no coordinator to trust, and no marketplace to match. A publisher funds a capsule and the protocol replicates, audits, pays, and slashes around it.

4. Layer Model and Heritage

DFSP is a capability layered on a running DIG Layer 2. It contributes the capsule, the proposer audit, the storage-reward rule, the slashing rule, and the layer-2 storage state that holds them. It reuses everything else, and the reuse is deliberate: each reused component carries a security argument that DFSP inherits rather than re-derives.

Component	Source	Use in DFSP
Consensus, proposer selection, attestation, checkpoint	Part 1	The substrate; the slot proposer audits, the validator set verifies, the checkpoint finalizes
Block subsidy and the five-role split	Part 1	The storage role-share is the hosting reward and its cap
Block body categories, DfspBlockApply, checkpoint digest	Part 1	The audit result and registry deltas ride in the block and fold into the checkpoint

Membership, majority, aggregation gadget and MPC ceremony	Part 1	Reused as-is; DFSP adds no new circuit and no new trust anchor
ParameterRegistry and Category-A governance	Part 1	DFSP parameters live in the same snapshot and update through the same circuit
Store artifact, URN, execution proofs, blinding	Part 3	The capsule format, its addressing, and its proofs
Coinset model, CLVM, BLS, block headers	Chia	The bond and lineage coins, signatures, verification, and the audit beacon

Mirrors are a distinct role from consensus validators. A validator stakes XCH to secure consensus; a mirror stakes `$DIG` to host a capsule. Slashing one role has no effect on the other, and an operator may run both, neither, or either. The separation gives the combined system two independent security budgets, analyzed in Part 11: attacking consensus requires a stake majority, while attacking a specific capsule's availability requires defeating its mirrors without the audit catching it. The two must be attacked separately.

Redundancy is full replication, not erasure coding. A capsule is a whole Digstore module held by each of its mirrors, so any one healthy mirror serves a request complete and a reader needs no coordination to reassemble fragments. Full replication trades storage overhead, N copies of every capsule, for retrieval simplicity, single-mirror sufficiency, and audit simplicity, every mirror is challenged on the same complete object. Given that the capsule is small and fixed and the protocol optimizes for durable availability rather than raw capacity efficiency, the trade favors replication.

Part 2: The Capsule

5. The Capsule as the Unit

A capsule is a Digstore store fixed at 100 megabytes. It is the atomic unit of hosting: the thing a mirror holds, the thing the audit challenges, the thing collateral is posted against, and the thing the reward is denominated in. Fixing the size is the single decision that removes size-weighting from the entire protocol. No reward formula carries a byte count, no audit challenge scales with capacity, no collateral computation depends on volume, and no Groth16 circuit takes a size as a public input. Every capsule is one equal slot.

The cost of a fixed unit is granularity. An application smaller than 100 megabytes pays for a full capsule and spends the remainder on padding; an application larger than 100 megabytes buys several capsules and composes them (Section 9). The benefit is that every accounting question in the protocol reduces to counting capsules and counting mirrors, never measuring bytes. One hundred megabytes is large enough to hold a substantial site or dataset shard in a single unit and small enough that full replication across N mirrors is cheap, that an audit challenge over the whole object is fast, and that a reader fetches it in a single session. The exact

figure is a structural constant, baked at activation rather than left to governance, because changing it would re-denominate every capsule in existence.

6. Padding, Blinding, and the Cover Set

A store smaller than 100 megabytes is padded to the full size with deterministic noise at compile time. The padding is not appended as an identifiable tail; it is woven into the store by Digstore's blinding (Part 3). Digstore places content chunks into a single interleaved pool together with filler and decoy chunks, and encrypts each chunk under a key derived from the resource URN. The padding is therefore indistinguishable, byte for byte, from real content: a mirror holding a capsule cannot tell which of its 100 megabytes is genuine data and which is filler, because both are ciphertext addressed by hashes the mirror cannot invert.

The noise is seeded from the `store_id`, so the padding is deterministic. Every mirror of a given capsule compiles or receives a byte-identical module with the same content root and the same program hash. This determinism is what lets the audit challenge arbitrary offsets across the whole 100 megabytes and verify the returned bytes against a single on-chain root: there is one canonical capsule, not a family of differently-padded variants.

The storage spent on padding buys two properties at once. First, accounting uniformity: every capsule is exactly one slot, with no size term anywhere, as Section 5 requires. Second, a privacy cover set: because filler is indistinguishable from content and the whole capsule is challenged as a unit, a mirror, an auditor, and a network observer learn nothing about how much real data a capsule holds or what it is. A one-kilobyte secret and a ninety-megabyte dataset present identically. Padding is overhead measured in disk and a feature measured in privacy, and for a protocol whose unit is small and fixed, the disk cost is acceptable.

7. The Anchor and the Lifecycle

A capsule's identity and current state live on the layer 2, anchored by a coin and mirrored in state for cheap lookup. The canonical identifier is a `store_lineage_coin`, a singleton on the layer 2 in the lineage style of CHIP-35. The capsule registry (Section 12) commits, for each capsule, its lifecycle state and its current pair of hashes drawn from that coin.

7.1 Two hashes, two roles

The anchor commits two distinct hashes and advances them independently:

- **`root_hash`** is the merkle root over the capsule's content. It is the subject of inclusion proofs: a reader or an auditor checks that returned chunks recompute this root.
- **`program_hash`** is the hash of the compiled Digstore module. It is the subject of execution proofs: a verifier checks that the module which produced a response is the one the anchor commits to.

The two move on different events. A content change advances `root_hash`. A metadata or logic change recompiles the module and advances `program_hash` while the content root may be unchanged. Because both are anchored, a verifier can distinguish "this mirror served correct content" (inclusion against `root_hash`) from "this mirror ran the correct module" (execution against `program_hash`), and the audit checks both.

7.2 The lifecycle state machine

A capsule moves through four lifecycle states as a function of its funding, tracked in the capsule registry and advanced by deterministic protocol operations at epoch boundaries:

- **Committed.** The capsule has been created and funded but has not yet reached its replica target; it is being onboarded onto mirrors.
- **Live.** The capsule holds its target replication and is being audited and paid. This is the normal serving state.
- **Grace.** The capsule's funding (its handle) has lapsed but the grace window has not expired; it remains served and audited, giving the publisher a window to renew.
- **Expired.** The grace window has passed; the capsule is no longer eligible, its mirrors stop earning on it, and it de-replicates as bonds are released.

Lifecycle transitions are not transactions. They are protocol-level state transitions computed identically by every validator from the registry and the funding state, committed to the capsule-registry root, and finalized through the checkpoint. There is no wall-clock ambiguity: epoch-relative time, derived from the layer-2 block height, is the canonical clock.

8. The URN and Provider Blindness

Addressing follows DIP-0001. A resource within a capsule is named by a uniform resource name:

```
urn:dig:chia:{store_id}[:{root_hash}][/{resource_key}]
```

The `store_id` identifies the capsule, an optional `root_hash` pins a specific version, and an optional `resource_key` selects a resource within it. The URN is the sole input to both locating and decrypting a resource, and this is the foundation of provider blindness.

8.1 Blind retrieval and blind storage

Two keys derive from the URN, and a provider holds neither. The **retrieval key** is a hash of the URN; it is the handle by which a chunk is requested and stored. The **decryption key** is derived from the URN by HKDF; it is held only by a client that knows the URN. A mirror, therefore, stores and serves ciphertext addressed by an opaque hash. It does not know the URN, cannot derive the decryption key, and cannot read the content it hosts. A gateway that relays requests sees only retrieval-key hashes and ciphertext, never the URN. Blindness holds end to end, from the mirror through any relay to the client, and the client performs the only decryption, locally, after verifying the response (Part 8).

This separation of the retrieval key from the decryption key is what lets the protocol pay third parties to host content they cannot read, route requests through untrusted gateways, and audit a mirror's possession of bytes it cannot interpret. The audit, the reward, and the slash all operate on ciphertext and on-chain hashes; none requires the plaintext or the URN.

9. Multi-Capsule Applications

An application larger than one capsule is composed from several. The publisher designates a **root capsule** whose content is a manifest: a list of the URNs of its child capsules. Each child is an independent capsule with its own `store_lineage_coin`, its own bond, its own mirrors, and its own audit. A reader resolves the application's handle to the root capsule, reads the manifest, and then resolves and fetches each child exactly as it would a standalone capsule.

Composition is recursive and uniform: a child may itself be a root capsule for a deeper manifest, and at every level the unit is the same 100-megabyte capsule with the same hosting, audit, reward, and slashing behavior. There is no special large-object code path, no sharding logic in the protocol, and no cross-capsule atomicity requirement. The application's structure lives in its manifests, which are themselves content under the same guarantees, and the protocol sees only a set of independent capsules. This keeps the protocol's surface small while letting applications scale to arbitrary size by multiplying the unit.

Part 3: State and Token

10. The Coin and State Boundary

DFSP draws one hard line between a coin and consensus state, and the line is principled rather than incidental. A coin is needed when value or authority must survive a non-protocol exit, that is, when a party must be able to act on it, or be made to forfeit it, in a way the protocol's own state transitions cannot fully capture. Two things meet that test: **slashable collateral**, which a spend must be able to seize without the owner's consent, and a **capsule's update authority**, which a writer must be able to exercise and delegate. Everything else, every index, balance, and running weight, is consensus state: validated by every node, committed in a merkle root, and finalized to Chia through the checkpoint.

