Early draft, version 0.18.0, July 2026

TAPEDRIVE: Fast Decentralized Storage with Replay

Spool Labs
info@tape.network

This paper is subject to the Legal Disclaimer in Appendix C.

Abstract

Decentralized storage networks have usually been built around one defining advantage: object permanence, surviving extreme failures, or strong incentives for serving data. Each advanced the field, but applications still had to adapt their data or workflows to the protocol beneath it. Cloud storage succeeded by making those details disappear, turning placement, redundancy, and recovery into properties of the service. What decentralized storage still struggles to do is get out of the application's way. Until convenience becomes part of the protocol itself, decentralized storage will remain specialized infrastructure rather than everyday storage.

TAPEDRIVE is a high-performance decentralized object store that makes convenience its superpower. It does so through three mechanisms. **Tape Replay** derives shared protocol state from activity recorded on Solana. **Tape Spooler** assigns storage deterministically as network capacity changes. **Tape Slicer** provides verifiable encoding and bandwidth-optimal repair. Together, these mechanisms make decentralized storage behave like ordinary object storage, whether it runs on one machine, on many, or as part of TAPENET, its open network.

1 Introduction

Everyday applications expect data storage to be *boring*. They store files by name, retrieve them, replace or delete, and choose how long to keep them. Behind those simple operations, a storage provider handles physical placement, redundancy, restoration, and capacity. Applications do not negotiate with disks or maintain a relationship with every machine that holds their data.

Centralized cloud storage meets this expectation so well that applications have stopped noticing it. One request stores an object of any size, and the same request works for a text file or a video archive. A team can build and operate an entire product without knowing where its data physically sits. Storage this easy fades into the platform, and for most applications there has been no reason to look further.

Yet decentralized storage offers guarantees that centralized cloud services do not. Storage can be independently verified, responsibility can be spread across operators, and data can remain reachable without placing one provider in control.

The economics point the same way. In an open network, prices come from competition among operators rather than from one provider's list. The difference is largest for reads, which the cloud meters at rates far above the cost of delivery because egress fees make leaving expensive [15, 38, 39]. Nor is performance the barrier it once was. Measurements of deployed networks show that the gap to centralized services is set by design choices, not by decentralization itself [40]. The spread between cloud prices and dedicated servers leaves room for applications to pay less while operators still earn from hardware they already run.

One emerging workload makes this distinction especially important. AI *agents* change how and where data is accessed. As agents move across devices, runtimes, and service providers, fast access near the edge becomes more important. Keypairs give them portable ownership and access across environments. An open network gives agents a common storage layer where they can share persistent state, exchange data, and integrate directly with one another.

These advantages should have made decentralized storage everyday infrastructure. *Why have they not?* Every design to date leaves part of the storage mechanism in the application's hands. Adopting one is a system design decision rather than a storage decision. The application must reshape itself around the protocol, and the parts it fails to manage surface as reliability problems or lost data.

That pattern runs through the field's evolution. IPFS decentralized discovery and transfer but left publishers responsible for retaining, or pinning, their data [1]. Arweave made permanent retention a network property, but deletion is impossible and every write prepays for centuries of storage [2]. Filecoin introduced an open market for storage, but durable writes depend on forming and renewing deals with providers [3]. Each system solved a real problem, and each left some operational burden with the application.

The newest designs, Walrus and Shelby, come closest to removing that burden. Both borrow an existing blockchain for coordination and use erasure coding to cut storage overhead while keeping data verifiable and fault tolerant [4, 5]. For large objects, the result starts to resemble cloud storage. The machinery has a fixed cost *per object*, though. A write pays for encoding, proofs, and certification whether it stores an archive or a thumbnail, and

System	Storage design				Coordination	
	Replication overhead	Native small objects	Efficient repair	Program-composed writes	Blockchain	Mechanism
AWS S3	est. $1.4\times$	✓	✓	×	—	Centralized
GCS	est. $1.4\times$	✓	✓	×	—	Centralized
Arweave	$15\times$	✓	×	×	—	PoW
Filecoin	3–$6\times$	×	×	×	—	PoSt
Walrus	$4.5\times$	×	✓	×	Sui	PoS
Shelby	$< 2\times$	×	✓	×	Aptos	PoS
TAPEDRIVE	$\sim 2\times$[1]	✓	✓	✓	Solana	PoS

Table 1: Storage-system design comparison [2, 3, 4, 5].

in everyday workloads most objects are small. Applications can escape the fee by bundling small objects into larger ones, but then packing and tracking those bundles becomes their job.

Every design so far makes storage something the application has to notice again. Cloud storage *never* does. It accepts data as applications produce it and hides placement, redundancy, repair, and lifecycle behind one interface. That is why it remains the default.

Convenience is therefore not a surface feature. It depends on *where* the protocol places responsibility: a decentralized object store cannot behave like cloud storage while passing retention, placement, or coding decisions back to the application.

A practical alternative should do the same across the useful range of object sizes. Applications should be able to write, replace, delete, and choose how long to store data without giving up durability, independent verification, or protection from any one provider suppressing it.

The supply side should be just as simple. New providers should add usable capacity without applications noticing, and operator rewards should come from the capacity that applications reserve in advance, at prices that compete with the cloud.

Storage becomes *boring* only when the application stops noticing it.

Introducing TAPEDRIVE

TAPEDRIVE is a decentralized object store in which applications reserve storage for a chosen lifetime and write data without selecting providers, choosing a redundancy scheme, or coordinating repair. Storage nodes hold and serve the payload bytes. Solana serves as the control plane, fixing which storage operations are final and which nodes carry them. The same software *can* run on one machine, on many machines an operator controls, or as part of **TAPENET**, the open network anyone may use or provide capacity to.

Applications interact with TAPEDRIVE as storage rather than as a market for storage. The same interface accepts small objects, large objects, and streams, while the protocol selects the appropriate storage path. Applications can replace or delete objects and choose how long to retain them.

This places TAPEDRIVE between cloud object storage and the newest decentralized networks, pairing the workload flexibility of S3 and GCS with verifiable storage, distributed responsibility, and efficient recovery. Table 1 summarizes that position.

Blockchains and storage networks solve different problems. A blockchain keeps shared state consistent by having validators agree on an ordered sequence of transitions [11]. Replicating every payload across validators adds copies rather than usable capacity. A storage network divides bytes among providers so that added nodes increase capacity, but it still needs a control plane that records writes, storage lifetimes, and responsibility. Applications that lack one often treat an object store as a log; the log is durable, and ordering, agreement, and programmatic control stay with the application.

TAPEDRIVE uses Solana as that control plane rather than operating a second consensus network. Solana's low transaction cost and fast finality allow storage operations to remain in the live transaction path, including frequent small writes [6, 7]. Because those operations execute as Solana instructions, applications can compose storage actions atomically with other on-chain activity and replay the same finalized operations into their own state, as TAPEDRIVE's storage nodes do.

TAPEDRIVE's storage nodes, not Solana, store, restore, and serve the payload bytes.

[1] Recommended Clay profiles place coded storage between $2\times$ and $3\times$ the source data before metadata and padding. TAPENET is expected to run $20/7 \approx 2.86\times$ for a wider recovery margin; local deployments may run as low as $1.4\times$.

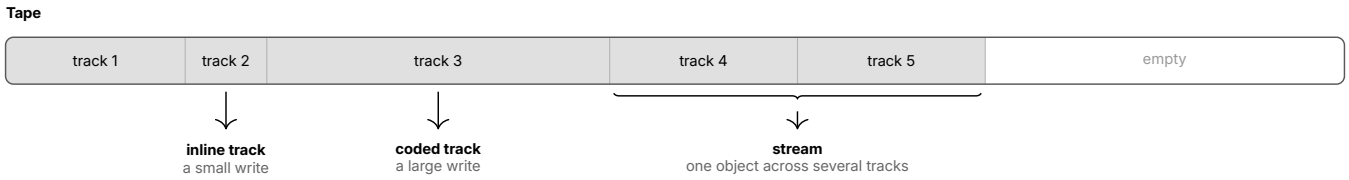


Figure 1: One tape. Writes append as tracks: inline for small writes, coded for large ones, a stream when one object spans several.

2 Models and Definitions

TAPEDRIVE operates in **epochs**. For each epoch e, the **committee** is the active set of staked storage nodes. The epoch also fixes one mapping from logical storage positions to their physical owners. Committee membership and ownership may change between epochs, but remain fixed while an epoch is active.

Logical and physical storage

A **tape** is a reservation of capacity over a chosen number of epochs. Data is appended to a tape as **tracks**. A track cannot be changed in place, but it can be deleted. Figure 1 shows one tape.

TAPEDRIVE organizes physical responsibility through **spools**. A spool is a logical position of stored data whose physical owner may change. A **spool group** G contains n positions assigned to n distinct registered storage nodes, and one node may own at most one position in the same group.

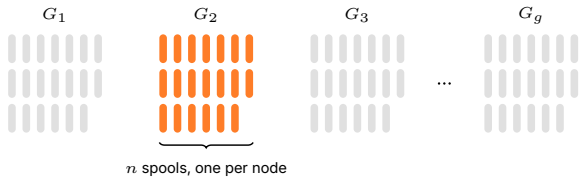


Figure 2: A track is assigned to one spool group.

Every track x is assigned to one group $G(x)$. The track is sliced across that group's spools, sharing the storage and service load among their owners. Its write and read protocols, and any later restoration work, involve only that group. A spool group's width n remains fixed within a protocol profile. Figure 2 shows the structure of a spool group.

Consensus layer

Solana atomically executes TAPEDRIVE control operations and fixes their order at finality [6, 7]. Honest storage nodes apply only successfully executed, finalized operations in that order and derive the same TAPEDRIVE state.

TAPEDRIVE inherits execution, authorization, ordering, and finality. It does not inherit guaranteed transaction inclusion, permanent transaction history, unbiased randomness, payload retention, or read availability.

That state identifies the live tracks, their commitments, the spool group to which each track is assigned, and the current owner of each spool position.

The control plane is a Solana runtime in every deployment. On TAPENET it is the public cluster. On one machine or on infrastructure one operator controls, it is a validator the operator runs, and everything TAPEDRIVE inherits from Solana then rests on that operator.

Cryptographic assumptions

TAPEDRIVE assumes a collision-resistant hash function, written H, binding Merkle commitments [14], and unforgeable BLS signatures [8]. An aggregate signature binds the signed message, epoch, group, and signer positions. These primitives let a participant verify signatures and detect bytes that differ from a registered commitment.

Adversary assumptions

For one group and epoch, let f be the maximum number of Byzantine positions. The network is asynchronous: messages may be delayed or reordered without a known bound. Safety does not depend on delivery time. A protocol step that waits for responses can complete only if enough honest participants reply before its epoch or applicable deadline ends. A group decision additionally requires valid signatures from the required threshold of owners.

The mechanism sections use q for a group agreement threshold and k for a reconstruction threshold. Decisions over values that every honest node recomputes from the same finalized state use the smaller attestation threshold $f + 1$.

3 Tape Replay

Solana orders and finalizes TAPEDRIVE operations. **Tape Replay** turns that ordered stream into the local protocol state storage nodes use to make decisions. That state gives every node a consistent view of which tapes and tracks exist, what each track commits to, which spool group stores it, and which node hosts each spool.

Strawman Design

The simplest mechanism would keep no local records. A storage node could instead fetch each decision's inputs from Solana as needed. This makes every decision depend on ad-hoc RPC requests and creates a consistency problem: the chain may advance between correlated fetches, returning answers from different blocks. Reads can be pinned to a common block, but that discipline must hold across every fetch, and each lapse is a silent bug.

A second option is to act on confirmed blocks instead of waiting for finality. Confirmation is faster, but a confirmed fork may still be replaced, and any node that applied its operations now holds tracks and assignments the history never contained. Honest nodes would stop agreeing on what the network stores.

Both are solved by advancing local protocol state only from finalized Solana blocks.

Replay Design

A block ingestor follows Solana block production one block at a time. From each successfully executed transaction, it parses TAPEDRIVE instructions and the program-data entries emitted during their execution, then merges them into compact replay records. These records are appended to the current epoch's replay log. The log becomes the local source of truth for protocol decisions, eliminating ad-hoc RPC state queries. Replay applies a record to protocol state only after its containing block is finalized. Figure 3 shows the replay log accumulating records from a window of recent Solana blocks.

Solana finality gives every node the same ordered record stream. Given the same prior state and deterministic transition rules, storage nodes that replay the same records apply the same changes in the same order and therefore reach the same state, following the state-machine replication pattern [11].

The replay log contains the metadata needed to rebuild TAPEDRIVE state, but not the payload bytes it describes. Storage nodes in the assigned spool groups store and serve those bytes. This separation lets later mechanisms preserve the common history and derive new spool assignments without treating Solana or the replay log as payload storage.

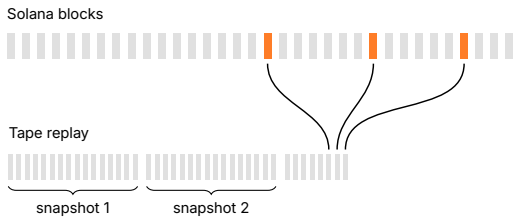


Figure 3: A window of recent Solana blocks distilled into the replay log. Earlier epochs' records are serialized into snapshots; the newest records are not yet in one.

4 Snapshots

A new or lagging node needs a verifiable way to reconstruct TAPEDRIVE state. It could begin with TAPEDRIVE's first Solana transaction and replay every later operation. That approach becomes increasingly impractical as the network ages, and it depends on historical transaction data that Solana does not promise to preserve forever.

An existing node could instead give the newcomer its current metadata database. That is fast, but requires the newcomer to trust that the database is complete and current. A stale or malicious node could omit records or present state derived from operations the network never accepted. Asking several nodes does not solve the problem unless the newcomer has a canonical commitment against which to verify their answers.

The replay log supplies the basis for that commitment. During each epoch, every committee member observes the same records in the same order and can serialize them into the same metadata log. TAPEDRIVE calls this per-epoch log a **snapshot** and preserves it in a dedicated snapshot tape.

A snapshot contains the records needed to rebuild protocol state, including tape and track metadata, object listings, committee information, and group assignments. It does not contain user payload bytes.

TAPEDRIVE uses its own tape mechanism to preserve the metadata from which its state can be reconstructed.

Strawman Design

The most direct design would treat the snapshot like an ordinary track. Ordinary tracks are assigned to one spool group and encoded across the nodes responsible for that group. Applying the same rule to a snapshot would make one group carry a protocol-wide responsibility while the rest of the committee played no part in storing it.

That asymmetry introduces a special role into an otherwise uniform protocol. The network would need to select the snapshot group, account for its additional work, and decide how that

responsibility changes between epochs. Rotating the role would move the imbalance rather than remove it.

Snapshot maintenance is simpler when every committee member performs the same role. Every node can derive the same snapshot, participate in agreement under the same rules, and store an equal share of its encoded representation. No designated snapshot operator or special group is required.

Snapshot Design

To distribute snapshot responsibility evenly across the committee, TAPEDRIVE encodes each snapshot in two stages. Tape Slicer later explains the underlying erasure-coding machinery in more detail; here, only the layout and its reconstruction thresholds are needed. Note that this encoding is unique to snapshots, where we want every node to participate equally, ordinary tracks use only the inner encoding mechanism.

Let g be the number of live spool groups. The outer code produces g equal-sized group symbols, of which any $\lceil g/3 \rceil$ recover the snapshot log. Each group receives one symbol. That symbol is then coded across the group's spools so that any k valid slices recover it. The outer stage distributes the snapshot across every group, while the inner stage distributes each group's share across its participating nodes.

The construction gives every group the same aggregate responsibility. Every member independently derives its encoded output and the same commitment; there is no separate party that constructs or assigns the snapshot. Each group votes on the snapshot commitment independently, and because honest members derive the same commitment, $f+1$ signatures from a group attest it. The snapshot vote establishes which commitment the network accepts.

TAPEDRIVE produces one snapshot tape for each epoch, containing a track for each spool group that participated in the snapshot. The tape commitment can be found on-chain and its address is derived from the epoch number. A new node can process the snapshot history directly in chronological order by finding the snapshot state on-chain. It begins with the first snapshot and continues one epoch at a time through the newest. For each epoch, it recovers the replay log, verifies it against the on-chain commitment, and applies its records through the same replay engine used for live blocks.

A lagging node follows the same process beginning with the first epoch missing from its local state. After applying the newest snapshot, it resumes finalized Solana block ingestion from that snapshot's ending slot. The node therefore reconstructs state from compact TAPEDRIVE records without trusting a centralized server or fetching and parsing every historical Solana block.

Snapshot replay reconstructs knowledge of which data exists and which groups are responsible for it. It does not recover the user payloads themselves. Those bytes remain with their assigned spool groups and are obtained through the ordinary storage path.

As the snapshot history grows, the protocol may introduce **checkpoint snapshots** that consolidate the effects of all prior snapshots into one new starting point and prune metadata no longer needed to reconstruct current state. Once a checkpoint is canonical and durably available, the earlier snapshot tapes it covers can be dropped, bounding the history a new node must retrieve and replay. The checkpoint format, pruning rules, and activation process are outside the scope of this paper.

5 Tape Spooler

TAPEDRIVE stores objects as one or more tracks on a tape. A tape is the on-chain commitment to the track data. The track data itself is erasure coded into slices, and each slice is sent to the node responsible for its spool during that epoch.

There are many tracks and many slices on the storage network, so we organize them into containers called spools. A slice maps directly to the spool using the index that was used to encode it. For example, if a track is sliced into 10 slices, slice 0 goes to spool 0 in the group, slice 1 goes to spool 1, and so on. Put another way, a single spool is a container for many slices from many tracks, and a single track's slices are spread across many spools.

We group spools into spool groups. A track is sliced into as many slices as there are spools within the group. All spool groups on the network have the same number of spools and the network grows by adding additional groups.

The slice-to-spool mapping is constant. What changes over time is the spool-to-node mapping. The **spool owner** is the physical node responsible for storing a spool and carrying its slices during an epoch.

From one epoch to the next, nodes may join, leave, or change capacity. The network must update the spool-to-node mapping to account for these changes. This is the role of the **Tape Spooler**. It computes the next epoch's mapping while preserving existing placements wherever possible to reduce unnecessary data movement.

Strawman Design

The simplest design is to leave every spool with its initial node mapping. When a node leaves, its spools remain assigned to a node that no longer serves them. New nodes cannot take over because nothing decides which spools they should receive. As departures accumulate, a group eventually loses enough owners that its tracks become unrecoverable.

Cloud storage systems solve this with a centralized placement service that maps logical storage to physical machines and updates that map as machines join and leave. TAPEDRIVE needs the same assignment function without depending on a central service.

Spooler Design

When a track is registered, a Solana program assigns it pseudorandomly to a spool group. The track stays in that group until it is deleted or its tape expires. Tape Spooler may change the nodes that are responsible for its spools between epochs, but it tries to keep existing owners wherever stake and capacity allow. Data therefore moves only for spools that move to a new node. Figure 4 illustrates one group's handoff between epochs.

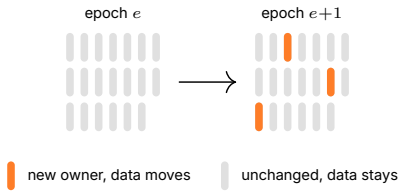


Figure 4: Spool owners across one epoch boundary. Most spools keep their owner; data moves only for spools that receive new owners.

A committee is the set of nodes with active stake admitted to store and serve data for an epoch. Membership is renewed each epoch. Any registered node with active stake may volunteer for the next committee before the current epoch closes. If more nodes volunteer than the committee can hold, those with the most stake are admitted. A node excluded by an eviction vote cannot join. Once the epoch closes, committee membership is fixed for the next epoch.

At the end of an epoch, the current committee plans the next epoch's spool assignments. Each node independently computes an assignment from the same Tape Replay state. To do this, each node excludes tracks that will be expired or invalid in the next epoch, determines the active tracks in each spool, and calculates the bytes the spool's next owner must carry. The candidate assignment also considers the stake weight of nodes admitted to the next committee, their declared capacity, and current spool ownership. These inputs are common, so honest nodes running Tape Spooler derive the same complete assignment.

The computation happens off chain because assigning all spools may require more data and work than Solana would allow within a transaction. It also avoids compute limits restricting the number of spools and groups that can be assigned.

Each node computes the candidate independently, and a different ownership map produces a different commitment. The group vote does not choose among candidates; it attests the assignment its signers recomputed. An honest owner signs only its own result, so a wrong root cannot gather more than f signatures, and a group approves the commitment once $f + 1$ of its owners sign it. The assignment is accepted only after every current group approves the same root. Target groups are then materialized individually from proofs against it. Materialization turns the global plan into the ownership records used in the next epoch. Solana verifies one bounded result at a time without trusting a scheduler to choose the complete assignment. Consensus remains global even though on-chain execution is incremental.

Several constraints govern the assignment:

1. No node may own more than one spool in the same group.
2. Nodes excluded from the next committee receive no assignments.
3. Spool counts are allocated in proportion to stake.
4. Existing owners keep their spools unless the preceding rules require reassignment.

A node may own multiple spools, but never more than one within the same group. Figure 5 shows one node's spools. A node that overstates its capacity may receive work it cannot serve. Repeated failures to answer storage challenges lead the committee to propose an eviction. The network is designed to recover from a node that leaves or is evicted.

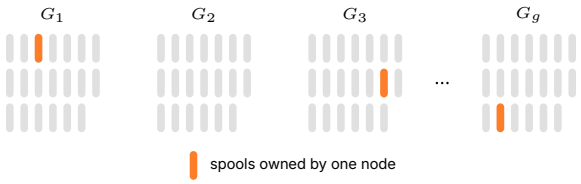


Figure 5: A node may own spools in several groups, but at most one in each.

6 Epoch Lifecycle

Moving a spool between nodes requires moving data, so assignments should not change every time a node joins or leaves. But leaving assignments fixed would prevent new nodes from participating. TAPEDRIVE updates assignments on an epoch cycle, keeping them stable between boundaries while providing a regular cadence for committee membership, voting, and rewards.

Each epoch moves through four phases: Sync, Snapshot, Active, and Closing. Reads and writes remain open throughout.

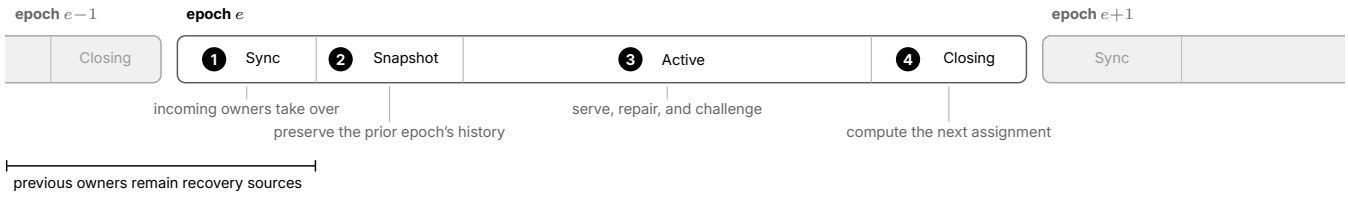


Figure 6: The four phases of an epoch. Previous owners remain recovery sources through Sync.

1. **Sync.** The new assignment becomes current when the epoch begins. New writes use the new owners immediately. Incoming owners copy the existing contents of their spools from previous owners and reconstruct anything that cannot be copied. Previous owners remain recovery sources for data written before the transition. A group completes Sync after $\max(k,\, f+1)$ of its current owners report that their assigned spools are ready. The phase ends after every live group completes.
2. **Snapshot.** The current committee derives the metadata snapshot for the preceding epoch from finalized Tape Replay state. Every live group approves the same snapshot commitment. The snapshot tape must also be materialized before the epoch enters Active.
3. **Active.** Current owners perform the network's ordinary storage work. They serve reads and writes, restore missing data, and respond to storage challenges. Registered nodes may volunteer for the next committee, and eviction votes determine which existing nodes remain eligible. Rewards for the preceding epoch are settled independently, so one stale node cannot prevent the network from advancing.
4. **Closing.** Membership in the next committee is frozen. Tape Spooler derives the next assignment from that committee and the finalized replay state. Every current group must approve the same assignment root, and every target group must be materialized before the epoch can advance. Once the assignment is complete, the next epoch begins in Sync.

Because the new assignment becomes current when Sync begins, new writes stop adding data to the previous owners. The set of older data that must be copied or reconstructed is fixed at the epoch boundary, while the overlap between previous and current owners keeps that data available during the handoff.

Previous owners have an economic reason to help the handoff finish. Rewards for an epoch can be claimed only after the next epoch reaches Snapshot, so a faster Sync lets them claim sooner. If a previous owner drops a spool, the incoming owner can recover or repair its slices from the rest of the group. An owner that leaves before claiming forfeits its reward once the network advances again, and the unpaid amount rolls into future reward pools.

Walrus places its boundary on the other side of synchronization, sending new writes to the incoming committee while continuing to direct reads to the outgoing committee until the handoff completes [4]. TAPEDRIVE makes the new assignment current first and retains previous owners as recovery sources. Both approaches prevent ongoing writes from extending the handoff indefinitely.

The lifecycle overlaps work across several epoch numbers. During epoch e, the network preserves the replay history and settles rewards for $e-1$, uses the committee and assignment for e, and prepares the committee and assignment for $e+1$. Changes that increase committee or group capacity are prepared one epoch farther ahead because the immediately following committee and assignment are already being assembled.

Track operations involve only one spool group. Epoch transitions involve every live group. Snapshot and assignment votes are therefore completed one group at a time, and target groups are materialized separately, keeping each Solana operation bounded. Every required group must still finish before the epoch advances, so the transition completes at the pace of the slowest group.

The synchronization step is specified in Algorithm 4 of Appendix A. Figure 6 summarizes the cycle.