Object	Coin or state	Why
Mirror bond, one per (node, store)	Coin	Slashable collateral; a single continuous bond, never re-posted
`store_lineage_coin`, per capsule	Coin	Capsule identity, current hashes, and writer/update authority
Consensus validator stake	Coin (Part 1)	Reused; not DFSP's to define
Capsule registry	State, indexed	Lifecycle and current hashes; enumerable for the audit
Mirror registry	State, indexed	The hosting set; per-entry accrued balance and failure weight

The discipline pays off in two ways. The on-chain coin footprint stays minimal: a registry entry is not a coin, and an accrued reward is not a coin until a mirror withdraws it, so the protocol does not mint a coin per mirror per epoch or hold a coin per pending balance. And the puzzle logic, the part of the system that must be written in CLVM and reasoned about adversarially, is confined to the two coins where value and authority actually change hands. State transitions, by contrast, are computed in protocol code that every validator runs identically and that the checkpoint anchors, which is cheaper to specify and to verify.

11. The Coins

11.1 The mirror bond

A mirror bond is a singleton coin holding a flat collateral in `$DIG` for exactly one capsule. A node serving K capsules posts K bonds, one per capsule. The bond is **single and continuous**: it is posted once when the mirror registers and is never re-posted across epochs. It persists for the life of the mirror entry and is consumed only by a slash or by a voluntary exit. Its puzzle exposes the spend paths that the lifecycle requires:

- **Slash.** The whole collateral is forfeited, without the owner's signature, on either of two triggers: the failure weight crossing the threshold, or a valid reader fraud proof (Part 7). The slash routes the collateral to a burn, with a reporter bounty carved out in the fraud-proof case.
- **Exit.** The owner withdraws the collateral after an unbonding delay. The delay, on the order of Part 1's bond-release window, exists so that a mirror cannot exit to dodge a slash that is already justified by accumulated failure weight; the weight and any pending fraud proof resolve before the collateral is released.

Holding collateral in a coin, rather than as a state balance, is what makes a slash unconditional. The protocol does not need the mirror's cooperation to seize a coin; the puzzle permits the slash path on presentation of the triggering evidence, and any party, or the block rule itself, can execute it.

11.2 The store_lineage_coin

The `store_lineage_coin` is the capsule's on-chain identity and the seat of its update authority. It is a singleton whose lineage is the canonical capsule identifier, in the style of CHIP-35's store anchor. It commits the capsule's current `(root_hash, program_hash)` and carries the writer authority to advance them, which may be delegated under the anchor's own policy. Advancing a version is a spend of this coin that re-commits the new hashes; the capsule registry mirrors the new pair when the spend is applied. Authority is held in a coin, rather than a state flag, for the same reason collateral is: a writer must be able to exercise and delegate control through puzzle-enforced spend paths, independent of the protocol's bookkeeping.

12. The Registries

All DFSP state that is not a coin lives in two indexed registries, both maintained by the block-application rule and both committed in roots that ride in the attestation and fold into the checkpoint.

12.1 The capsule registry

The capsule registry is indexed by `store_id`. Each entry holds the capsule's lifecycle state and its current `(root_hash, program_hash)`, mirrored from the `store_lineage_coin`. The registry exists so that the set of capsules, and in particular the set of live capsules, is cheap to enumerate. The audit samples from it, the lifecycle machinery advances entries through it, and a reader reads a capsule's current hashes from it as the root of trust before fetching.

12.2 The mirror registry

The mirror registry is indexed by `(node, store_id)` and is the hosting set: the list of which nodes mirror which capsules. It is the structure the audit samples, the reward credits, the slash check reads, and a reader discovers mirrors from. Each entry is the `MirrorEntry` of Section 17, and crucially each entry carries, alongside the served version and endpoint, two pieces of accounting that a naive design would split into separate stores but which belong on the entry: an `accrued_balance` (reward state, Section 29) and a `fault_weight` with its `fault_block` (the slashing accumulator, Section 34). Folding both onto the mirror entry means the slashing store and the reward ledger are the same indexed structure the audit already touches, with no extra table to maintain or join.

12.3 Maintenance and finalization

Both registries are maintained by `DfspBlockApply` (Section 47), the block-application rule that Part 1 already runs for layer-2 state. Each block, it applies the audit result and the registry deltas, then requires the block header's committed registry roots to match the recomputed state; a validator that computes a different root rejects the block, so the registries advance only by agreement. The roots ride alongside the consensus state root in every BLS attestation and fold into the checkpoint digest at epoch close, so DFSP state is hard-finalized to Chia on the same path and with the same cryptographic strength as consensus state.

13. Emission

`$DIG` is minted on Part 1's perpetual block schedule, a Chia-style curve with a non-zero asymptote, and each block's issuance is split across five reward roles: proposer, attester, inclusion-delay, checkpoint signer, and the **storage provider**. The first four pay consensus work. The fifth is the entire hosting reward, and DFSP defines it.

13.1 The storage role-share is the cap

The storage role-share is a fixed fraction of each block's subsidy, set as a Category-A governance parameter. It is the only `$DIG` that can reach mirrors, which makes the share percentage the emission cap by construction: there is no separate ceiling to compute or enforce, because a fixed slice of a scheduled subsidy is inherently bounded. Each block, that slice is split among the mirrors the block's proposer audited and verified (Part 6). The reward arithmetic, like all of Part 1's issuance, is deterministic from the block contents and recomputed by every validator, so disagreement on who is paid how much is disagreement on the block.

13.2 Automatic dilution

Because the share is a fixed quantity split among a block's verified passers, per-mirror reward falls automatically as participation rises. More mirrors in the network means more audited per block and a smaller cut each; fewer mirrors means a larger cut. The protocol never adjusts a per-mirror rate, sets a target, or runs a controller. Total hosting emission stays exactly the role-share regardless of network size, and the market finds the equilibrium mirror count at which the diluted reward still covers storage cost.

13.3 The bootstrap schedule

The storage role-share is set high at genesis and declines on a fixed, public schedule to a steady-state value. This solves the bootstrap problem, that the network must emit enough `$DIG` for a genesis set of mirrors to stake before hosting can run, without a premine or a treasury. The activation height is set after enough `$DIG` has been emitted to the genesis mirror set, and from activation the storage share funds hosting directly. The other four roles expand proportionally as the storage share declines. The schedule is committed in client code and is not extendable by governance on the basis of network conditions: the share declines regardless, so mirrors must develop a sustainable position or exit.

14. Burn and Supply

Against scheduled emission, the protocol burns `$DIG` at every point where the network is consumed. The sinks are structural, not discretionary.

Burn sink	Trigger	Scales with
Handle fee	Handle registration and renewal	Naming demand
Gas base fee	Every layer-2 transaction	All network activity
Capsule creation	Posting a capsule for hosting	New capsules; rations capsule count
Version update	A write advancing the root or program hash	Update frequency
Slash remainder	A reader fraud proof, after the bounty	Misbehavior

Handle fees burn in full. Because hosting is funded by the subsidy, a handle does not pay for hosting; it buys a name and the eligibility to be hosted, and the payment is destroyed. The gas base fee follows Part 1's fee model, a fraction of every transaction's fee is burned. Capsule creation and version updates burn fees that track write demand and ration the creation of new capsules so that the audit and reward load grows with genuine usage. A slash, in the fraud-proof case, pays the reporter a bounty and burns the remainder, so enforcement is incentivized and the slash is still a net sink.

Supply behavior is the difference between scheduled emission and usage-driven burn. At low usage, emission dominates and `$DIG` is mildly inflationary; at high usage, with many handles, many transactions, and frequent updates, burn can dominate and `$DIG` can turn net deflationary. The crossover is a function of network activity, handle count, transaction throughput, and capsule churn, not a parameter the protocol targets. The protocol aims at no supply number; activity moves the crossover.

15. Handles and Eligibility

A capsule is funded and named through a handle, drawn from the DIG naming registry (the XCHandle system). The handle is the publisher-facing object: a human-readable name that resolves to a capsule's `store_id`, purchased and renewed for a fee, with tiered pricing and a renewal grace period.

15.1 The eligibility gate

Hosting eligibility is gated on a paid handle. A capsule is eligible to be a paid, audited mirror target only while its handle is current; the capsule registry reflects this in the lifecycle state, moving a capsule to grace and then expired as its handle lapses. The handle is therefore the funding lever: paying and renewing it keeps the capsule live and its mirrors earning, and letting it lapse de-replicates the capsule. Because hosting itself is paid by emission, the handle fee is purely a gate and a burn, not a hosting payment, which is why it is destroyed in full.

15.2 Pricing and the cost floor

Handle pricing is tiered, and the cheapest tier is floored so that the fee a publisher pays over a term is not less than the cost of the hosting the protocol provides over that term, N mirrors auditing a capsule across the term's blocks. The floor prevents a publisher from funding hosting more cheaply through the handle than the emission that pays for it would cost, which keeps the emission-funded model from being arbitraged by underpriced names. Above the floor, tiers price desirable names by demand, and those fees, like all handle fees, burn.

Part 4: The Mirror

16. The Mirror Role

A mirror is an operator that holds a complete copy of a capsule, answers audits on it, and earns the storage role-share for the blocks in which it passes. The role is defined by two on-chain objects: a continuous bond coin (Section 18) and a registry entry (Section 17). Together they say that a particular node, identified by a BLS key, has staked a particular collateral against a particular capsule and stands ready to serve it.

Mirrors are distinct from consensus validators and stake a distinct asset. A validator stakes XCH and secures consensus; a mirror stakes `$DIG` and hosts a capsule. The two roles are slashed independently, audited independently, and rewarded from different parts of the block subsidy, and an operator is free to run both, one, or neither. Co-locating the roles is an operational convenience, not a protocol coupling, and the security analysis (Part 11) treats them as separate budgets precisely because the protocol does.