7 Tape Slicer

Storing every large track in full with every owner in its group would make TAPEDRIVE too expensive. Erasure coding lowers that storage cost, but a straightforward code can require downloading the entire track to replace one missing slice. Since nodes fail and spool ownership changes over time, slice replacement is not an edge case.

Tape Slicer reduces both costs. It divides a track into n slices

such that any k can reconstruct the track, while providing a repair path that normally rebuilds one missing slice without downloading the complete track.

Strawman Design

The simplest design stores a complete copy of every track with every spool owner in its group. A reader can use any copy, and a replacement node can copy the track from any remaining owner. However, a group with n spools stores n copies of every track. That simplicity is useful for small objects, as the next section explains, but it is too expensive for large tracks.

Erasure coding reduces this storage cost. A conventional Reed-Solomon code produces n slices such that any k of them can reconstruct the track. Each slice is roughly $1/k$ the size of the original data, so the group stores roughly n/k times the track size rather than n times the track size [9]. The track remains recoverable while as many as $n - k$ slices are unavailable. Figure 7 shows this process.

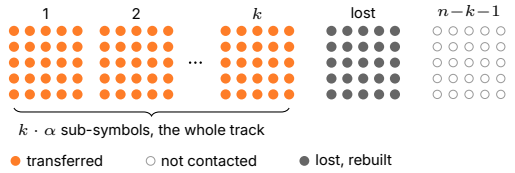


Figure 7: Recovery. Any k survivors send complete slices.

Recovering a track and replacing a slice are different operations. A reader needs the complete track, so downloading k complete slices makes sense. A replacement spool owner needs only the one slice assigned to its spool. With a straightforward Reed-Solomon layout, however, it may still have to download k complete slices, reconstruct the entire track, and encode the missing slice. Restoring a slice that is only a fraction of the track can therefore require transferring the equivalent of the whole track.

This becomes expensive when Tape Spooler moves a spool to a new owner. That spool contains slices from many tracks, and reconstructing an entire track for each missing slice can turn a routine handoff into a large transfer.

We call rebuilding the complete track from any k valid slices **recovery**. We call rebuilding one missing slice **repair**. TAPEDRIVE needs both: recovery for reads and as a reliable fallback, and repair that restores one slice with substantially less network traffic.

Slicer Design

Tape Slicer uses Clay codes for coded tracks. Like Reed-Solomon, Clay is a Maximum Distance Separable code: any k of the n slices can reconstruct the track. Clay also provides a repair

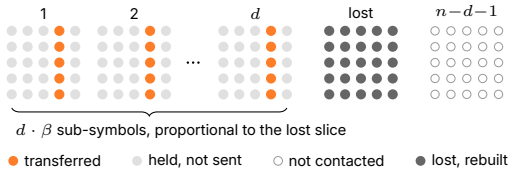


Figure 8: Clay repair. Each of d helpers sends β sub-symbols.

path in which d helpers each send only selected portions of their slices. It provides the lowest possible repair bandwidth for this amount of stored redundancy [12, 13]. Figure 8 shows the difference: recovery collects k complete slices, while repair collects smaller contributions from d helpers.

Tape Slicer processes a track as a sequence of stripes. A stripe is a portion of the track that is encoded independently. Each stripe produces n coded symbols, one for each spool in the group. The symbols assigned to the same spool across all stripes are joined together to form that spool's slice. Only the final stripe is padded when necessary.

The encoding description records the Clay parameters, stripe layout, original track length, and symbol placement. These values make the encoding deterministic: given the same track and encoding description, every participant produces the same n slices.

The client divides each completed slice into fixed-size sample leaves and builds a Merkle tree for that slice. The roots of the n slice trees become the leaves of a second Merkle tree, whose root commits to the complete ordered codeword.

This two-level structure serves two purposes. A spool owner can verify that its complete slice belongs at its assigned position, while a storage challenge can request one small sample leaf and its proof rather than the entire slice. Challenge proofs therefore remain bounded as tracks grow.

The client registers the track metadata and commitment, then sends each spool owner its slice and a proof connecting that slice to the commitment. The owner recomputes the slice root, verifies the proof, and accepts the slice only if it matches the registered commitment. The same check applies when a slice is copied or reconstructed later.

When a spool moves to a new owner, the new owner first tries to copy its existing slices from the previous owner. If a slice is unavailable, the new owner repairs it by asking d other owners for the portions required by the Clay repair equations. It combines those portions into the missing slice and checks the result against the track commitment before storing it. The helpers do not need to be trusted because incorrect contributions cannot produce the committed slice.

Repair uses more helpers but transfers substantially fewer bytes than reconstructing the complete track. If the node can-

not reach the required d helpers, or repair does not produce the committed slice, it falls back to recovery. It collects any k valid complete slices, reconstructs the track, and regenerates its assigned slice. The preferred path saves bandwidth; the fallback works with fewer reachable owners.

Objects larger than one coded track use the same mechanism repeatedly. The client writes the object as ordered segment tracks and then writes a manifest that records their order, offsets, and lengths. A range read reconstructs only the segments that overlap the requested range. Segmentation increases the maximum object size without introducing another encoding or verification model.

Section 9 follows these slices through registration, distribution, certification, and reads.

Symbol Placement

Clay produces source and recovery symbols for every stripe. The simplest placement keeps each symbol position on the same spool across every stripe. Some spool owners would therefore always store source symbols, while others would always store recovery symbols.

Direct reads can use source symbols without reconstructing the track. With a fixed placement, reads at particular byte offsets would repeatedly contact the same source positions. Those owners would carry more traffic for the same storage responsibility, while owners holding recovery symbols would receive less.

Tape Slicer instead rotates the symbol-to-spool mapping between stripes. The rotation visits every spool before repeating, giving each owner approximately the same mixture of source and recovery symbols. The encoding description records the rotation, so readers and repairers can locate every symbol. Rotation does not change the code's recovery guarantees; it prevents the different code roles from being permanently assigned to the same nodes.

8 Small Objects

Not every object is large. A storage network must work for objects of any size. On-chain activity produces accounts measured in bytes, and rent gives applications a standing reason to compress state and store it off chain.

Strawman Design

The simplest option is to write every track the same way. For a large payload, the storage saved by coding justifies every step of the coded write.

A small file pays the same fixed costs for far fewer useful bytes. The encoder may pad it to a minimum coding unit. Every slice still carries metadata. The client still contacts the same number of storage nodes, and certification still requires their signatures and another Solana transaction. None of these costs shrinks in proportion to the payload.

The stored-byte crossover can be stated directly. Let b be the payload size, a the minimum coding unit, h the per-slice metadata, and k the reconstruction threshold. Let $R(b)$ be the bytes stored as complete copies and $C(b)$ the minimum bytes stored by the coded representation. If every spool in a group of width n stores the complete payload, then

$$R(b) = nb.$$

For a nonempty payload no larger than one coding unit, the coded representation has the lower bound

$$C(b) = \frac{n}{k}a + nh.$$

Complete copies use fewer stored bytes whenever

$$b < \frac{a}{k} + h.$$

The group width n cancels because both paths use the same n spools. Each spool stores either the complete payload or one slice and its metadata. The exact boundary depends on the code and representation, but every construction with a minimum coding unit or fixed per-slice metadata has such a boundary. Encoding, slice distribution, proofs, signatures, and certificate publication add costs beyond the stored bytes counted here, further favoring complete copies for small files. Figure 9 shows the crossover.

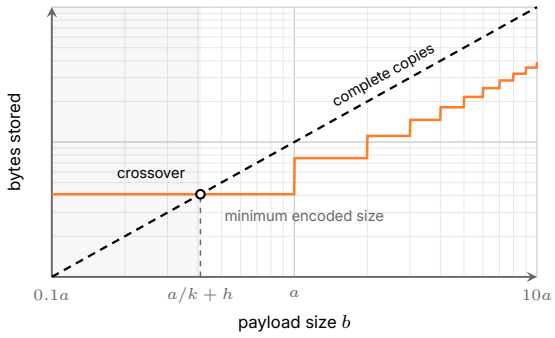


Figure 9: On logarithmic axes, complete copies scale with the payload, while padding and per-slice metadata give the encoded representation a minimum size that steps up once per coding unit.

Shelby shows the effect of a minimum coding unit. Its client pads input to 10 MiB, and its ten-of-sixteen Clay profile turns that unit into 16 MiB of coded data [5]. A payload much smaller than 10 MiB still pays for the full padded representation. This floor comes from Shelby's chosen unit, not from Clay codes themselves, which provide bandwidth-efficient repair.

A second option is to bundle small writes together. Walrus Quilt groups small files into a larger encoded unit and reports large reductions compared with storing them separately [10]. The saving is real, but it only arrives if the application collects its files into the unit the coding path expects.

Inline Track Design

TAPEDRIVE instead stores small tracks as complete copies. The client library selects the representation internally, so applications submit data the same way they always do. It uses complete copies only when it makes sense to do so and the write fits within Solana's transaction size budget.

The resulting **inline track** carries its payload inside the write instruction. One Solana transaction registers the track, records its payload hash, assigns it to a spool group, and marks it certified. Nothing else is required: no encoding, no slice distribution, no signature collection, no second transaction. A tiny write is certified the moment its transaction finalizes.

Once the block is final, Replay appends the track metadata to the replay log and hands the payload to storage. Every storage node hosting a spool in the assigned group keeps a complete copy, giving the group n copies on distinct nodes. The certification is the finalized transaction itself rather than q storage-node signatures, and nodes outside the group carry only the metadata.

Inline writes have a secondary benefit. Any Solana transaction or program can call the same write instruction, allowing on-chain activity to place small payloads directly into TAPEDRIVE. Compressed NFTs illustrate one possible use: their state data is recorded in transactions rather than account space and is usually indexed by centralized RPC providers [34]. Writing such data to a tape as part of the same flow reduces reliance on separate indexing infrastructure.

General Solana account compression likewise requires off-chain account bytes and an index for recovery. SIMD-0341 proposed this model but the underlying storage and indexing need remains [35]. Common accounts, including 165-byte token accounts, fit within TAPEDRIVE's small inline range. TAPEDRIVE could therefore provide a distributed index instead of leaving that role to centralized RPC providers. The design of these integrations is outside the scope of this paper.

9 Storage Correctness

Tape Replay records what exists, Tape Spooler decides which nodes are responsible for it, and Tape Slicer decides which bytes each of those nodes keeps. A write uses all three at once, and the sections that introduced them separately have not yet shown the complete path. Walrus names the target such a path must meet: **Complete Data Storage**, defined by Write Completeness, Read Consistency, and Validity [4]. TAPEDRIVE adopts the same target, scoped to one track inside its assigned spool group. This section follows the write and read paths end to end, then shows they satisfy the target.

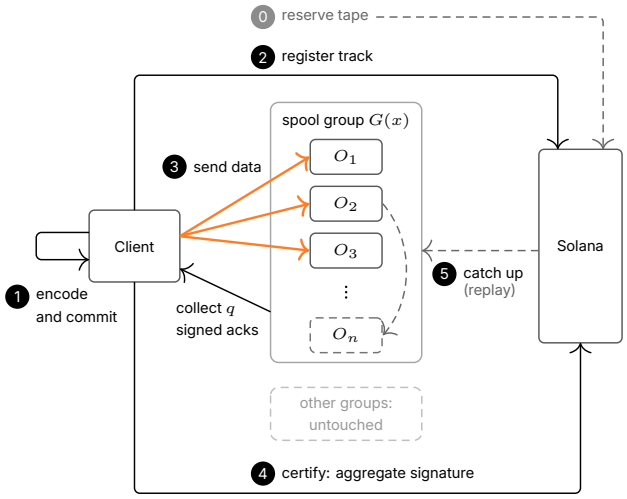


Figure 10: Writing a coded track.

Writing a Coded Track

The client library routes each payload by size. Everything beyond the inline range follows the coded path, which proceeds in five steps. Figure 10 traces them. Every write begins from a live tape whose reservation was paid once (0); no step below buys space.

1. **Encode and commit.** The client cuts the payload B into stripes and applies the Clay code, producing the n slices $s_1, \ldots, s_n$. It builds a Merkle root over each slice and a second tree over the n roots, whose root C_x commits to the complete ordered codeword. It also computes the value hash $h_x = H(B)$ and fixes the encoding description θ: code profile, stripe layout, original length, and symbol-to-position rotation. Any party holding θ can reproduce this encoding exactly.
2. **Register.** The client submits one Solana transaction carrying h_x, θ, C_x, and any object metadata. Execution verifies the writer's authority over a live tape and

assigns the track to one live spool group. Once the containing block is finalized, replay places the track, its commitments, and its group assignment into every participant's state. The registration names the data; no payload bytes have moved yet.