Every mirror holds the whole module. Redundancy is full replication, not erasure coding: there are no parity shards, no reconstruction threshold, and no coordination among mirrors to answer a request. Any one healthy mirror serves a capsule complete, which makes retrieval a single-mirror operation with trivial failover and makes the audit a uniform challenge, every mirror is asked about the same complete 100-megabyte object against the same on-chain root. The cost is N full copies per capsule; the benefit is that no reader, auditor, or mirror ever depends on the simultaneous health of a quorum of mirrors to complete an operation.

17. The Mirror Entry

The registry entry for a mirror records who is hosting what, at which version and address, against which bond, and with what running reward and failure state. It is consensus state, indexed by `(node, store_id)` in the mirror registry:

```
MirrorEntry {                        // consensus state, indexed by (node, store_id)
   node_pubkey:     Bytes48,     // node BLS key; audit and payout identity
   store_id:        Bytes32,     // the capsule
   root_hash:       Bytes32,     // served content root
   program_hash:    Bytes32,     // served module hash
   address:         Bytes,       // advertised endpoints (ipv6 primary, ipv4 fallback)
   bond_id:         Bytes32,     // the continuous bond coin it stakes
   accrued_balance: u64,         // reward credited by audits, minted on withdrawal
   fault_weight:    u64,         // decaying failure accumulator (fixed point)
   fault_block:     u64,         // L2 block at which fault_weight was last updated
}
```

The fields divide into identity, service, and accounting. The identity is `node_pubkey`: the BLS key that signs the mirror's audit responses and receives its reward, the single cryptographic identity the audit and payout both key on. The service fields are `root_hash` and `program_hash`, the version the mirror is currently serving, and `address`, the endpoints it advertises, an IPv6 address primary with an IPv4 fallback. The collateral link is `bond_id`, the continuous bond coin this entry stakes.

The accounting fields are the two the protocol folds onto the entry rather than splitting into separate stores. `accrued_balance` is the reward the audits have credited but the mirror has not yet withdrawn (Section 29). `fault_weight` with `fault_block` is the leaky-bucket failure accumulator and the block at which it was last aged (Section 34). There is no size field, because the capsule is constant; the entry never carries a byte count. Keeping reward and failure state on the same indexed entry the audit already reads means a block's reward credit and slash check touch one structure, not three.

18. The Bond

The bond is the mirror's stake: a single continuous coin holding a flat `$DIG` collateral for one capsule, posted once at registration and never re-posted. A node mirroring K capsules posts K bonds, one per capsule, so collateral is per-capsule and a slash on one capsule does not touch the node's stake on another. Section 11.1 specifies the coin; this section specifies how its spend paths bind to the mirror lifecycle.

The bond exposes a slash path and an exit path, and the asymmetry between them is the whole point. The slash path forfeits the entire collateral with no owner signature, on either trigger, the failure weight crossing the

threshold, or a valid reader fraud proof. The exit path returns the collateral to the owner, but only after an unbonding delay on the order of Part 1's bond-release window. The delay closes the obvious evasion: a mirror that has been failing audits, and whose failure weight is climbing toward the threshold, cannot simply exit and reclaim its collateral ahead of the slash. The unbonding delay holds the collateral in place long enough for accumulated failure weight to cross, and for any in-flight fraud proof to land, before release. A mirror that wants its collateral back while in good standing waits out the delay; a mirror trying to dodge a justified slash finds the collateral still seizable throughout the delay.

19. Mirror Lifecycle

A mirror entry has four operations, which together move it from posting collateral through earning to release:

- **Register.** Post the bond coin and write the registry entry. The mirror begins earning the first block in which it passes an audit; it is not paid merely for existing in the registry.
- **Update.** Re-commit a new served version (a new `(root_hash, program_hash)`) or a new address. A mirror updates its served version when the capsule's anchor advances, within the version grace window (Section 46).
- **Slash.** The bond is spent in full when the failure weight crosses the threshold, or when a reader fraud proof lands, and the entry is removed (Part 7).
- **Exit.** Release the bond after the unbonding delay and remove the entry, the voluntary, in-good-standing departure.

The binding between a mirror's `node_pubkey` and its advertised `address` is enforced by the audit, not by a separate registration check. A passing audit response is signed by the entry's key, so a node that advertises an address it does not actually serve from cannot produce a valid signed proof from that address, and simply fails the audit. There is no probation period and no separate liveness bond. The failure weight (Section 34) absorbs the transient misses that a newly-registered or briefly-interrupted mirror will have, and new entries are weighted into the audit sample so that the window in which a fresh mirror is unproven stays short. A mirror proves itself by passing audits, and the protocol measures it continuously rather than gating it at entry.

Part 5: The Audit

20. Auditing as a Consensus Duty

The auditor is the block's proposer. Part 1 selects, for each slot, one primary proposer deterministically from the prior state root, the epoch, and the slot index, with a fallback sequence of alternate proposers if the primary does not broadcast within the slot's timeout. DFSP gives that proposer one additional duty: while building its block, it audits a seed-forced sample of mirrors and proposes, in the block body, which of the sampled mirrors that block's storage role-share pays.

The rest of the validator set does not re-probe the mirrors. When a validator receives the proposed block, it recomputes the sample from the public seed, re-verifies the cryptographic proof the proposer collected for each

rewarded mirror, and then attests to the block under Part 1's ordinary attestation rules. The audit thus produces no new consensus round, no new message type, and no new quorum: it rides entirely inside the existing propose-and-attest flow. A block carries its audit result the way it carries its transactions, and a validator validates the audit result the way it validates the transactions, by recomputing and checking, then attesting.

This placement is what lets DFSP add a storage audit to a running chain without adding a circuit, a committee, or a probing protocol. The work of connecting to mirrors is done once per block, by the one party already chosen to build that block, and the work of verifying that the connection's results were applied correctly is done by everyone, offline, from data already in the block.

21. One Auditor per Block

The decision that exactly one validator probes mirrors per block, rather than the whole set, is deliberate, and it is what keeps the audit from becoming a denial-of-service attack on the network's own mirrors. Consider the alternative. If every validator independently probed every sampled mirror to verify availability for itself, then a popular capsule, or any capsule in a small sample, would face a probe from the entire validator set on every audit. The most-hosted, most-valuable capsules would absorb the heaviest probing load, scaling with validator-set size, and an operator of a popular capsule would be punished with traffic for being popular. The audit would manufacture exactly the load spike it exists to insure against.

Under the proposer-audit model, exactly one validator, the slot's proposer, connects to each sampled mirror, against a bounded sample, once per block. The probing load on the network is one proposer's worth per block, and it spreads across blocks and across the rotating proposer selection, so over an epoch the audit traffic is distributed across the whole validator set as each takes its turn proposing, but at any moment only one validator is probing. A mirror sees an occasional probe when it happens to fall in a proposer's sample, never a simultaneous probe from every validator. The cost of this efficiency is that the set must verify the proposer's results rather than reproducing them, which the next sections show is sound for every check except one, reachability, and that the residual trust in that one fact is bounded and consensus-measured.

22. The Audit Seed

The sample a proposer must audit is forced by a seed that the proposer of that block cannot grind. The seed binds three inputs, none of which the current proposer controls:

```
audit_seed = Hash(epoch_beacon || parent_hash || mirror_registry_root)
```

- `epoch_beacon` is the Chia block header hash at the epoch's anchor height. It is exogenous to the layer 2, produced by Chia's proof of space and time, and no layer-2 proposer has any influence over it.
- `parent_hash` is the hash of the previous layer-2 block. It was fixed by the previous proposer, before the current proposer began to act.
- `mirror_registry_root` is current consensus state, the committed root of the mirror registry, identical for every validator.

22.1 Why the current proposer cannot grind it

The proposer building this block controls none of the three inputs. The beacon is Chia's, set before the epoch began and reorg-settled by the time audits run; the parent hash is the prior block's, already on the chain; the registry root is consensus state. A proposer therefore cannot shape its own sample, cannot arrange to audit a capsule it co-owns, cannot steer the reward toward a mirror it controls, and cannot spare a mirror it wants to protect from a failing audit. The sample it must audit is determined before it has any ability to influence it.

22.2 Verification of the beacon

Every validator and full node already follows Chia to track the checkpoint, so the beacon is a fact the whole set can check. The block-application rule requires `epoch_beacon` to equal the Chia header hash at the epoch's anchor height and rejects any block that seeds from a different value. The anchor height is chosen deep enough that the beacon block is reorg-settled on Chia before the epoch's audits begin, so the seed is stable: there is no window in which a late Chia reorg could retroactively change the sample a block was required to audit.

22.3 Unpredictability to mirrors and the residual

The `parent_hash` term also makes the sample unpredictable to mirrors until the prior block lands. A mirror learns whether it is in the next block's sample only about one slot in advance, far too little time to come online only when it is about to be audited and go dark otherwise. The one residual grinding vector is a *prior* proposer grinding its own block hash to nudge the *next* block's sample. That attack costs a full block grind, helps only a colluding successor proposer (the prior proposer cannot reward itself through the next block's sample), and can only shift the sample toward mirrors that genuinely serve, since a fabricated pass is independently impossible (Section 26). It is expensive, narrow, and self-limiting, and it is bounded further by the honest-majority-of-proposers assumption DFSP already inherits from Part 1.

23. The Sample

The seed draws a fixed-size sample of `(node, store)` entries from the mirror registry each block. The sample is sized to fit the work a single proposer can do within the slot window: the proposer must connect to each sampled mirror, issue a challenge, and collect a signed response, all inside its slot, so the count is bounded by what is achievable in roughly a ten-second slot.