3. **Distribute.** The finalized assignment identifies the group's current owners. The client sends each owner its assigned slice together with the Merkle proof connecting that slice to its position under C_x. The owner recomputes the slice root, verifies the proof against the registered commitment, stores the slice, and only then signs an acknowledgement over the track. A signature vouches for bytes the signer has verified and stored, not for bytes it expects to receive.
4. **Certify.** The client collects acknowledgements from q distinct spool owners, aggregates them into one compact certificate, and submits it in a second Solana transaction. When that transaction finalizes, the track is certified. Certification is TAPEDRIVE's proof of availability. From this point the group, not the writer, is responsible for the payload: the client may delete its local copy and go offline, and any later reader can hold the network to the registered commitment.
5. **Catch up.** An owner that was slow, unreachable, or assigned after distribution learns about the certified track from replay rather than from the client. It repairs its slice from d helpers, or falls back to recovery, collecting k verified slices, reconstructing the payload, and re-encoding it to regenerate its own position. Either way it keeps only a slice that verifies against C_x. Storage that begins with q spool owners therefore spreads to every honest owner without further work from the writer.

Writing a Stream

An object larger than one coded track becomes a stream. The client cuts it into segments, each at most one coded track, and writes every segment through the five steps above. Segments are independent tracks with their own group assignments, so their write paths can run concurrently against different groups. When every segment is certified, the client writes a manifest track recording their order, offsets, and lengths. The manifest travels the same write path and comes last, so by the time it certifies, every track it names is already certified. The manifest's certification completes the object: from that point the stream, like any single track, is the network's responsibility.

Writing an Inline Track

A payload inside the inline range compresses the five steps into registration. Encoding reduces to the value hash, and the write instruction carries the payload itself; execution records h_x and the group assignment, and the track is certified when that single transaction finalizes. Distribution ❸ and certification ❹ vanish: there is no slice to send and no certificate to submit, because the finalized event carries the payload to every owner in the assigned group, and each owner stores a complete copy as it processes the event. Catch up ❺ remains. An owner that missed the event fetches a copy from the group, accepting it only when its hash equals h_x. The payload itself travels through the transaction.

Reading a Track

A reader begins from replayed state, which supplies the track's value hash, its commitment, its spool group, and the group's current owners. Every honest reader replays the same finalized events, so two readers asking for the same track check their answers against the same commitment.

We write $\perp$ when a reader rejects an invalid committed representation.

For an inline track, the reader asks owners in the group for the payload and accepts the first response whose hash equals h_x. One honest owner suffices.

For a coded track, the reader requests slices from the group's owners in parallel and discards any response whose Merkle proof does not match its committed position. From the first k valid slices it decodes a candidate payload. It then re-encodes the candidate under θ and accepts only if the re-encoded codeword reproduces C_x and the candidate hashes to h_x. If both match, the reader returns the payload. If no payload can match, because the writer committed to an invalid encoding, every reader that terminates returns $\perp$, whichever k slices it happened to draw.

A read submits no transaction and changes no TAPEDRIVE state. The asynchronous model shapes what a slow read means: a read that has not yet gathered enough valid responses is pending, not failed, and a timeout is not evidence against the data.

Reading a Stream

A stream read starts with its manifest, an ordinary track read. The manifest names the segment tracks and their byte ranges, so a range request resolves to the segments that overlap the requested interval and leaves the rest untouched. Each needed segment is then read and verified as an ordinary coded track. The object's bytes are exactly the verified bytes of its tracks; assembling them adds no new trust.

Correctness Properties

The paths above are evaluated as a storage protocol for one live track x with value hash h_x, assigned to spool group $G(x)$.

Writers and readers may be Byzantine. The properties below distinguish the guarantees of an honest write from the consistency required even when the writer is Byzantine.

For the properties below, a write is successful once its track is certified in finalized state.

Write Completeness. If an honest writer successfully writes payload B as track x, every honest spool owner in $G(x)$ eventually stores the representation assigned to its position, the complete payload for an inline track or the committed slice for a coded track.

Read Consistency. Two honest readers of the same finalized state return the same payload for x or both return $\perp$, even when the writer is Byzantine.

Validity. If an honest writer successfully writes B as x, then while x remains live, every honest reader eventually returns B.

The properties rest on two threshold inequalities. Certification must remain reachable when every Byzantine spool owner stays silent, and a certificate must pin down enough honest data to rebuild the track:

$$q \le n - f \qquad \text{and} \qquad k \le q - f.$$

Eviction and invalidation additionally require intersecting quorums,

$$2q > n + f,$$

so that any two certified results share an honest signer and cannot conflict. Assignment and snapshot approval do not need intersection. Every honest owner recomputes the same value from the same finalized state and signs only its own result, so a wrong value gathers at most f signatures and $f + 1$ signatures prove the common one. The storage arguments below use only the first two inequalities.

Everyone reasons from the same state. Honest participants replay the same finalized events in the same order and apply the same deterministic changes, so they hold the same TAPEDRIVE state. The writer, the owners, and every reader therefore agree on the live track, its h_x, C_x, and θ, its spool group, and the group's current owners. A participant that has fallen behind sees a shorter history, not a different one.

A certificate pins k honest slices. Of the q signers in step ❹ of the coded write, at most f are Byzantine, so at least $q - f \ge k$ are honest. Step ❸ made each honest signature stand for a verified, stored slice. A certificate therefore guarantees, at the moment it forms, at least k honest spool owners whose slices match C_x, exactly enough to reconstruct the payload.

Write Completeness. For an inline track, the finalized registration carries the payload, so every honest owner stores its copy while processing the event, and a laggard accepts a fetched copy only when its hash equals h_x. For a coded track, step ❸ gives owners their verified slices directly, and the certificate margin keeps step ❺ supplied: a missing owner can always find k honest verified slices, rebuild the payload, and regenerate exactly its assigned slice, checking it against C_x. Eventual delivery does the rest.

Read Consistency. An inline track has one canonical payload in its finalized registration, and a reader accepts a response only when its hash equals h_x, so two accepted inline reads cannot differ. For a coded track, the read path discards unverifiable slices, decodes, and re-encodes. Deterministic encoding and a binding commitment leave no room between two accepted answers: if two readers accepted different payloads, both payloads would re-encode to codewords bound by the same C_x, which commitment binding excludes except with negligible probability. If the writer committed to an encoding that no payload produces, no candidate can reproduce C_x, so every terminating reader returns $\perp$ regardless of which slices it drew.

Validity. An honest writer commits a valid encoding of B, so the certificate margin leaves at least k honest owners holding slices of the true payload. An honest reader eventually collects their responses, all of which verify, decodes B, and passes the re-encoding check by construction. The reader returns B.

A stream adds no separate property. Its manifest and segments are ordinary tracks, and a stream read returns the same bytes for two readers exactly when its track reads do.

All three properties hold under the asynchronous network model, so the write and read paths implement Asynchronous Complete Data Storage in Walrus's vocabulary, scoped to one track and its spool group. The guarantees take one active assignment of spools to owners as a fixed input. Maintaining them across a change in physical ownership additionally requires the handoff conditions established by the epoch lifecycle. Algorithms 1–3 in Appendix A specify the write, read, and recovery procedures, including every verification condition used by these arguments.

10 Storage Challenges

A track is certified when enough spool owners sign that they received its data. Those signatures prove receipt at one moment, not storage for the rest of the tape's lifetime. A storage node can sign truthfully, remain eligible for future work, and later quietly discard bytes it expects nobody to request. TAPEDRIVE uses storage challenges to catch silent loss before it threatens recovery.

A challenge asks a spool owner to return a **proof of access** on short notice. This is either a Merkle proof for a coded slice or the complete replicated data from an inline write. The request must be unpredictable, every spool owner must be able to reproduce it, and recording the result must not put every check on Solana.

Strawman I: Wait for a Read

The simplest design waits for a read to fail. It adds no traffic after the write, but a rarely read track could fall below its recovery threshold before the loss is detected. Certification would become the last evidence that the data remained available.

Strawman II: Put Every Proof on Solana

Putting every proof of access on Solana would reveal loss before an application read and give every observer the same public record. It would also make the common case scale with stored data and challenge frequency, even while every spool owner behaved correctly. TAPEDRIVE instead keeps routine checks off chain and uses Solana only for finalized control state and fresh public values.

Strawman III: Predictable Off-Chain Sampling

Moving checks off chain removes that routine chain work, and deriving the schedule from the epoch nonce lets every spool owner reproduce it. But the nonce is known when the epoch begins. If it also selected the samples, a dishonest owner could retain only those bytes or recover them shortly before their turn. The schedule may be known in advance; the samples cannot be.

Whirlwind Design

TAPEDRIVE's leaderless storage challenge system uses future Solana blocks to select unpredictable requests off chain, then combines matching signatures on timely proofs of access into independent success certificates for each spool.

The design relies on a simple asymmetry: reading retained data is fast, while rebuilding discarded data requires network transfer and may require decoding. The response deadline and challenge cadence are chosen to make repeated recovery a poor substitute for storage.

Whirlwind runs repeated rounds throughout each active epoch. Every round tests each active spool independently and may produce one success certificate per spool:

1. **Schedule.** Round times are derived deterministically from finalized epoch state. Each round identifies a short window of future Solana slots. The first block in that window that later appears in finalized history becomes the round's **entropy block**. If no block from the window finalizes, the round is void. Figure 11 shows this schedule.
2. **Select.** The entropy block's hash is combined with the epoch, group, round, and spool to select one entry from that spool's sample set. The set is fixed by TAPEDRIVE state on the block's ancestry. Each coded-slice leaf is one entry, and each inline track is one bounded entry.
3. **Respond.** Participants act on produced blocks before finality. The challenged owner derives the request on the branch it observes, reads the selected data, and broadcasts a signed proof of access before a deadline measured from the candidate block's slot. A response tied to a block that does not survive finality is discarded.
4. **Certify.** Once the candidate block is confirmed, the other spool owners verify the proof of access, the challenged owner's signature, and the recorded arrival time against that block's state. An owner that accepts the response relays it and signs the epoch, group, round, challenged spool, and candidate block. At q matching signatures, any owner may aggregate them into a success certificate. The certificate becomes standing evidence only if its entropy block finalizes.
5. **Record.** Each owner keeps its own history of success certificates and local misses. Certificates continue circulating after their round, so a late certificate may replace a recorded miss before anyone acts on it. Figure 12 shows one group's record across an epoch.

Each certificate names one challenged spool. The challenged

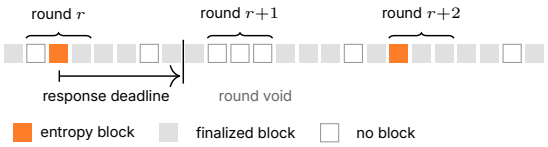


Figure 11: Whirlwind challenge rounds scheduled by Solana slot height.

owner may contribute one signature but cannot certify itself. Since at most f signers are Byzantine, a certificate contains at least $q - f$ honest signatures. Signatures made against different block candidates cannot aggregate.

Whirlwind produces positive evidence of timely access, not proof of failure. The absence of a certificate may mean that the owner withheld its response, but an honest response or certificate may also have been delayed. A missing result for one spool therefore counts only as a local miss and does not delay other spools or later rounds.

Whirlwind uses the block hash for unpredictability, not unbiased randomness. The canonical-block rule prevents challenge participants from choosing whichever observed block gives a favorable sample. Any remaining block-producer influence is part of the challenge calibration rather than a new randomness assumption.

Suppose an owner has discarded a fraction p of its sampling set. If each round selects independently and selecting missing data prevents a certificate, the probability that ℓ rounds all avoid the missing data is

$$(1 - p)^{\ell}.$$

More frequent rounds reduce this probability. The expression does not account for every network condition or reconstruction strategy: delayed traffic can hide an honest response, and a dishonest owner may sometimes reconstruct missing data before the deadline. A challenge tests access at one sampled moment; it does not prove continuous storage between rounds.

Eviction Design

Whirlwind does not evict storage nodes directly. Repeated local misses may instead lead spool owners to propose an **eviction**. After q owners in the affected group approve the proposal, more than two thirds of the live groups must approve the eviction.

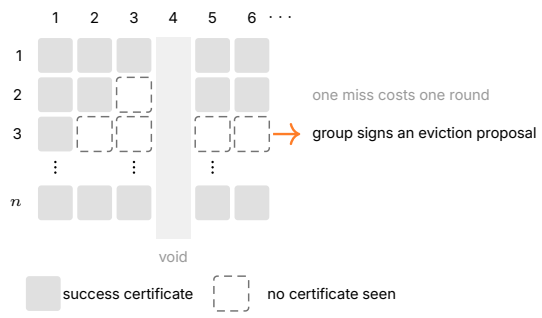


Figure 12: One spool group's Whirlwind challenge record across an epoch; rounds are columns, spools are rows.

Solana verifies the votes, not the underlying challenge histories. One missing certificate therefore cannot remove a node, and a minority cannot impose its local observations on the network.

An approved eviction removes the storage node from the next committee and its assignments. Tape Spooler assigns its spools to new owners, and Tape Slicer restores any missing data during the following handoff. Reward slashing remains outside the scope of this paper.

11 Continuity and Durability

Long-term storage depends on two related properties. Continuity keeps a track readable across epoch transitions. Durability restores lost data before the remaining slices fall below the reconstruction threshold.

Redundancy and Recovery

A track remains assigned to the same spool group, and each slice remains mapped to the same spool. At an epoch boundary, Tape Spooler may assign those spools to new physical nodes.