The sample's behavior has two regimes relative to the size of the hosting set. Below the sample's reach, when the registry holds fewer entries than the sample size, every capsule is audited every block: full coverage, every mirror checked continuously. Above the sample's reach, when the registry is larger than a single block can cover, the draw is uniform across entries and capped at N per capsule, so that no single popular capsule consumes the whole sample and audit attention spreads across the hosting set. The cap at N matches the replica target: there is no value in auditing more than N mirrors of one capsule in a block, because N is the number the protocol pays and maintains.

The relationship between sample size, registry size, and audit frequency sets the *audit gap*, the expected number of blocks between successive audits of a given mirror, which is a central quantity for slashing calibration. A mirror in a small registry is audited nearly every block; a mirror in a large registry is audited less often, on a frequency that the failure-weight half-life must exceed (Section 35) so that sustained failure still accumulates faster than it

decays between sparse audits.

24. The Audit Checks

The proposer issues a seed-derived challenge to each sampled mirror, connects to its advertised address, and collects a signed response. For each mirror it evaluates five checks. Four of the five are self-verifying against the on-chain anchor and will be re-checked by the whole validator set; only the first rests on the proposer's direct observation.

- **Reachability.** The mirror answered at its advertised address. This is the one fact only the proposer observes directly, the single trusted input to the whole audit.
- **Possession.** Random offsets drawn from the seed, spanning the whole 100 megabytes including padding, return the correct bytes. Because the padding is woven through the store and indistinguishable from content, possession of the whole object is tested, not possession of a known data region.
- **Inclusion.** The returned chunks carry merkle paths that recompute the capsule's committed `root_hash`. This proves the bytes are the genuine content, not arbitrary data of the right length.
- **Execution.** A zero-knowledge execution proof verifies against the committed `program_hash`, establishing that the correct compiled module produced the response. Where producing a full execution proof is prohibitively expensive, Digstore's hardware-attested proof stands in, under Part 3's attestation, at the same verification interface.
- **Freshness and version.** The execution proof commits to a recent Chia block header (Digstore freshness), so the response cannot be a precomputed or replayed artifact, and the served `(root_hash, program_hash)` matches the anchor's current pair, subject to the version grace window.

The challenge is seed-derived so that the offsets a mirror is asked about are unpredictable until the block's seed is known, which prevents a mirror from caching answers to a fixed challenge while discarding the data. Possession, inclusion, execution, freshness, and version are all checkable from the response against on-chain hashes, so any validator can re-evaluate them from the block alone. Reachability is the only check that requires having been the party that connected, and it is the only one the set cannot reproduce.

25. The Proof Bundle

For each mirror the proposer marks as passing, the block body carries that mirror's signed proof bundle in a dedicated `audit_results` category. The bundle contains the challenged chunks, their merkle paths, and the fresh execution proof, all signed under the mirror's `node_pubkey`. The set of mirrors the proposer marks as failing is implicit: it is the seed-derived sample minus the set of passers carried in the block. The proposer does not include proofs for failures, because a failure is the absence of a valid passing proof, and there is nothing for a failing mirror to have signed.

Carrying the signed bundles in the block is what makes the audit verifiable by the set without re-probing. The bundle is the mirror's own cryptographic statement, fresh and signed, that it held and served the challenged bytes from the committed version. A validator does not need to have connected to the mirror to check that statement; it

needs only the bundle and the on-chain anchor. The block, therefore, contains within itself everything required to re-verify every claimed pass, and the only thing it cannot contain is proof that an unreachable mirror was in fact unreachable, which is the reachability residual.

26. Validation in DfspBlockApply

When a validator applies a proposed block, Part 1's `DfspBlockApply` re-runs the verifiable part of the audit on the whole block before the validator attests. The procedure is deterministic and identical on every node:

```
DfspBlockApply(block, state):
   # 1. Reconstruct the sample the proposer was required to audit.
   seed   = Hash(epoch_beacon(block) || block.parent_hash || state.mirror_registry_root)
   sample = draw_sample(seed, state.mirror_registry)        # fixed size, capped at N per store
   require block.audit.sample_commitment == commit(sample)  # same sample, no substitution

   # 2. Re-verify every claimed pass from the block alone.
   for entry in block.audit.passers:
       m = state.mirror_registry[entry.node, entry.store_id]
       require entry in sample
       require merkle_verify(entry.chunks, entry.paths, m.root_hash)        # inclusion
       require exec_verify(entry.exec_proof, m.program_hash)                # execution
       require fresh(entry.exec_proof, recent_chia_header)                  # freshness
       require version_ok(entry.served_pair, m, grace_window)               # version
       require bls_verify(m.node_pubkey, entry.signature, entry.response)   # signature

   # 3. Credit reward to verified passers; advance failure weight on the rest.
   failers = sample - block.audit.passers
   share   = storage_role_share(block)
   per     = share / count_capped_per_store(block.audit.passers, N)
   for entry in block.audit.passers:  state.mirror_registry[entry].accrued_balance += per
   for entry in failers:              advance_fault_weight(state.mirror_registry[entry], block.height)

   # 4. Apply lifecycle and registry deltas, then require the committed roots to match.
   apply_lifecycle_and_deltas(block, state)
   require block.header.capsule_registry_root == commit(state.capsule_registry)
   require block.header.mirror_registry_root  == commit(state.mirror_registry)
   # Any failed require rejects the block.
```

Three things follow from this procedure. First, the proposer cannot fabricate a pass: every passer must carry a fresh, signed, on-chain-verifiable proof, and a forged or stale proof fails one of the `require` checks, so the block is rejected. Second, the proposer cannot bias the selection: the sample is recomputed from the seed and a substituted sample fails the commitment check. Third, the reward and slash arithmetic is deterministic and recomputed by every validator, so a proposer that credits the wrong amounts, or slashes the wrong entries, produces a block whose registry roots do not match and is rejected. No validator other than the proposer connects to a mirror at any point; the entire validation is offline, against the block and the anchor.

27. The Trust Surface

Exactly one fact in the whole audit is not reproducible by the validator set: whether a given mirror actually answered the proposer at its advertised address. Reachability is the proposer's direct observation, and it is the entire trusted surface of the protocol. Everything else, possession, inclusion, execution, freshness, version, the reward arithmetic, the sample selection, is recomputed and re-verified by every validator from the block and the on-chain anchor.

Because reachability is the only trusted input, a dishonest proposer can lie in only one direction. It cannot fabricate a pass, since a pass requires a fresh signed proof the set re-verifies (Section 26). It cannot bias which mirrors it audits, since the sample is seed-forced and recomputed (Section 22). What it *can* do is withhold a deserved reward from a mirror that actually served, or claim that a mirror which actually answered was unreachable, marking a serving mirror as a failure. Both lies are bounded. Each costs the target one block's share of reward and one failure-weight increment, and a single increment never slashes; a slash requires the failure weight to cross a threshold built from many increments contributed by many independent proposers over many blocks (Part 7). A lie by one proposer in one block is therefore a one-block, one-increment event that no single actor can escalate into a slash.

The assumption that makes this safe is an honest majority of proposers, which is exactly Part 1's consensus assumption; DFSP introduces no new trust assumption of its own. A slash for unavailability reflects sustained failure measured across many rotating proposers, not the word of any one auditor, and a proposer that systematically lies about reachability is, in aggregate, behaving as a dishonest fraction of the validator set, which Part 1's security model already bounds. The single trusted fact is the minimum the proposer-audit model requires, and its abuse is contained by the same majority assumption that secures the chain.

Part 6: Rewards

28. Per-Block Reward

A mirror earns per block, for the blocks in which it is audited and passes. There is no standing accrual for mere registry membership and no shared pool that fills over time. Each block, the storage role-share for that block is split equally among that block's verified passers, capped at N per `store_id`, and the resulting share is credited to each passer's `accrued_balance` on its mirror entry.

Three cases follow from this rule. A mirror not drawn into a block's sample earns nothing from that block; it is simply not up for audit. A mirror drawn and failing earns nothing from that block and takes a failure-weight increment, but is not slashed for the single miss. A mirror drawn and passing is credited its equal share of that block's storage role-share. Reward is thus strictly a function of audits passed, and a mirror's income over time is the sum of its per-block credits across the blocks in which it was sampled and served.

29. Accrual and Settlement

`accrued_balance` is Part 1's protocol-managed reward state: a number on the mirror entry, not a coin. It is never a coin until the mirror chooses to realize it. A mirror converts accrued reward to spendable `$DIG` with a `WithdrawAccrued` transaction, which decrements the balance on the entry and mints one `$DIG` coin to the mirror's address, authorized by the node's wallet key over a domain-tagged signed payload (in the style of Part 1's `SignedPayload`, with a `"DIG_DFSP_WITHDRAW_ACCRUED"` domain tag):

```
WithdrawAccrued(node, payout):
   require payout <= sum over the node's entries of accrued_balance
   require bls_verify(node.wallet_key,
                      SignedPayload("DIG_DFSP_WITHDRAW_ACCRUED" || node || payout || nonce),
                      sig)
   decrement accrued_balance across the node's entries by payout
   mint one $DIG coin of value payout to node.address
```

Settlement is lazy by design. Because reward accrues as state and mints only on withdrawal, the protocol creates no coin per block and no coin per mirror per audit. A node that mirrors many capsules accrues across all of them and settles them in a single mint when it chooses, so the on-chain coin creation rate is set by how often operators withdraw, not by the audit cadence. This is the practical payoff of the coin-versus-state line from Section 10: the reward ledger is cheap state that the audit already maintains, and it becomes a coin only at the moment value leaves the protocol.

30. Dilution and the Cap

The storage role-share is a fixed quantity per block, so per-mirror reward falls as the number of audited, passing mirrors rises. This dilution is automatic and requires no controller. When the network grows and more mirrors pass audits per block, the same fixed share divides into smaller per-mirror credits; when mirrors leave, the share divides into larger ones. The protocol sets no per-mirror rate and chases no target. Total hosting emission is exactly the role-share regardless of how many mirrors participate, so the emission cap holds by construction and the market, not the protocol, discovers the mirror count at which the diluted per-mirror reward still covers an operator's storage cost.

The per-block split is capped at N per `store_id` before division, which interacts with dilution to shape where mirrors go. Within a block, no more than N passers of a single capsule are paid, so a capsule with more than N healthy mirrors does not draw more than N capsules' worth of that block's share toward itself, and the (N+1)th mirror of an over-replicated capsule earns nothing on the excess.

31. Convergence to N

Equal-weight reward and the cap at N together drive the hosting set toward exactly N replicas per capsule with no central assignment. The argument is a simple incentive gradient. A capsule with fewer than N mirrors is under-replicated: a new mirror joining it is within the cap, passes audits, and earns a full share, so under-replicated capsules attract mirrors. A capsule with exactly N mirrors is at target: a further mirror would be the (N+1)th, earn nothing on the excess, and merely add storage cost, so mirrors do not join capsules already at N. A capsule above N pays nothing on the surplus, so the surplus mirrors have an incentive to leave and re-home on an under-replicated capsule.

The equilibrium is N replicas per funded capsule, reached by rational operators each maximizing their own diluted, capped reward, without any protocol component assigning mirrors to capsules. The protocol publishes the gradient, under-replicated capsules pay, over-replicated capsules do not, and operators climb it. New funded capsules and capsules whose mirrors were slashed or exited appear as under-replicated and are filled by operators seeking the full share, which is the same mechanism that produces self-healing (Section 41).

32. Income Dynamics

A mirror's expected income tracks its share of the registry, realized through the audits it passes, and the per-block variance averages out over many blocks. In expectation, a mirror holding a fraction of the registry's entries is sampled in proportion to that fraction and, passing, earns in proportion to it; the law of large numbers smooths the block-to-block randomness of which entries are drawn. Two mirrors of identical reliability and capsule count have the same expected income regardless of which specific blocks happened to sample them.

The realized income depends on reliability and on the audit gap. A mirror that passes every audit earns its full expected share; a mirror that fails a fraction of audits earns less, in proportion to its pass rate, and accumulates failure weight toward a possible slash if the failure rate is high enough (Section 35). In a large registry where the audit gap is wide, income arrives in lumpier increments, since a given mirror is sampled less often, but the long-run expectation is unchanged. Income is therefore predictable in aggregate and stochastic per block, which is appropriate for a reward that funds ongoing hosting rather than discrete deliveries.

Part 7: Slashing

33. Two Triggers

Slashing backs every hosting claim with the bond, a coin a spend can take in full. There are two triggers, and they are keyed to who is in a position to detect the fault. **Liveness** faults, a mirror failing to answer audits, are detected by the proposer audit and punished through a decaying failure weight that slashes only on sustained failure. **Correctness** faults, a mirror serving verifiably wrong or stale content, are detected by any reader and punished immediately through a fraud proof. The split matters because the two fault types differ in provability: a wrong answer is a provable artifact, while a missing answer is not, and the protocol slashes only what can be proven.

34. The Leaky-Bucket Failure Weight

A single failed audit does not slash. It is one proposer's word, and it may be a momentary blip, a brief network interruption, a restart, a transient route failure. Slashing on one miss would punish noise and would hand a single dishonest proposer the power to slash by lying about reachability once. Instead, each failed audit adds a fixed weight to the entry's `fault_weight`, and that weight decays continuously with the layer-2 blocks since it was last touched.

34.1 The update rule

The accumulator is a leaky bucket. When block `b` audits a mirror, the stored weight is first aged, then updated:

```
age:     fault_weight *= decay_factor ^ (b - fault_block)
update:  fault_weight += FAILURE_INCREMENT   if the audit failed
                      += 0                   if the audit passed
set:     fault_block   =  b
```

The decay is geometric in the number of blocks elapsed. Isolated failures age toward zero before the next audit arrives, so a mirror that fails once and then serves reliably sheds the increment and never approaches the threshold. Failures dense in audited time outrun the leak: increments arrive faster than decay removes them, the weight climbs, and it eventually crosses the threshold.

34.2 Why geometric decay

The decay must be geometric (equivalently, exponential in elapsed blocks), and the choice is not cosmetic. Only geometric decay is memoryless, which is the property that lets the entire failure history compress into a single number plus a timestamp. Because `d^(a+b) = d^a · d^b`, aging a stored weight forward by the elapsed block count is exact regardless of when the constituent increments were added; the protocol never needs to remember individual failures or their times. A logarithmic or linear decay would lack this property: it would force the protocol to retain a per-failure history to compute the current weight correctly, and it would barely decay over realistic spans, so old failures would never be forgiven. Geometric decay is the unique rule under which `(fault_weight, fault_block)` is a sufficient statistic for the whole history.

34.3 Fixed-point determinism

The decay is computed in fixed-point arithmetic, so every validator computes the same weight from the same inputs to the bit. This is what lets the slash be a consensus operation: `DfspBlockApply` recomputes each audited entry's aged-and-updated weight identically on every node, and re-verifies any slash that results. The only non-reproducible input behind the weight is the sequence of reachability observations the proposers reported; the arithmetic on top of those observations is deterministic and checked by all.

35. Threshold and Tolerated Failure Rate

When an aged-and-incremented weight crosses a threshold, the bond is slashed in full. The threshold, together with the failure increment and the decay half-life, defines the **tolerated failure rate**: the steady-state fraction of audits a mirror may fail without its weight crossing. At a steady failure rate, the weight approaches an equilibrium where per-audit increments balance inter-audit decay; if that equilibrium sits below the threshold, the mirror survives indefinitely, and if it sits above, the mirror is eventually slashed. The three parameters are chosen so the equilibrium crosses the threshold at the failure rate the protocol considers a breach of its implied service level.

The half-life carries a hard constraint relative to the audit gap (Section 23). It must exceed the expected gap between a mirror's audits; otherwise, in a large registry where audits are sparse, a failing mirror's weight would decay substantially between hits and never accumulate, letting a persistently dark mirror escape because it is audited too rarely for its failures to compound. Setting the half-life above the audit gap ensures that sustained failure accumulates faster than it leaks even under sparse sampling. The half-life, the per-failure increment, and the threshold are Category-A governance parameters, calibrated so the tolerated failure rate matches a realistic availability target and the implied service level is neither so strict that ordinary operational noise slashes honest mirrors nor so loose that a frequently-dark mirror keeps its bond.

36. The Slash Event

The slash is event-driven and lands in the block where the crossing occurs. Because decay only ever lowers the weight, the weight can cross the threshold only at the moment a failure increment is added, never between audits. The block-application rule therefore catches every crossing at its cause: in the same block that applies a failing audit and pushes a weight over the threshold, it executes the slash. There is no separate sweep, no scheduled check, and no possibility of a crossing going undetected, because the only event that can produce a crossing is one the block is already processing.

The slash spends the bond coin in full along its slash path, with no owner signature, and removes the mirror entry. The whole collateral is forfeited; the slash is not graduated. Because the triggering weight is recomputed deterministically by every validator (Section 34.3), the slash is itself a consensus operation that every node re-verifies: a block that slashes an entry whose recomputed weight did not cross is rejected, and a block that fails to slash an entry whose weight did cross is likewise invalid. The only input the set takes on trust behind a liveness slash is the sequence of reachability claims that built the weight, and that sequence reflects many independent proposers, so a wrongful slash would require a coordinated majority misreporting reachability over many blocks, which Part 1's appeal window (its optimistic fraud-proof window) additionally covers.

37. The Reader Fraud Proof

The second trigger is a reader's fraud proof, for correctness, and it is immediate. It does not wait for a threshold and does not depend on the audit; any party that holds the right artifact can slash a misbehaving mirror in a single step. The artifact is narrow and specific:

- **Signed.** A response signed by the mirror's `node_pubkey`. Not bytes a reader claims came from the mirror, but bytes the mirror cryptographically committed to.

- **Fresh.** Its execution proof commits to a recent Chia block header and to the challenge nonce, so it is a current statement, not an old one or a precomputed one.
- **Verifiably wrong.** The chunks fail to recompute the committed `root_hash`, or the execution proof fails against the committed `program_hash`, or the served version is stale past the grace window.

Anyone holding such an artifact submits it to slash the bond in full, taking a reporter's bounty with the remainder burned. A single honest reader can prove the fault on ciphertext alone, against on-chain hashes, so the path is cartel-proof: it needs no quorum, no committee, and no cooperation from other mirrors or validators. One correct artifact from one reader is sufficient, which is what makes correctness enforcement robust even if every other participant is hostile.

The fraud proof's purpose is to catch what the random audit can miss. A mirror could serve correct, signed proofs to the proposer when audited while serving wrong content to a particular reader between audits. The audit, sampling, might not catch this targeted misbehavior; the fraud proof does, because the targeted reader holds the mirror's own signed wrong response and can slash it. The two triggers are therefore complementary: the audit measures availability across the set, and the fraud proof punishes any provable act of serving falsehood, including the targeted kind the audit would not sample.