The transfer happens during Sync. An incoming spool owner first copies the spool's data from its previous owner. If a coded slice is missing or invalid, Tape Slicer reconstructs it from the other spools in the group. An inline track can be copied from any owner with a valid complete copy. In either case, the incoming owner verifies the recovered data against the track's original commitment.

New writes use the new owners as soon as the epoch begins, so the set of older data being transferred does not continue to grow. Previous owners remain recovery sources for that older data until Sync completes. Reads remain available throughout the handoff: a coded track can be read from any k valid slices, while an inline track needs one valid copy.

An owner reports readiness only after its assigned spools are active and no recovery remains pending. A spool group completes Sync after $\max(k,\ f+1)$ of its current owners report readiness. The gate is deliberately low. Owners that left the network can never report, and the reassignment that would replace them waits on the far side of the transition, so a higher gate lets missing owners block the very step that repairs the group.

At least one of the reporting owners is honest, so an inline track keeps a complete verifiable copy across every handoff. When the missing owners have crashed or left rather than lied, every reporter is honest, and the k bound in the gate means the ready owners alone can reconstruct every coded track. Byzantine owners can report readiness falsely, and in the worst case as few as $\max(k,\ f+1) - f$ honest owners are ready when the phase ends.

The track commitment remains the same across reassignment, so readers continue to verify data against the same value. When the reporters include k honest owners, the group is self-sufficient from the boundary onward: owners that were not ready obtain or reconstruct their assigned data from the ready owners, and under eventual communication every honest owner acquires the data assigned to its spools. Fewer honest reporters narrow that margin without by themselves losing data. Previous owners that remain reachable and current owners still recovering continue to hold slices, and storage challenges during Active expose whichever spools stay empty so that repair and eviction can restore the margin. The write and read guarantees carry across the epoch boundary with a recovery margin that is narrowest at the boundary and widens as the epoch proceeds.

Restoring Lost Data

For a coded track, certification establishes the initial recovery set: at least q owners acknowledged valid slices when the write completed. Durability depends on maintaining that recovery margin as nodes lose data or leave the network.

Storage challenges provide continuing evidence that owners can access their assigned data. An owner that discovers a missing slice can reconstruct it from the other spools in its group. If an owner repeatedly fails to answer challenges, the group may propose an eviction. An approved eviction excludes the node from the next committee, and Tape Spooler assigns its spools to another node.

During the following Sync, the replacement requests each live slice from the previous owner. If the previous copy is unavailable, Tape Slicer reconstructs the slice from the other spools in the group and verifies it against the original commitment. The replacement reports readiness only after that work is complete.

Durability comes from the interaction of several mechanisms:

1. Certification establishes that enough valid data exists after a write.
2. Storage challenges make persistent unavailability increasingly visible.
3. Eviction removes a node that the network no longer wants to rely on.
4. Tape Spooler assigns its spools to a replacement.
5. Sync restores the data and re-establishes the group's recovery margin.

Inline tracks follow the same ownership loop without erasure-code reconstruction. Each spool owner stores a complete copy, so a replacement can fetch any surviving copy and verify its value hash. The finalized Solana transaction may provide another source while its history remains available. TAPEDRIVE does not depend on Solana retaining that history permanently.

A stream adds no separate durability mechanism. Its manifest and segments are ordinary tracks, and each must remain recoverable for the complete object to remain readable.

How Much Loss a Group Can Tolerate

A coded track is certified after q spool owners acknowledge valid slices. At most f of those owners can be Byzantine. Because $q - f \geq k$, certification leaves at least k honest owners with valid slices. The track remains reconstructable even if every Byzantine signer immediately discards its data.

This guarantee applies separately to each spool group. The estimate below follows Shelby's independent-failure model, whose analysis shows that erasure coding combined with timely detection and restoration can provide strong durability [5]. Registered identities do not prove independent operators or infrastructure, so correlated failures can exceed the model's estimate. TAPENET spreads a group's owners across operators; on one machine or at one site they share one failure domain.

Loss During the Restoration Window

Independent failures become dangerous when they accumulate faster than the network can detect and restore them. Let MTTD be the mean time to detect a loss and $\text{MTTR}_{\text{rebuild}}$ the mean time required to restore it. Their sum defines the critical period.

$$T_{\text{critical}} = \text{MTTD} + \text{MTTR}_{\text{rebuild}}$$

Shelby models the probability that an initial loss is followed by the additional $n-k$ losses needed to make a track unrecoverable [5]. For an annual position-loss rate λ and T_{critical} measured in hours, the model gives the following annual loss probability.

$$P_{\text{loss}} \approx (n\lambda)\binom{n-1}{n-k}\left(\frac{\lambda T_{\text{critical}}}{8760}\right)^{n-k}$$

The paper's 10-of-16 example assumes a 50% annual position-loss rate, 24 hours to detect a loss, and 12 hours to rebuild it. This gives an annual loss probability of 3.01×10^{-12} per stored set.

A conservative profile puts TAPEDRIVE at thirteen tolerated unavailable positions, more than doubling the six-position failure margin in that example.[2] Under the independent-failure model, this wider margin sharply reduces the calculated loss risk.

[2] The conservative analysis uses $k = 7$ of $n = 20$ positions and $d = 16$ repair helpers, giving a coded storage multiplier of $20/7 \approx 2.86$ before metadata and padding. Production coding parameters have not yet been fixed.

Durability as the Network Grows

Each track remains within its original spool group as the network grows. Adding groups creates capacity for new tracks without widening the storage path of existing tracks. An existing track still has n slices, a writer still waits for q acknowledgements, and a reader still needs k valid slices.

This keeps the fault boundary and ordinary object path limited to one spool group. Additional groups do not increase the redundancy of existing tracks.

Some coordination remains network-wide. Snapshot and assignment votes require every live group to finish, so a slow group can delay an epoch transition even while its tracks remain readable. That rule is deliberate. A group that cannot produce even $f+1$ signatures is not slow but dark, and advancing past it would move the network forward while some of its data is unaccounted for. Adding groups also increases the chance that at least one group is slow or has correlated operators.

Horizontal growth therefore adds capacity while preserving a fixed path for each track. The durability loop remains the same at every scale: maintain enough valid slices, replace owners the network no longer relies on, and restore missing data before further losses make a track unrecoverable.

12 Access and Delivery

TAPEDRIVE stores a coded track as slices spread across several storage nodes. Browsers and ordinary object-store clients do not speak the protocol needed to find those nodes, collect enough valid slices, and reconstruct the track. A **gateway** performs that work and exposes the result through familiar HTTP and S3 interfaces.

A conventional content delivery network can cache gateway responses near users. Storage nodes remain focused on storing and restoring data, gateways provide application access, and CDNs handle repeated public traffic.

Direct Access

Protocol-capable clients can access the storage network directly. An inline write travels through Solana, while a coded write sends slices to the track's assigned spool owners. A verification-capable client can discover those owners, collect valid slices, and reconstruct a track.

Storage nodes may require direct readers to use a registered network identity with a minimum amount of stake. Each node chooses its own threshold, and a reader needs access to enough owners in the group to recover the track. Anyone may register and stake an identity; committee membership is not required.

This gives storage nodes a way to reserve bandwidth for protocol traffic. A coded read contacts several owners, and popular public objects can produce much more traffic than writes. The stake threshold makes sustained demand attributable while leaving direct access permissionless.

Gateway Service

Anyone may operate a gateway by registering an identity, carrying enough stake to reach storage nodes, and following Tape Replay. A gateway never joins the storage committee merely because it serves traffic. It holds no spools, casts no storage votes, and receives no storage reward. Its revenue comes from the service it provides around the network.

The result is the same broad division seen around Solana RPC. A public endpoint makes the system immediately useful, while applications with sustained traffic pay an infrastructure provider or operate their own service. A public gateway can serve browsing, development, and light use under conservative request and byte limits. Independent gateways may sell greater volume, lower latency, geographic coverage, S3 access, or managed writes. Applications may instead run gateways themselves and control their own stake, caches, limits, and deployment.

How Gateways Serve Data

A gateway is a lightweight TAPEDRIVE node optimized for serving reads. Figure 13 traces the read path. The gateway uses the same node machinery, including Tape Replay 0, to follow finalized events and maintain the tape, track, object, and assignment indexes needed to answer requests. A request by object name 1 resolves to the latest valid track with that name 2. A

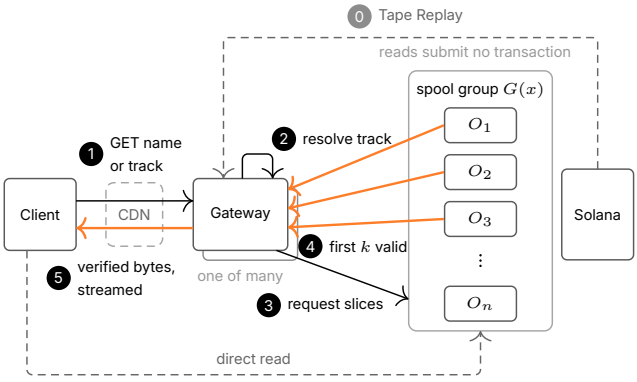


Figure 13: Reading a coded track through a gateway. A CDN can cache decoded responses near users, and a staked identity can read from the group directly.

request by track address selects the track directly. For a stream, the manifest identifies the segments and byte ranges needed for the response.

For an inline track, the gateway fetches a complete copy and verifies its value hash. For a coded track, it requests slices from the group's owners in parallel ❸, rejects slices whose proofs do not match the commitment, and reconstructs the track from the first k valid responses ❹. It verifies the reconstructed value before serving it ❺.

Requesting slices from more than k owners keeps slow or unavailable owners off the critical path. Reconstruction proceeds stripe by stripe, allowing the gateway to begin streaming before it has reconstructed the complete object. A range request reconstructs only the stream segments that overlap the requested interval.

The storage protocol requires no transaction for a read and makes no change to Solana or TAPEDRIVE state. A gateway may add a Solana transaction to charge for its service under its own economic model. Data throughput depends on storage-node and gateway capacity rather than control-chain execution. Gateways may cache verified slices and reuse them across reads and range requests.

S3 and the Last Mile

An object store is not itself a content delivery network. Production S3 deployments commonly place a CDN in front of the storage origin for high-volume public traffic. TAPEDRIVE follows the same division. At public scale, a gateway acts as the pull-through origin, while the CDN absorbs repeated last-mile requests and caches decoded HTTP responses at the edge. A cache hit changes where the response comes from, not which bytes match the track commitment; applications can still verify the returned bytes, or skip the caches and read directly.

Existing S3 tooling can use the network without learning the slice, certification, or assignment mechanisms beneath it. The S3 interface maps familiar concepts onto TAPEDRIVE's object model:

S3 concept	TAPEDRIVE concept
Bucket	Tape
Object key	Track name
Object	Latest valid track under that name
ETag	Track commitment
PUT or multipart completion	Delegated TAPEDRIVE write

The interface supports common list, head, get, range, put, delete, and multipart operations using SigV4-signed requests. Tape expiry and track replacement continue to follow TAPEDRIVE's lifecycle rules.

For writes, a tape owner may appoint a gateway as a delegate. The gateway can then register, certify, and delete tracks for that tape, and the owner may revoke the delegation. Delegated writes follow the same registration, distribution, and certification path as direct writes.

The same HTTP surface can serve a tape's named objects as a static website. Stable track addresses support long-lived assets, while name-addressed paths can resolve to newly written tracks. A CDN can cache the resulting site at the edge.

Public and Private Storage

TAPEDRIVE is designed to make public data easy to publish and retrieve. Many of its natural workloads are intended for broad access: websites, public datasets, software artifacts, media, and files shared through a stable link. A public track can be read directly by a protocol-capable client or served through a gateway using familiar HTTP and S3 interfaces. Gateways can place conventional content delivery networks in front of frequently requested objects, allowing public data to be cached and delivered close to readers.

Public access does not mean that every object stored on TAPEDRIVE must be public. Applications also produce data intended for a smaller audience. A repository may contain unfinished work. An agent may accumulate memory, intermediate results, or private artifacts. A backup may need to remain available for years without being readable by anyone who discovers its address. Supporting these workloads requires a way to control who may retrieve an object without making access depend on an account held by one gateway or service provider.

TAPEDRIVE's ownership model provides a natural foundation for private storage. A track can be marked as public or private when it is registered. A public track follows the ordinary read path. For a private track, a storage node returns its assigned slice only when the request includes a valid authorization signed by the tape owner or by a delegate the owner has approved. An authorized client or gateway presents that authorization to the track's storage nodes, collects enough slices, and reconstructs and verifies the object.

This makes access follow control of the tape rather than membership in a particular service. An owner can authorize another person, an application, an agent, or a gateway to read a private track. That authorization can be limited in duration or scope, allowing access to be shared without transferring ownership of the underlying data. A service can therefore work with private repositories, application state, or agent artifacts while the owner remains free to use another authorized service later.