38. Un-fakeability

The fraud proof cannot be faked against an honest mirror, because an honest, current mirror signs only correct, current responses and therefore never produces an artifact that satisfies all three conditions. A reader attempting to frame an honest mirror fails on every available vector:

- **Forge the signature.** The reader has no signing key for the mirror and cannot produce a response validly signed under `node_pubkey`.
- **Pass a valid response off as failing.** Verification reruns on chain when the proof is submitted; a response that actually recomputes the root and verifies against the program hash passes the re-check, and the fraud proof is rejected.
- **Use bytes corrupted in transit.** A man-in-the-middle can corrupt a response, but corrupted bytes no longer carry the mirror's valid signature; the corrupted artifact fails the signature check and is inadmissible. A transit problem cannot frame the mirror, because the proof demands a valid signature over wrong content, which an attacker who can only corrupt cannot produce.
- **Replay an old correct response as stale.** An old response commits to an old Chia block, which proves the mirror served correctly *then*; it does not prove the mirror is serving stale content *now*. A staleness proof requires a response that is both fresh (recent block commitment) and stale (a version past the grace window), a combination only a genuinely misbehaving mirror produces.

The only mirror a reader can slash with a fraud proof is one that actually signed wrong or stale content and handed it over. The signature requirement defeats forgery and transit framing, and the freshness requirement defeats replay; together they reduce the admissible artifacts to exactly the set an honest mirror never creates.

39. Omission Is Not Slashable

Failing to receive content is not a fraud proof and cannot be made into one. Non-receipt has indistinguishable causes: the reader's own connection, a mirror briefly down, censorship or a partition in transit, or a reader simply lying about what it received. None of these is provable by a third party from a reader's claim of silence, and a protocol that slashed on such a claim would be slashing on an unfalsifiable assertion that any party could make against any mirror. The protocol therefore never slashes on a claim of non-receipt.

The gap that omission leaves is covered without readers, by two mechanisms. First, failover: every mirror is a full replica, so a reader that receives nothing from one mirror fetches from another of the capsule's N mirrors and is not harmed by a single dark or slow one. Second, the proposer audit: persistent unreachability is measured by the audit and punished through the leaky-bucket weight, across many independent proposers over many blocks, rather than asserted by any one reader. A mirror that is genuinely and persistently unavailable accumulates failure weight from the audit and is eventually slashed for liveness; a mirror that is merely unavailable to one reader at one moment is routed around. In neither case does a reader's experience of silence directly slash a mirror, and in both cases the protocol's response is sound.

Underneath all of this, a reader never has to trust a mirror in the first place. The read path verifies every response locally against the on-chain anchor before decrypting (Part 8), so a malicious mirror cannot make a reader accept wrong content: a wrong response fails the reader's own verification and is rejected (and, if signed, can be turned into a fraud proof), and a missing response is failed over. The worst a bad mirror can cost an honest reader is a wasted round-trip, never an accepted falsehood. The fraud proof is the punishment layer for provable correctness faults; it is not the mechanism that keeps a reader safe, and its deliberate silence on omission is therefore not a safety gap.

40. Validator Slashing and Appeals

A mirror's bond and a validator's stake are slashed under different rules, and reachability disagreement is never a validator offense. A validator's own stake is slashable only for Part 1's objective consensus offenses, provable from single-validator evidence: equivocation (proposing two conflicting blocks for a slot) and voting over a block known to be invalid, among the correlation-penalty offenses Part 1 defines. A proposer's audit behavior is not in this set. A proposer that misreports reachability is not committing a slashable consensus offense; it is contributing a (bounded, threshold-gated) failure increment that the rest of the protocol contains, as Section 27 describes.

DFSP adds no appeal process of its own beyond Part 1's. The fraud-proof path carries hard, self-verifying evidence and needs no appeal: the artifact either re-verifies on chain or it does not. The liveness-weight path is final on a deterministic accumulator whose only non-reproducible input, the reachability sequence, reflects a majority of proposers, so a wrongful liveness slash would require the same hostile majority that Part 1's own security model bounds; Part 1's optimistic fraud-proof appeal window (its bond-withdraw-delay window) provides the backstop against a wrongful slash that both protocols share. There is no DFSP-specific tribunal, because neither trigger needs one: correctness is settled by re-verification, and liveness is settled by deterministic recomputation under the chain's majority assumption.

41. Self-Healing

Either trigger, a liveness slash or a correctness fraud proof, forfeits the whole bond and removes the mirror entry. The immediate effect is that the capsule drops from N replicas to N–1. That under-replication is exactly the condition that the reward gradient (Section 31) resolves: an under-replicated capsule pays a full share to a new mirror within the cap, so an operator seeking reward homes a fresh mirror onto the slot, and the capsule returns to N. The same is true of a voluntary exit: an exiting mirror leaves a slot below N, which the gradient refills.

Replication therefore self-heals without any protocol component assigning replacements. The protocol does not detect under-replication and dispatch a mirror; it simply continues to pay the full share to mirrors of under-replicated capsules, and operators, maximizing their own reward, fill the gaps. A capsule loses a mirror to misbehavior, the slot becomes the most profitable place for a new mirror to go, and an operator takes it. The combination of full-bond slashing, which removes a faulty mirror cleanly, and the convergence gradient, which refills its slot, makes the hosting set continuously restore itself toward N healthy replicas per funded capsule as faults occur.

Part 8: Retrieval

42. The Read Path

Reading is free and verified end to end by the reader. It costs no `$DIG`, because hosting is funded by emission, not by retrieval. After resolving a name and reading two pieces of layer-2 state, the rest is an off-chain fetch with self-contained verification and local decryption. The six steps are:

1. **Resolve.** Look up the handle for the capsule's `store_id` and form the URN. A reader that already holds a URN skips this step entirely.
2. **Anchor.** Read the current `(root_hash, program_hash)` for the capsule from the capsule registry. This pair is the root of trust for everything that follows.
3. **Discover.** Read the capsule's mirror entries and their advertised addresses from the mirror registry.
4. **Fetch.** Request the resource by `retrieval_key` from any one mirror. The mirror returns ciphertext chunks with a merkle proof and an execution proof.
5. **Verify.** Check the merkle proof against `root_hash`, the execution proof against `program_hash`, and the response's freshness against a recent Chia block.
6. **Decrypt.** Derive the decryption key from the URN by HKDF and decrypt the verified chunks locally, checking the AEAD tag.

Steps 1 through 3 read on-chain state (the handle resolution and the two registry reads); steps 4 through 6 are an off-chain exchange with a single mirror plus local computation. The reader contacts one mirror, not a quorum, because every mirror is a full replica, and it verifies before it decrypts, so trust never rests on the mirror.

43. Reader Safety

Every step of the read path is checked locally against the on-chain anchor, so a reader never accepts a wrong or stale response. The anchor's `(root_hash, program_hash)` comes from consensus state the reader reads for itself; the mirror's response is checked against that anchor; and the decryption key comes from the URN the reader holds, not from anything the mirror provides. Trust rests in the anchor, not in any mirror.

This makes a malicious or faulty mirror harmless to correctness. A mirror that returns wrong content fails the reader's merkle or execution check and is rejected before decryption; if the wrong response was signed, the reader additionally holds a fraud proof and can slash the mirror (Section 37). A mirror that returns tampered ciphertext fails the AEAD tag check after decryption. A mirror that returns nothing is routed around: the reader fetches from another of the capsule's mirrors. In every case the reader either obtains verified-correct content or moves on; it never consumes a falsehood. The worst a bad mirror imposes on an honest reader is the latency of a wasted round-trip and a failover, which is the bound that lets the protocol decline to slash on omission (Section 39) without weakening the reader's guarantee.

44. The Client

The primary client is a browser extension. It runs the read path in a background web worker and registers the `dig://` URI scheme, so that a `dig://` reference in a page or a link resolves, fetches, verifies, and decrypts through the extension transparently to the application above it. The web worker keeps verification and decryption off the page's main thread, and the extension surfaces the result to the page only after the response has passed verification and the AEAD tag has checked, so the application never sees unverified bytes.

Registering a scheme rather than relying on a hosted resolver is what keeps the strong-trust path entirely client-side: the extension is the resolver, the verifier, and the decryptor, and it depends on no server to vouch for content. The on-chain anchor it checks against is read from the layer-2 state the client already follows, so the trust root of a `dig://` fetch is the same checkpoint-finalized state that secures the rest of the protocol.

45. The Gateway

A gateway relays requests between a reader and mirrors, and the protocol's blindness is designed so that a gateway in the path learns nothing it should not. The extension fetches through a gateway that carries only ciphertext and `retrieval_key` hashes and never sees the URN; provider blindness (Section 8) therefore holds with the gateway in the path exactly as it holds at the mirror, and the gateway is a dumb relay of opaque bytes. Because the extension still verifies and decrypts locally, a gateway cannot alter or forge content undetected: a tampered relay fails the reader's verification just as a tampered mirror would.

A reader without the extension may use a gateway that decrypts public capsules and serves them over ordinary `https://`. This is a weaker-trust path: the reader trusts the gateway to verify and decrypt on its behalf, rather than doing so itself, and so trusts the gateway not to serve falsehood. It exists for compatibility, to make public capsules reachable from an unmodified browser, and it trades the client-side trust guarantee for that reach. The strong path (extension, local verification) and the weak path (gateway-side verification over `https://`) coexist, and a reader chooses its trust model by choosing its client.

46. Versioning

A capsule's anchor advances as its content or module changes (Section 7), and the read path resolves versions against the anchor's current pair and the grace window. A **latest** request, one whose URN omits a pinned `root_hash`, accepts the current version or the immediately previous one during the update grace window, so that a read issued while a version transition is propagating to mirrors succeeds against either the new or the just-superseded version rather than failing. A **pinned** request, one whose URN includes a specific `root_hash`, requests exactly that version and accepts nothing else.

The grace window is the same window that governs the audit's version check (Section 24) and the capsule's lifecycle grace state (Section 7), so a mirror that is briefly serving the previous version during a transition is neither failed by the audit nor rejected by a latest reader within the window. Outside the window, serving a superseded version is staleness: the audit fails it, and a reader holding a signed stale-past-grace response can submit a fraud proof (Section 37). Versioning is thus consistent across the three places it appears, retrieval, audit, and lifecycle, all keyed to the same anchor and the same grace window.

Part 9: Consensus Integration

47. DFSP in the Block

DFSP rides entirely inside Part 1's existing block structure and block-application rule; it adds no new circuit, no new message type, and no new consensus round. Part 1's block body already carries categorized contents (transactions and the various deltas a block applies), and DFSP adds its categories to that body: an `audit_results` category carrying the proposer's signed proof bundles for the block's passers (Section 25), and the registry-delta categories that record mirror registrations, updates, slashes, exits, withdrawals, and capsule lifecycle transitions for the block.

Part 1's `DfspBlockApply` is the rule that applies these. As specified in Section 26, on every validator it reconstructs the seed-forced sample, re-verifies every claimed pass from the block alone, credits reward to verified passers, advances failure weight on the failers, applies the lifecycle and registry deltas, and requires the block header's committed capsule-registry and mirror-registry roots to match the recomputed state, rejecting the block on any mismatch. Deterministic per-epoch operations, in the manner of Part 1's epoch-close procedure, advance lifecycle states and any periodic bookkeeping at epoch boundaries, again identically on every node. The block-application rule and the epoch-close procedure are where all DFSP state transitions happen, and both are ordinary parts of the layer-2's existing validation, extended with DFSP's categories rather than wrapped in anything new.

48. The Checkpoint

DFSP state is finalized to Chia on the same path as consensus state, with the same cryptographic strength. The capsule-registry and mirror-registry roots ride in every block header alongside the consensus state root, and they are covered by the same BLS attestation the validators produce over the block. At epoch close, Part 1's checkpoint block commits, among the header-Merkle root, the new state root, the validator root, and the exit-ledger root, the DFSP registry roots as part of the checkpoint digest. That digest is proven and signed through Part 1's checkpoint circuit and submitted to Chia, where confirmation to the configured depth hard-finalizes the epoch.

The consequence is that a finalized DFSP registry state is as settled as a finalized account balance: both are committed in the same checkpoint, proven by the same Groth16 proof, signed by the same BLS aggregate, and anchored by the same Chia confirmation. A mirror's accrued balance, its failure weight, the hosting set, and every capsule's lifecycle and current hashes are all checkpoint-finalized state, not soft or off-chain data. There is no separate finalization for storage; storage finality is consensus finality.

49. No New Circuit

DFSP adds no new Groth16 circuit and no new trust anchor. The checkpoint that finalizes DFSP state is proven by Part 1's existing checkpoint circuit, built from the same membership, majority, and aggregation gadgets, verified against the same verification key produced by Part 1's MPC ceremony. Because DFSP's roots fold into the checkpoint digest that the existing circuit already commits to, finalizing storage state requires no change to the circuit, no new public inputs beyond those the digest carries, and no new trusted setup. Deploying DFSP adds no ceremony and introduces no new cryptographic assumption.

DFSP's parameters live in the same place as Part 1's. The capsule size, the replica target N, the audit sample size, the storage role-share, the failure increment, the failure half-life, the threshold, the grace window, and the unbonding delay are entries in the shared parameter snapshot that Part 1's `ParameterRegistry` holds, on the extension surface Part 1 reserves for protocols built on the layer 2. They update through Part 1's governance circuit, the same shape as the checkpoint and collateral-update paths, so a DFSP parameter change is a Category-A action with the same threshold, strength, and on-chain cost as any other. DFSP does not introduce its own registry, its own governance surface, or its own circuit; it extends Part 1's.

50. What Stays Part 1

Everything DFSP does not explicitly add stays exactly as Part 1 defines it. Proposer selection is unchanged: DFSP uses the proposer Part 1 already selects per slot, with Part 1's fallback sequence, and adds only the audit duty to that role. The block format is unchanged except for the additional body categories. The BLS attestation, the three-tier finality (proposed, soft-final by quorum, hard-final by Chia confirmation), and the epoch lifecycle are unchanged. Validator slashing is unchanged: the consensus offenses and their correlation penalties are Part 1's, and DFSP adds none. The five-role subsidy split is unchanged in mechanism; DFSP only defines what the storage role does with its share. The five L1 puzzles, the network coin, the checkpoint singleton, the parameter registry, and the rest, are unchanged.

This is the point of building DFSP as a capability rather than a chain. The consensus, the finalization, the governance, and the trust anchor are all Part 1's, carrying Part 1's security arguments, and DFSP inherits them rather than restating or weakening them. A reviewer who has accepted Part 1's consensus does not need to re-examine consensus to evaluate DFSP; the new surface is the capsule, the audit, the reward rule, the slashing rule, and the storage state, and everything those rest on is the chain that already exists.

Part 10: Economics, Bootstrap, and Governance

51. The Economic Model

The economic model has three calibrated quantities, the reward, the bond, and the burns, and one emergent quantity, the supply crossover. The reward per mirror per block is the storage role-share divided among that block's verified passers; over time, a mirror's income is its expected share of the role-share scaled by its pass rate (Part 6). The model is viable when the diluted per-mirror reward, at the equilibrium mirror count, covers an operator's marginal cost of storing and serving a 100-megabyte capsule across the audit cadence. The storage role-share is set, and declines on schedule, to keep N mirrors per capsule viable at expected storage cost and `$DIG` price, and the automatic dilution (Section 30) means the protocol does not need to track cost or price directly: it holds the share fixed and lets the mirror count find the level at which reward meets cost.

The bond is calibrated against the reward, not against the value of the data. It is set above the reward a mirror can earn before a sustained outage would cross the failure threshold, so that going dark is loss-making: a mirror that stops serving forfeits a bond worth more than the reward it could have collected in the interval before slashing. This makes honest service the dominant strategy without the protocol needing to price the hosted content, which it cannot see (Section 6). The burns are calibrated as gates and sinks rather than revenue: the handle fee is floored at the cost of the hosting it gates (Section 15), and the other sinks track activity. The crossover between net inflation and net deflation is then emergent, set by handle count, transaction volume, and capsule churn against the scheduled emission, and the protocol targets no particular point on that curve.

52. Bootstrap

DFSP must solve a chicken-and-egg problem at launch: hosting is paid in `$DIG`, but mirrors must stake `$DIG` to host, so the first mirrors need `$DIG` before the hosting reward has paid anyone. The solution is the declining storage role-share (Section 13.3), funded entirely by emission with no premine and no treasury. The share is set high at genesis, so that early mirrors earn enough to cover the cost of seeding the network's first capsules, and it declines on a fixed public schedule to a steady-state value as the network matures. Activation is set at a block height by which enough `$DIG` has been emitted for a genesis set of mirrors to stake, and from that height the audit, reward, and slash machinery runs.

Early hosting is bootstrapped by posting seed capsules, foundational datasets and the assets they reference, which the elevated early share makes profitable to mirror, and by sponsored onboarding, in which a publisher funds handles to draw mirrors onto its capsules. Because the share declines on schedule regardless of conditions, the elevated reward is explicitly temporary: mirrors that enter during the high-share period must develop a position that remains viable as the share falls to steady state, or exit. There is no governance lever to halt the decline on

the basis of network conditions, which prevents the bootstrap subsidy from becoming a permanent entitlement. The absence of a premine or treasury means the bootstrap is paid by the same emission that pays steady-state hosting, just front-loaded by the schedule.

53. Parameters

The protocol's tunable parameters live in the shared parameter snapshot and update through Category-A governance (Section 54). Structural constants that re-denominate existing capsules, principally the capsule size, are baked at activation rather than left mutable.

Parameter	Role	Class
Capsule size (100 MB)	The fixed unit of hosting	Structural constant, set at activation
Replica target N	Mirrors per capsule; the reward cap per store per block	Category A
Audit sample size	Entries a proposer audits per block; bounded by the slot window	Category A
Storage role-share	Fraction of the block subsidy that funds hosting; the whole hosting emission and its cap	Category A, on a declining bootstrap schedule
Failure increment	Weight added to a mirror's bucket per failed audit	Category A
Failure half-life	Decay half-life of the failure weight; set above the audit gap	Category A
Slash threshold	Failure weight at which the bond is slashed in full	Category A
Version grace window	Span in which the previous version is still served and accepted	Category A
Unbonding delay	Delay before an exiting bond releases; covers pending slashes	Category A (mirrors Part 1's bond-release window)
Beacon anchor depth	Chia confirmation depth at which the audit beacon is taken	Category A

The failure increment, half-life, and threshold are calibrated together to set the tolerated failure rate (Section 35); the storage role-share and N are calibrated together to set viability and redundancy (Section 51); and the grace window is shared across retrieval, audit, and lifecycle (Section 46). A parameter change that loosens slashing or alters the share takes effect through the same proven, signed, Chia-anchored governance action as any other Category-A change, so the storage protocol's parameters carry the same governance strength as consensus parameters.