Private access and encryption provide different protections. Requiring signed authorization prevents honest storage nodes

from serving private tracks to unauthorized readers. It does not hide the stored representation from the nodes responsible for it. Storage nodes retain their assigned slices, nodes involved in recovery may obtain enough data to reconstruct a track, and an authorized gateway sees the object it reconstructs. Users who need confidentiality from storage nodes and gateways can encrypt their data before writing it. Private access then limits who can retrieve the ciphertext, while encryption limits who can understand it.

Inline tracks require special treatment because their payloads are carried by the Solana transaction that registers them. Restricting later reads cannot make those transaction bytes private. Private tracks must therefore use the coded storage path or contain data encrypted before registration.

The complete design and implementation of private tracks, including authorization formats, delegation, revocation, and gateway integration, are left to a future paper.

13 Operator Controls

Storage operators decide how TAPEDRIVE grows and set its storage economics. Each active node publishes its preferred committee capacity, spool-group count, and storage price. At epoch close, the committee resolves those preferences through a **quorum-weighted vote**.

Growing the Network

Committee capacity sets how many storage nodes a future committee can admit. Nodes join through the ordinary stake and eligibility process. Once admitted, their stake and declared capacity determine how Tape Spooler distributes spools. Raising committee capacity creates room for more operators to participate.

Spool-group count determines how many groups divide storage responsibility. Each new group adds capacity for additional tracks while existing tracks remain in their original groups. Before a new group becomes active, it is created and its spools are assigned to committee nodes.

Storage price determines what users pay when they buy storage capacity. Changes apply to future purchases, while storage already paid for keeps its original price.

Voting Preferences

A node gets one unit of voting weight for each spool it owns. Nodes carrying more storage responsibility therefore have more influence over network settings.

Each node publishes a preferred value for every setting. At epoch close, a fixed quorum rule combines those preferences. Every participant begins with the same committee, spool assignments, and published values, so every participant derives the same result.

The resolved settings become active at epoch boundaries. Changes to committee capacity and spool-group count take effect after the required committees, groups, and spool assignments have been prepared.

Beyond the voting preferences, each storage node controls how its own machine participates. It publishes the minimum stake a direct reader must present before consuming its bandwidth.

Evicting a Node

The network may evict a node whose performance or behavior it will no longer accept. Reasons may include slow reads, failure to serve assigned data, or malicious behavior. The challenge system provides evidence of slow or missing responses, but the network does not rely on that evidence alone. A node may be evicted even if it has passed every challenge.

An approved eviction vote that reaches quorum majority removes the node from the next committee. Tape Spooler assigns its spools to replacements, and the ordinary handoff process restores any missing data.

Evicting a Track

If the network decides that a track should be removed, TAPEDRIVE provides a few mechanisms for this. A track can leave the network's valid set after its spool group approves an **invalidation** vote. At least q current spool owners sign an aggregate vote naming the track. The Solana program verifies the group, signatures, and threshold.

A node may also publish a **refusal** for a track or an entire tape. An operator might refuse data for legal, geopolitical, or business reasons. Tape Replay makes the refusal visible to every participant. The node stops storing and serving the affected data, and those bytes are removed from its reward weight. The decision affects only that node. Other spool owners continue carrying the track, so it remains available.

Beyond this, each gateway chooses what its application-facing service returns. A gateway may decline an object for legal, geographic, or business reasons, remove its cached data, and return a removal response. The track remains available through other gateways and direct reads while it remains valid on the storage network.

14 Economics

TAPEDRIVE is designed to make decentralized storage economically viable, not merely technically possible.

On TAPENET, storage revenue must cover operator costs at a price users are willing to pay. Over the long run, that revenue must come from users, not token emissions. The comparison that matters is with the cloud, where most of this data lives today, and its list prices sit well above the dedicated servers underneath them.

Amazon S3 Standard lists storage at roughly $23 per TiB-month, Google Cloud Storage at roughly $20, and Azure Blob Storage at roughly $18 [15, 38, 39], while dedicated servers suitable for storage place raw capacity between approximately $1.61 and $4.77 per TiB-month [16, 17, 18, 19]. These are rented machines, with the provider's margin, power, and network included; an operator running its own hardware sits below this range, whether it runs a private TAPEDRIVE on that hardware or joins TAPENET and earns from capacity it already runs. Figure 14 compares the monthly storage prices.

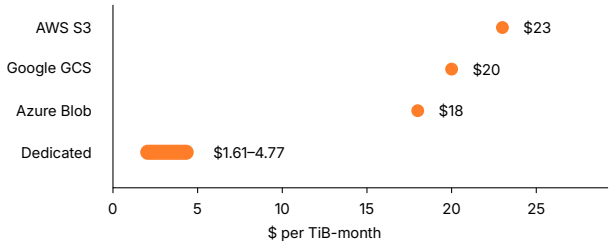


Figure 14: Monthly storage prices. Cloud list prices against raw dedicated-server capacity.

The gap widens for delivery. S3 egress fees add another $0.09 per GB, or roughly $90 for each TiB delivered; Google and Microsoft meter delivery at $0.12 and $0.087 per GB [15, 38, 39]. Dedicated servers commonly include public traffic or charge about $1 per additional TB [16, 17, 19]. Figure 15 compares the delivery prices.

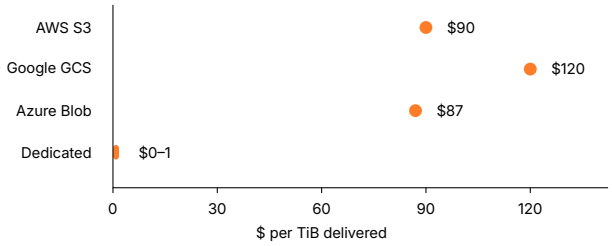


Figure 15: Delivery prices. Dedicated servers include public traffic or meter it at about a dollar per TB.

For a very conservative erasure-coded profile, the corresponding physical storage cost is roughly $3.22 to $14.31 per logical TiB-month. The range excludes metadata, repair traffic, and operational headroom. Even at its upper end, it sits below the cloud list prices quoted above, and the spread between them is where storage node operators earn their revenue.

The revenue picture is better than the range above suggests. A particular user's storage reservation is not equal to network utilization. Users reserve tape capacity ahead of use, and the storage nodes need to account for only the utilization side. The physical cost behind a reserved TiB is therefore lower than the cost behind a stored one. With that gap included, operators have room to price storage well below S3's list prices.

Reserving Storage

Cloud services such as Google Drive, Google Photos, and Dropbox sell fixed storage tiers by month or year [36, 37]. Users choose a tier based on how much capacity they expect to need and pay for the complete tier whether they fill it or not. The provider receives the plan price while keeping the purchased capacity available, whether or not the user fills it.

TAPEDRIVE uses the same capacity-plan model behind its S3-compatible interface. The user chooses a capacity R and a lifetime of L epochs. If p_{store} is the price of one capacity unit for one epoch, the tape cost is

$$C_{\text{tape}} = p_{\text{store}} RL$$

The user pays once in **TAPE**, the token used by the network for storage payments, staking, and rewards. The network creates the tape storage resource only if enough capacity remains available in every epoch of its requested lifetime. This makes future storage predictable and avoids the need for a separate payment for each write.

Prepayment simplifies later writes. Once a tape has been purchased, the owner can write, replace, and delete tracks without another TAPE payment. Each operation carries only its ordinary Solana transaction fee in SOL. The write path does not need to include the TAPE token accounts or the TAPE mint, and the same simplification carries into TAPEDRIVE's SDKs and tools. This is especially important for existing Solana applications that wish to use TAPEDRIVE's inline write feature where transaction byte space is limited.

Deleting a track frees that space for another write. Buying more capacity or time uses the storage price active at that time.

Storage Utilization

A tape's purchased capacity and the bytes currently stored inside it are different quantities. Suppose a tape has capacity R and its live tracks use uR, where $0 < u \leq 1$. The effective price for each occupied byte is

$$p_{\text{occupied}} = \frac{p_{\text{store}}}{u}$$

A half-full tape costs the same as a full tape with the same capacity and lifetime. Purchased capacity determines the user's payment and whether the requested storage allocation fits. Storage rewards are divided according to the live bytes assigned to each node.

Setting the Storage Price

Storage and operating costs change over time. TAPEDRIVE therefore lets storage nodes set the storage price through their quorum-weighted vote. The price determines how much TAPE a user pays for capacity. TAPEDRIVE defines how nodes agree on the value and leaves the value itself to the operators.

When a user buys storage, the payment is divided across the tape's lifetime so storage nodes are paid during the epochs in which they provide the service.

Storage Incentives

A new storage network needs enough operators and capacity to attract users before customer payments are large enough to support those operators. A permissionless subsidy vault exists to fill that initial gap.

Let S_e be tape payments scheduled for epoch e, V_e the amount released from the subsidy vault, and U_e unpaid rewards carried forward from earlier epochs. Together they form the epoch's storage reward pool.

$$F_e = S_e + V_e + U_e$$

Each epoch releases a fraction of the subsidy vault's remaining balance, so its contribution declines over time. As more tapes are purchased, scheduled customer payments become the primary source of storage rewards.

Storage rewards follow the bytes assigned to each node. Let A_i be the bytes stored in node i's assigned spools during the prior epoch, and let D_i be the bytes covered by that node's declared refusals. Its payable assigned bytes are

$$A_i^{\text{pay}} = A_i - D_i$$

When the committee has assigned data, node i receives the following gross storage reward.

$$r_i = F_e \frac{A_i^{\text{pay}}}{\sum_j A_j}$$

The denominator includes all bytes assigned across the committee. Rewards withheld because of a refusal remain unpaid and enter a later epoch as carryover.

This makes payment follow physical responsibility. Larger spools carry more slice data and receive more reward weight. Inline tracks contribute their complete copies. A refusal removes the affected bytes from that node's payable weight.

Stake determines whether a node enters the committee and influences how many spools Tape Spooler assigns to it. Once assigned, rewards follow the bytes in those spools. Poor performance has an economic consequence through eviction: an evicted node loses its future spool assignments and the rewards attached to them. Holding or staking TAPE alone does not entitle a holder to protocol rewards. Storage compensation is earned by nodes for providing assigned storage services under the protocol.

Each node's reward flows through its staking pool. Delayed stake activation and withdrawal keep committee changes separated from the movement of stored data.

Storage and Service Revenue

Storage nodes, gateways, and CDNs provide different services through separate economic paths. The epoch reward pool compensates storage nodes for assigned data. Gateways and CDNs may charge customers for optional access and delivery services.

The protocol charges nothing per read and does not prescribe a gateway business model. For example, a gateway may charge users directly or add a service fee to a Solana transaction it submits on their behalf. A CDN may separately charge for high-volume delivery.

The design and pricing of gateway and CDN services are outside the scope of this paper. Tape payments fund long-term storage, assigned bytes determine node rewards, and customers separately pay for any additional services they choose.

15 Evaluation

We evaluate TAPEDRIVE at the object interface, measuring complete writes and direct reads on a live prototype. These operations include local preparation, transfer to or from storage nodes, Solana coordination, certification, reconstruction, and verification. The objective is to determine how the prototype

behaves across TAPEDRIVE's three object representations and which stages control completion time.

Experimental Setting

The measured network contained roughly 50 storage nodes running on modest cloud instances and coordinated through Solana's public devnet. Clients ran from matching cloud instances in `us-east1-c` and `europe-west3-a`. The campaign covered six payload sizes: 512 B, 1 MiB, 5 MiB, 50 MiB, 512 MiB, and 1 GiB. These sizes exercise inline tracks, coded tracks, and streams.

The campaign ran three iterations at each client and size using fresh tapes and tracks. We take the median result at each location, then report the median and range across the two locations.

A write begins when the client starts the operation and ends when certification becomes visible in a storage node's replayed state. Nodes used confirmed Solana commitment during the campaign. A direct read obtains committed data from the assigned spool group, verifies the returned material, reconstructs coded tracks where necessary, and compares the result byte-for-byte with the original payload. Every direct read in the campaign completed successfully and reproduced the original bytes.

End-to-End Operations

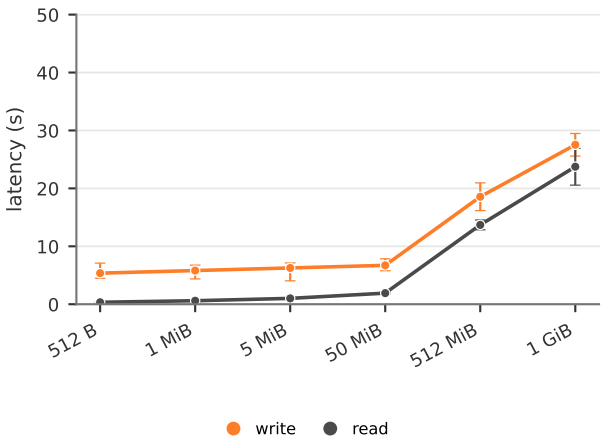


Figure 16: End-to-end write and read latency. The first four measurements are single-track operations dominated by fixed coordination. The 512 MiB and 1 GiB samples are streams of 64 MiB segment tracks, so latency rises as each operation processes more tracks. Whiskers summarize observed variation.

Across the single-track samples, write latency remains dominated by fixed coordination costs: the fitted trend rises gently from approximately 5.4 seconds at 512 B to 6.7 seconds at 50 MiB, with the per-size differences falling inside the observed run ranges. The next two measured sizes use the stream path. A 512 MiB stream contains eight 64 MiB segment tracks, while a 1 GiB stream contains sixteen, in both cases followed by a manifest track. The pipeline overlaps work across segments, so latency does not grow linearly with track count, but each additional track still adds encoding, transfer, registration, and certification work. Writes therefore increase to approximately 18.6 seconds at 512 MiB and 27.5 seconds at 1 GiB.

Direct reads complete in approximately 0.34 seconds at 512 B and 1.9 seconds at 50 MiB. Stream reads must resolve the manifest and gather enough slices for each segment track, increasing latency to approximately 13.7 seconds at 512 MiB and 23.7 seconds at 1 GiB. The geographic range also widens at these sizes because a direct reader must gather particular slices from their assigned storage nodes.

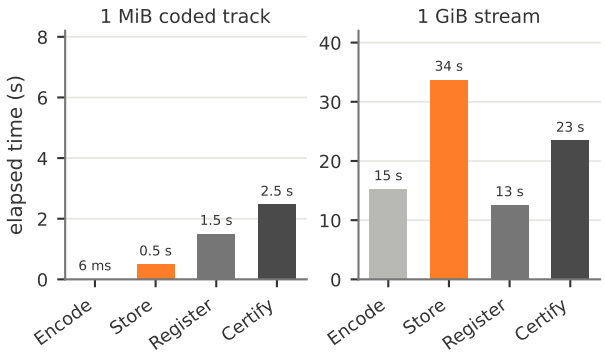


Figure 17: Major write phases for a 1 MiB coded track and a 1 GiB stream. Concurrent phases are shown separately and do not sum to wall time. Lower is better.

The phase breakdown in Figure 17 explains the transition. For ordinary coded tracks, local preparation finishes well before registration and certification, leaving Solana coordination as the principal fixed cost. For streams, preparation, distribution, registration, and certification overlap. Storage transfer becomes the longest write phase as payload size increases.

Direct reads show a similar division. Reconstruction remains smaller than slice download throughout the measured ladder. At 1 GiB, the median measured download phase is approximately 86 seconds, compared with approximately two seconds for reconstruction. The current prototype is therefore limited primarily by coordination for smaller writes and by storage-node transfer for larger operations, rather than by local coding.

These results characterize the complete prototype rather than any individual hashing or coding implementation. Changing those

implementations may alter the phase values, but the end-to-end measurement method remains the same.

Scaling to Many Petabytes

The campaign measures individual client operations against a live storage fleet. It does not measure aggregate throughput under increasing concurrency or operation latency at petabyte deployment sizes.

The protocol nevertheless provides a structural scaling property. An ordinary track is assigned to one spool group of fixed width n. A coded read needs k valid slices from that group, and a coded write distributes slices only to that group. Consequently, the storage-node fanout of an individual track operation depends on the coding profile and spool-group width, not on the total number of nodes or groups in the network.

Adding spool groups can therefore increase contributed capacity without lengthening the protocol path for an existing track. It also permits operations assigned to different groups to proceed on different storage resources. This does not guarantee linear throughput growth: realized performance still depends on assignment balance, demand concentration, node bandwidth, and the rate at which Solana can process registration and certification operations.

For coded payloads, logical capacity follows directly from contributed physical capacity. If storage nodes contribute a total of C_{physical} bytes and the selected profile stores r coded bytes per logical byte, then

$$C_{\text{logical}} \approx \frac{C_{\text{physical}}}{r}.$$

For illustration, readily rentable storage nodes range from four 22 TB drives to twenty-four 24 TB drives.[3] Across 1,000 nodes, these configurations provide 88–576 PB of physical capacity. At a representative $2.5\times$ coded-storage multiplier, that provides approximately 35–230 PB of logical coded capacity before metadata, inline replication, and operational headroom. It requires no change to the protocol. TAPEDRIVE is designed to scale horizontally: operators add conventional drives and storage nodes, the network adds spool groups, and every existing track retains the same fixed-width write, read, and recovery paths.

The evaluation establishes that the prototype completes and verifies operations from bytes through gigabytes on a roughly 50-node network, and that ordinary track operations remain group-local. Aggregate throughput under concurrency as the deployed network grows remain subjects for future measurement.

[3] These are ordinary bare-metal configurations, not exotic storage hardware. Hetzner lists a dedicated storage server with four 22 TB SATA drives, while OVHcloud offers storage servers with twenty-four 24 TB SAS drives [16, 17].

16 Related Work

Decentralized storage began with a narrower goal than cloud object storage. Freenet sought to keep publications from being suppressed, and BitTorrent showed that peers could distribute popular files without every transfer passing through the original publisher. These systems made dissemination harder to control, but a file remained available only while some participant chose to serve it [22].

IPFS made content addressing the common interface to this lineage. A file is represented as blocks in a Merkle directed acyclic graph, and its content identifier lets a reader verify the blocks regardless of which peer supplies them. A distributed hash table locates providers, while Bitswap exchanges the data. The identifier proves which bytes were requested; it does not make a peer store them. Publishers therefore pin content themselves or rely on a separate storage service [1]. The consequence is measurable. Half of IPFS peers stay online for less than four days, while networks that pay their operators sustain median uptimes of more than a month [40].

Sia made retention an agreement the blockchain could enforce. A client forms a storage contract with a host, the host periodically proves that it still holds the data, and the chain releases payment when the proof succeeds. Sia recommends spreading an erasure-coded file across several hosts because no single host can provide useful availability on its own. This turns spare disk capacity into an open market, but the renter software still chooses and maintains the set of host relationships that makes one file durable [23].

Filecoin expands this model into a storage market secured by two complementary proofs. Proof of Replication shows that a provider created a distinct physical encoding of the client's data. Proof of Spacetime shows that the provider kept that encoding for the agreed interval. These proofs let the chain verify retention without retrieving the complete file and let providers compete for large storage commitments [3, 24]. The protocol's unit is still a time-bounded deal with a provider. Applications can place intermediaries in front of that machinery, but the machinery remains part of making and renewing a durable write.

Arweave chooses a different contract with the user. Its Proof of Access lineage ties block production to access to earlier data, while an endowment funds continued storage from an initial payment. This makes permanence a property of the network rather than a sequence of renewable agreements. It is a strong fit for public records and archives that should outlive their publisher [2]. The same choice is a poor match for data with an ordinary end of life. Replacing an application object creates another permanent record, and deletion is outside the storage guarantee.

Erasure coding reduced the byte cost of decentralized stor-

age. Reed-Solomon codes established that any sufficient subset of coded pieces can reconstruct the original data, allowing a system to tolerate failures without storing many complete copies [9]. Replacing one missing piece may require downloading enough pieces to reconstruct the complete object. Regenerating codes study how to reduce this replacement traffic without surrendering the storage savings [12].

Storj applies conventional Reed-Solomon coding to a cloud-shaped object service. A coordinator selects storage nodes, distributes encrypted coded pieces, checks them, and arranges replacement. The design achieves low storage overhead across a broad set of independently operated nodes [25]. When one piece is lost, however, replacement transfers data proportional to the segment rather than the missing piece.

Walrus combines an existing coordination chain with a storage committee and a two-dimensional code. Its Red Stuff construction spreads each blob across the committee's shards, restores lost data with traffic proportional to the missing sliver rather than the complete blob, and supports storage challenges that remain sound in an asynchronous network [4]. A writer collects storage-node acknowledgments into an availability certificate on Sui, and a multi-stage epoch transition transfers shards between committees without closing reads or writes. Under its committee-wide Byzantine bound, Walrus preserves recoverability after the loss of a large fraction of all shards while keeping storage overhead far below complete committee replication [4].

That committee-wide scope also sets the width of every write. Each blob carries commitments for the Red Stuff positions, so more shards mean more metadata and coordination per blob. Walrus reduces the stored form of this list through another encoding step, but the fixed per-blob component remains visible for small writes [4].

Shelby takes the other modern branch. It uses Aptos for coordination, Clay codes for storage, a dedicated network backbone, and separate RPC and storage roles to target hot, read-intensive data. Clay codes are both maximum-distance-separable and bandwidth-optimal for single-piece repair: any sufficient set recovers the data, while the preferred repair path transfers less than conventional Reed-Solomon reconstruction [5, 13]. This lets Shelby publish storage overhead below two complete copies while rewarding both retention and fast delivery.

Shelby keeps each coded storage set narrow rather than stretching one code across its full provider network. A write and read therefore contact a bounded set even as the network grows. The corresponding durability statement belongs to that set, not to the complete committee. TAPEDRIVE makes a similar scaling choice with spool groups and adopts Clay repair, but spends more redundancy to keep a wider recovery margin inside each group. It separates serving from storage rather than adopting Shelby's dedicated delivery network.

Several other parts of TAPEDRIVE also descend from established distributed storage work. RADOS and Ceph separated logical data placement from particular machines, and CRUSH showed how every participant can compute a pseudorandom placement from common state without a central lookup table [26, 27, 28]. Ceph and the self-hosted object stores that followed it give one operator an S3-compatible store across the machines it controls. TAPEDRIVE runs in that setting as well, and carries the same placement ideas to owners no single party controls. Provable Data Possession and Proofs of Retrievability established that a client can sample an untrusted store rather than download a complete file to check it [29, 30]. Tape Spooler and TAPEDRIVE's challenge mechanism apply these ideas inside a permissionless, epoch-based committee.

Small files exposed fixed storage costs long before decentralized networks. Facebook built Haystack because storing billions of photos as ordinary files made metadata lookups consume more disk operations than reading the photos. Haystack placed many logical photos inside large append-only physical files and kept a compact index in memory. One application object no longer required one filesystem object [31]. Its successor f4 separated warm blobs from the hot path and applied erasure coding where the access pattern and object population justified it [32]. Both systems made the physical representation a storage-system decision.

Decentralized storage adds new fixed costs to the same old problem. A coded write may need a minimum coding unit, a commitment for every coded position, distribution to several nodes, acknowledgments, and a chain update. They do not shrink with a tiny object. Below the crossover derived for Tape Replay, the machinery can outweigh the data.

Walrus documents this effect directly. Its costs guide gives a one-megabyte example with about 4.5 MB of encoded payload and 64 MB of metadata, for roughly 68 MB of billed storage [33]. Quilt addresses the fixed cost by batching small blobs into one encoded unit. Walrus reports savings of more than 400 times for 10 KB blobs relative to storing them separately [10]. The saving is real and confirms that the per-blob floor, not the payload, dominates at that size.

Shelby reaches the same boundary through padding rather than commitment metadata. Its physical coding unit is roughly 10 MiB, and its paper states that smaller files are zero-padded to that size. Shelby therefore targets blobs at least as large as the coding unit and expects clients to combine smaller files [5]. The low overhead of its Clay profile is real once the unit is full; below it, the padding determines the effective overhead.

Batching is an effective answer when many objects can share one physical unit, but it amortizes the fixed cost of the coded

write rather than removing it, and it leaves the size decision with the application. TAPEDRIVE makes that distinction inside the storage protocol.

TAPEDRIVE builds on these systems. Its contribution is not one universally superior encoding or proof mechanism. Every network in this lineage chose a specialty, something unique to focus on. None chose convenience. Convenience is TAPEDRIVE's superpower.

17 Conclusion

Decentralized storage has shown that data can remain verifiable and available without being entrusted to one company. What it has not made routine is the act of using it. TAPEDRIVE starts from the premise that an application should see an object store, not a storage market or an encoding system. A tape gives the application capacity for a chosen lifetime and accepts objects in the form the application produces them.

The prototype shows that this abstraction works end to end. On a roughly 50-node network coordinated through Solana devnet, TAPEDRIVE certified writes and verified direct reads from 512 B through 1 GiB. The design also confines each ordinary track to one fixed-width spool group, so its storage-node path remains bounded as the network grows.

An application writes the data it has, at the size it has, and chooses how long to keep it. The network takes responsibility for keeping that data verifiable and available as operators and assignments change. The complexity still exists, but it no longer leaks through the interface or reshapes the application around the storage system. A team can start TAPEDRIVE on one machine or on infrastructure it already controls, and move to TAPENET without changing the application. Cloud storage became ubiquitous when applications stopped having to think about disks. TAPEDRIVE brings that expectation to a network no single provider controls. It makes decentralized storage boring.

18 References

[1] J. Benet. "IPFS: Content Addressed, Versioned, P2P File System." arXiv:1407.3561, 2014. https://arxiv.org/abs/1407.3561.

[2] Arweave. "Arweave: The Permanent Information Storage Protocol." Lightpaper. https://www.arweave.org/files/arweave-lightpaper.pdf.

[3] Protocol Labs. "Filecoin: A Decentralized Storage Network." Whitepaper, 2017. https://filecoin.io/filecoin.pdf.

[4] G. Danezis, G. Giuliari, E. Kokoris Kogias, M. Legner, J.-P. Smith, A. Sonnino, and K. Wüst. "Walrus: An Efficient Decentralized Storage Network." arXiv:2505.05370, 2025. https://arxiv.org/abs/2505.05370.