54. Governance

DFSP's parameters are governed by Part 1's Category-A process and by nothing else. A Category-A change is a single L1 spend that consumes the `ParameterRegistry` singleton and provides a 32-byte digest committing to the parameter change, a Groth16 proof against the current validator Merkle root that a supermajority of validators ($2k > n$) signed, and a BLS aggregate signature over the digest. This is the same circuit shape, the same threshold, and the same on-chain cost as a checkpoint update; a DFSP parameter change is computationally identical to DIG finality. DFSP adds its parameters to the same snapshot and changes them through the same path, with no separate governance surface of its own.

The limits on what governance can do are Part 1's, and they bound DFSP as well. Governance cannot run arbitrary code, change puzzle logic, or alter the wire format; those are Category-B client-release changes, out of the on-chain process. Governance cannot de-list a validator or a mirror, mint `$DIG`, or create a treasury. It can only mutate the agreed parameter snapshot. For DFSP this means governance can retune the audit sample, the slashing parameters, the share, N, and the windows, within the protocol's fixed structure, but it cannot rewrite the audit, redefine the capsule, seize bonds outside the slashing rules, or redirect emission to a treasury. The storage protocol is as constrained by the governance limits as consensus is, because it is governed by the same mechanism.

Part 11: Security Analysis

55. Threat Model

The combined system presents two independent security budgets, and an attacker must defeat them separately. The **consensus budget** is Part 1's: subverting block production, finalization, or governance requires a stake majority of the validator set, secured by XCH collateral and the correlation-penalty slashing of the consensus offenses. The **data-layer budget** is DFSP's: making a specific funded capsule unavailable, or causing a falsehood to be accepted, requires defeating that capsule's mirrors in a way that produces no slashable evidence reaching the chain. Because mirrors stake `$DIG` and validators stake XCH, and the two are slashed independently, neither budget can be spent to buy the other.

An attacker against the data layer has a small set of options, each addressed below: influence which mirrors are audited or rewarded (the seed and selection), manufacture failures against honest mirrors (the audit's trust surface and the slashing rule), serve falsehood to readers without being caught (the read path and the fraud proof), or deny a capsule's availability without accumulating slashable weight (replication and the audit). The analysis shows each is either impossible, bounded to a single block, or self-limiting, under the honest-majority-of-proposers

assumption DFSP inherits.

56. Security Properties

The protocol's guarantees and the mechanism each rests on:

Property	Rests on
Content genuineness	Merkle inclusion against the on-chain `root_hash`
Correct, current service	Execution proof against the on-chain `program_hash`, plus version match
Freshness	Execution proof committed to a recent Chia block header
Possession of the full capsule	Seed-derived random-offset challenge across the whole 100 MB
Provider blindness	URN-derived keys the provider never holds
Cold-content availability	Reward for audited holding, independent of read demand
Collateral at risk	A single bond coin a slash spends in full, with no protocol exit around it
Bounded audit load	One proposer probes per block; the set re-verifies offline
Liveness enforcement	Decaying failure weight crossing a threshold; sustained and multi-proposer
No reward grinding	Audit seed mixes the exogenous Chia beacon, the parent hash, and the registry root
Honest audit result	The proposer cannot fabricate a pass or bias selection; the set re-verifies both
Cartel-proof correctness	A single reader slashes a signed wrong response on ciphertext, with no quorum
Bounded inflation	The storage role-share is the whole hosting emission and its own cap

57. Attack Analysis

Proposer reward grinding. A proposer cannot shape the sample it audits, because the seed mixes the exogenous Chia beacon, the prior block's hash, and the registry root, none of which the current proposer controls (Section 22). The residual, a prior proposer grinding its block hash to nudge the next sample, costs a full block grind, helps only a colluding successor, and can shift the sample only toward mirrors that genuinely serve, since a fabricated pass is independently impossible. It is bounded by the same majority assumption that secures the chain.

Manufactured failures. A dishonest proposer can falsely mark a serving mirror unreachable, but the lie is bounded to one block's reward and one failure-weight increment, and a single increment never slashes (Section 27). A slash requires the weight to cross a threshold built from many increments by many independent proposers over many blocks, so manufacturing a slash against an honest mirror requires a sustained, coordinated majority misreporting reachability, which is the chain's own residual risk and is additionally covered by Part 1's appeal window.

Serving falsehood. A mirror cannot make a reader accept wrong content, because the reader verifies every response against the on-chain anchor before decrypting (Section 43). A mirror that serves a signed wrong or stale response hands the reader a fraud proof that slashes it in full (Section 37), and the fraud proof cannot be faked against an honest mirror, since it requires a fresh, validly-signed, verifiably-wrong artifact that an honest mirror never produces and an attacker cannot forge, replay, or synthesize from corrupted transit bytes (Section 38).

Targeted withholding and selective service. A mirror that passes audits while withholding from a specific reader commits omission, which is not slashable because non-receipt is unprovable (Section 39). The attack is bounded and unprofitable: the reader fails over to another of the N replicas and is not harmed, the capsule stays available, and the withholding mirror gains nothing. A mirror that withholds from auditors too, rather than selectively, accumulates failure weight and is slashed for liveness.

Transit attacks and replay. A man-in-the-middle can corrupt or drop responses but cannot frame a mirror: corrupted bytes lose the mirror's valid signature and are inadmissible as a fraud proof, and a dropped response is omission, handled by failover. Replay is defeated by freshness: every audit response and every fraud-proof artifact commits to a recent Chia block and a challenge nonce, so an old response cannot be passed off as a current one in either direction (Sections 24, 38).

Eclipse and sybil. An attacker that runs many mirrors of a capsule cannot earn beyond N per block on it, because the per-block split is capped at N per `store_id` (Section 30), so sybil mirrors of one capsule do not multiply reward. An attacker that eclipses a single mirror's view of the network attacks consensus, not DFSP, and is bounded by Part 1's networking and the fact that audit seeds derive from finalized state and the Chia beacon rather than from peer gossip.

Exit to dodge a slash. A mirror cannot exit to reclaim its bond ahead of a justified slash, because the exit path releases collateral only after an unbonding delay during which accumulated failure weight crosses and any in-flight fraud proof lands (Section 18). The collateral stays seizable throughout the delay.

58. Residual Risks

Three residual risks remain, and the protocol states them plainly rather than claiming to eliminate them. First, a **hostile majority of proposers** can withhold rewards from serving mirrors or manufacture failure weight against them, because reachability is the one fact the set takes on trust. This is not a new risk: it is the chain's own honest-majority assumption, and a proposer majority that abuses reachability is, in aggregate, a dishonest validator majority, which Part 1's security model already bounds and Part 1's appeal window backstops. DFSP adds no trust assumption beyond this.

Second, **throughput is not guaranteed**. A passing audit and the reward it earns prove that a mirror was reachable and serving correct, current content at the moment of audit; they do not guarantee a given reader's bandwidth or latency at an arbitrary later moment. The protocol guarantees availability in the sense of audited holding and single-mirror sufficiency across N replicas, not a service-level bandwidth to every reader. A reader's recourse for a slow or selectively-unavailable mirror is failover, not a protocol remedy.

Third, **DFSP depends on the layer 2**. Its state finalizes through Part 1's checkpoint to Chia, so DFSP's liveness and finality are bounded by Part 1's, which are in turn bounded by Chia's. A halt or a deep reorg at a lower layer propagates upward. This is the cost of inheriting a security budget rather than minting one: DFSP is exactly as live and as final as the chain it is built on, no more and no less.

Part 12: Conclusion

59. Status and Roadmap

DFSP is specified as a capability of an already-running DIG Layer 2. It activates at a block height, from which `DfspBlockApply` audits, rewards, and slashes within ordinary block validation, and it requires no new circuit, no new ceremony, and no new trust anchor. The design locks the capsule as the unit, the proposer audit as the enforcement mechanism, the leaky-bucket weight and the reader fraud proof as the two slashing triggers, the storage role-share as the bounded hosting emission, and the coin-versus-state line as the architectural boundary. What remains ahead of activation is calibration and client maturation: setting the slashing parameters against measured audit cadence and operational noise, setting the storage share and its decline against expected cost and price, and hardening the browser extension, the gateway, and the operator tooling.

The protocol is deliberately small. It adds a storage market to a consensus chain by reusing the chain's proposer, subsidy, checkpoint, and trust anchor, and by confining its own novelty to five things: the capsule, the audit, the reward rule, the slashing rule, and the storage state. Everything that secures it is the chain that already exists, and everything it adds is verifiable from the block and the on-chain anchor. A storage network in which a capsule is hosted because it is funded, proven held under audit, and served under collateral, is the result.

60. References

- DIG Network, Part 1: Consensus and the Chia-Anchored Layer 2. Proposer selection, three-tier finality, the epoch lifecycle, the five-role block subsidy, the four consensus offenses and correlation-penalty slashing, the five L1 puzzles, the checkpoint circuit and MPC ceremony, the ParameterRegistry and Category-A governance.
- DIG Network, Part 3: The Digstore Artifact. The content-addressable WASM store, URN-derived encryption and provider blindness, the interleaved chunk pool with filler and decoys (blinding), execution proofs and freshness, and the hardware-attested proof alternative.
- DIP-0001: DIG Uniform Resource Names. The `urn:dig:chia:{store_id}[:{root_hash}][/{resource_key}]` addressing scheme.
- CHIP-35: Singleton store lineage. The lineage-coin pattern underlying the `store_lineage_coin` capsule anchor.
- Chia Network: the coinset model, CLVM, BLS signatures, singletons, and proof of space and time, the L1 substrate the layer 2 anchors to and the source of the audit beacon.