[5] G. Goren, A. Hariri, T. D. R. Hartley, R. Kappiyoor, A. Spiegelman, and D. Zmick. "Shelby: Decentralized Storage Designed to Serve." arXiv:2506.19233, 2025. https://arxiv.org/abs/2506.19233.

[6] A. Yakovenko. "Solana: A New Architecture for a High Performance Blockchain." Whitepaper v0.8.13, 2018. https://solana.com/solana-whitepaper.pdf.

[7] Solana Foundation. "RPC Methods," "Transaction Confirmation and Expiration," "Common JSON Data Structures," and "Programs." Solana documentation. https://solana.com/docs/rpc; https://solana.com/developers/guides/advanced/confirmation; https://solana.com/docs/rpc/json-structures; https://solana.com/docs/core/programs/program-execution. Retrieved July 16, 2026.

[8] D. Boneh, B. Lynn, and H. Shacham. "Short Signatures from the Weil Pairing." *ASIACRYPT 2001*, pp. 514–532, 2001. https://doi.org/10.1007/3-540-45682-1_30.

[9] I. S. Reed and G. Solomon. "Polynomial Codes Over Certain Finite Fields." *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. https://doi.org/10.1137/0108018.

[10] Walrus Foundation. "Next Walrus Evolution: Quilt Redefines Small File Storage at Scale." 2025. https://blog.walrus.xyz/introducing-quilt/; https://docs.wal.app/docs/system-overview/quilt. Retrieved July 19, 2026.

[11] F. B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." *ACM Computing Surveys*, 22(4):299–319, 1990. https://doi.org/10.1145/98163.98167.

[12] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. "Network Coding for Distributed Storage Systems." *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010. https://doi.org/10.1109/TIT.2010.2054295.

[13] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, S. Hussain, and S. Nandi. "Clay Codes: Moulding MDS Codes to Yield an MSR Code." *FAST '18*, pp. 139–154, 2018. https://www.usenix.org/conference/fast18/presentation/vajha.

[14] R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function." *CRYPTO '87*, pp. 369–378, 1987. https://doi.org/10.1007/3-540-48184-2_32.

[15] Amazon Web Services. "Amazon S3 Pricing." https://aws.amazon.com/s3/pricing/. Retrieved July 16, 2026.

[16] Hetzner Online GmbH. "Storage Servers: SX Line." https://www.hetzner.com/dedicated-rootserver/matrix-sx/. Retrieved August 3, 2026.

[17] OVHcloud. "Dedicated Server Prices: High Grade." https://www.ovhcloud.com/en/bare-metal/prices/?display=list&range=scale%7Chighgrade. Retrieved August 3, 2026.

[18] WorldStream B.V. "Storage Dedicated Servers." https://www.worldstream.com/en/dedicated-servers/storage/. Retrieved July 16, 2026.

[19] Scaleway. "Dedibox Store Dedicated Servers" and "Network Options for Dedibox Servers." https://www.scaleway.com/en/dedibox/store/; https://www.scaleway.com/en/dedibox/network/. Retrieved July 27, 2026.

[20] Spool Labs. "Tapedrive: Decentralized Object Storage." Source repository, snapshot `dd61c821aa3dc13c3023e523bb20abea2801a57f`. https://github.com/spool-labs/tape. Retrieved July 19, 2026.

[21] Spool Labs. "Clay: Coupled Layer Erasure Codes." Source repository,

snapshot 45b5523b3b40efe9339fb4e9291fdcc7303c8182. https://github.com/spool-labs/clay. Retrieved July 19, 2026.

[22] E. Daniel and F. Tschorsch. "IPFS and Friends: A Qualitative Comparison of Next Generation Peer-to-Peer Data Networks." *IEEE Communications Surveys & Tutorials*, 24(1):31–52, 2022. https://doi.org/10.1109/COMST.2022.3143147.

[23] D. Vorick and L. Champine. "Sia: Simple Decentralized Storage." Whitepaper, 2014. https://sia.tech/whitepaper.pdf.

[24] B. Fisch. "Tight Proofs of Space and Replication." *EUROCRYPT 2019*, pp. 324–348, 2019. https://doi.org/10.1007/978-3-030-17656-3_12.

[25] Storj Labs, Inc. "Storj: A Decentralized Cloud Storage Network Framework." Whitepaper v3, 2018. https://www.storj.io/storjv3.pdf.

[26] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters." *PDSW 2007*, 2007. https://ceph.com/assets/pdfs/weil-rados-pdsw07.pdf.

[27] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System." *OSDI '06*, pp. 307–320, 2006. https://www.usenix.org/legacy/event/osdi06/tech/weil.html.

[28] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data." *SC '06*, 2006. https://doi.org/10.1109/SC.2006.19.

[29] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. "Provable Data Possession at Untrusted Stores." *CCS '07*, pp. 598–609, 2007. https://doi.org/10.1145/1315245.1315318.

[30] A. Juels and B. S. Kaliski Jr. "PORs: Proofs of Retrievability for Large Files." *CCS '07*, pp. 584–597, 2007. https://doi.org/10.1145/1315245.1315317.

[31] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. "Finding a Needle in Haystack: Facebook's Photo Storage." *OSDI '10*, pp. 47–60, 2010. https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Beaver.pdf.

[32] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. "f4: Facebook's Warm BLOB Storage System." *OSDI '14*, pp. 383–398, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar.

[33] Mysten Labs. "Costs." Walrus Developer Guide, source snapshot 6aba4f7b87359f3d8be3ec57e09fbec7465e1d6e. https://github.com/MystenLabs/walrus/blob/6aba4f7b87359f3d8be3ec57e09fbec7465e1d6e/docs/book/dev-guide/costs.md?plain=1#L42-L44. Retrieved July 15, 2026.

[34] Metaplex Foundation. "Storing and Indexing NFT Data." Bubblegum V2 documentation. https://developers.metaplex.com/smart-contracts/bubblegum-v2/stored-nft-data. Retrieved July 21, 2026.

[35] I. Durovic. "SIMD-0341: v0 Account Compression." Draft Solana Improvement Document, source snapshot 944752816a6f06ddb6e712abc90cbef80557f80c, 2025. https://github.com/igor56D/solana-improvement-documents/blob/944752816a6f06ddb6e712abc90cbef80557f80c/proposals/0341-v0-account-compression.md; review discussion at https://github.com/solana-foundation/solana-improvement-documents/pull/341. Closed without merge January 16, 2026. Retrieved July 21, 2026.

[36] Google. "Google One Plans." https://one.google.com/about/plans. Retrieved July 27, 2026.

[37] Dropbox, Inc. "Dropbox Plans." https://www.dropbox.com/plans. Retrieved July 27, 2026.

[38] Google. "Cloud Storage Pricing." https://cloud.google.com/storage/pricing. Retrieved July 28, 2026.

[39] Microsoft. "Azure Blob Storage Pricing" and "Bandwidth Pricing." https://azure.microsoft.com/en-us/pricing/details/storage/blobs/; https://azure.microsoft.com/en-us/pricing/details/bandwidth/. Retrieved July 28, 2026.

[40] Z. Wu, B. Lin, N. Nikiforakis, and A. Balasubramanian. "Degrees of Decentralized Freedom: Comparing Modern Decentralized Storage Platforms." *IFIP TMA 2025*, 2025. https://tma.ifip.org/2025/wp-content/uploads/sites/14/2025/06/tma2025_paper16.pdf.

Appendix A: Detailed Algorithms

This appendix supplements the protocol description with the client, storage, and assignment procedures used by the correctness arguments.

Table 2: Main notation used by the detailed algorithms.

Notation	Meaning
tape, B, meta, x	Tape, payload, optional object metadata, and resulting track.
H, h_x	Cryptographic hash and the value hash registered for track x.
θ, s_i, proof$_i$	Track's registered encoding description, slice at position i, and its commitment proof.
C, C_x	Locally computed commitment and coded-track commitment registered for track x.
state$_e$, N_{e+1}	Replay-derived state at the epoch cutoff and next-epoch nonce.
$G(x), g, n$	Track x's spool group, number of live groups, and positions per group.
q, k, d	Agreement threshold, reconstruction threshold, and repair-helper count.
$D, \perp$	Set of verified slice responses and rejection of an invalid committed representation.
$\mathcal{C}, \mathcal{C}'$	Candidate committee and its eligible next-epoch members.
$W = (W_j)$	Per-position storage footprint of each spool group.
T	Target spool counts assigned to eligible nodes.
$O^{\text{old}}, O^{\text{new}}$	Previous and candidate ownership maps.
Assign$_j$, C_A proof$_j$	Assignment payload for group j, its proof, and the root committing all group payloads.
$\prec_{e+1}, v$	Nonce-derived node order and the node running synchronization.

Algorithm 1 Client operations

procedure WriteTrack(tape, B, meta)
  require Authorized(tape) $\wedge$ Live(tape)
  if InlineEligible(B, meta) **then**
    $x \leftarrow$ RegisterInline(tape, B, meta)
    await Finalized(x); **return** x
  $\theta \leftarrow$ EncodingDescription(B)
  $(s_1, \ldots, s_n, C) \leftarrow$ EncodeTrack(B, θ)
  $x \leftarrow$ RegisterCoded(tape, $H(B), \theta, C$, meta)
  $G \leftarrow$ AssignedGroup(x)
  for $i \in [1, n]$, set proof$_i \leftarrow$ Proof(C, i)
    send $(x, i, s_i,$ proof$_i)$ to Owner(G, i)
  $\mathcal{A}_{\text{ack}} \leftarrow$*await* q distinct valid acknowledgements
  SubmitCertificate(x, Aggregate($\mathcal{A}_{\text{ack}}$))
  await Certified(x); **return** x

procedure ReadTrack(x)
  require Live(x) $\wedge$ Certified(x)
  $G \leftarrow$ AssignedGroup(x)
  if Inline(x), *await* B from G with $H(B) = h_x$; **return** B
  $D \leftarrow$*await* k distinct responses satisfying VerifySlice
  return DecodeTrack(x, D)

Algorithm 2 Coding and recovery helpers

procedure EncodeTrack(B, θ)
  $(s_1, \ldots, s_n) \leftarrow$ Encode$_\theta(B)$
  $C \leftarrow$ Commit$_\theta(s_1, \ldots, s_n)$
  return $(s_1, \ldots, s_n, C)$

procedure VerifySlice(x, i, s, proof)
  return VerifyMerkle(C_x, i, Root(s), proof)

procedure DecodeTrack(x, D)
  $\theta \leftarrow$ EncodingDescription(x)
  $B \leftarrow$ Decode$_\theta(D)$
  $(s_1, \ldots, s_n, C) \leftarrow$ EncodeTrack(B, θ)
  if $C \neq C_x \vee H(B) \neq h_x$ **then return** $\perp$
  return B

procedure RecoverPosition(x, i)
  $s_i \leftarrow$ ClayRepair(x, i, d helpers)
  proof$_i \leftarrow$ CommitmentProof(x, i)
  if VerifySlice($x, i, s_i,$ proof$_i$) **then return** s_i
  $D \leftarrow$*await* k distinct responses satisfying VerifySlice
  $B \leftarrow$ DecodeTrack(x, D); **if** $B = \perp$ **then return** $\perp$
  $\theta \leftarrow$ EncodingDescription(x)
  $(s_1, \ldots, s_n, _) \leftarrow$ EncodeTrack(B, θ); **return** s_i

Algorithm 3 Storage-node operations

procedure ObserveInline(x, B)
  if Finalized(x) $\wedge$ Owns($G(x)$) $\wedge H(B) = h_x$, Persist(x, B)

procedure StoreSlice(x, i, s, proof)
  if $\neg$ Registered(x) $\vee \neg$ Owns($G(x), i$) **then return** $\perp$
  if $\neg$ VerifySlice(x, i, s, proof) **then return** $\perp$
  Persist(x, s)
  return Sign($x, C_x, G(x), E(x)$)

procedure Serve(x, i)
  if Inline(x) **then return** StoredPayload(x)
  return (StoredSlice(x, i), StoredProof(x, i))

procedure Synchronize(p)
  for each live track x assigned to $G(p)$, transfer the representation for Position(p)
  if absent or invalid, run RecoverPosition(x, Position(p))
  attest ready only when p is active and no recovery remains pending

Algorithm 4 Tape Spooler

procedure BuildAssignment(state$_e$, $\mathcal{C}$, N_{e+1}, g)
  $W \leftarrow$ ActiveSpoolFootprints(state$_e$, g)
  $\mathcal{C}' \leftarrow$ EligibleNextCommittee($\mathcal{C}$)
  $T \leftarrow$ StakeTargets($\mathcal{C}', W$)
  $O^{\text{old}} \leftarrow$ CurrentOwners(state$_e$, g)
  $O^{\text{new}} \leftarrow$ Migrate($O^{\text{old}}, T, W, N_{e+1}$)
  require ValidAssignment(O^{new}, T, W)
  for $j \in [1, g]$, Assign$_j \leftarrow (j, O^{\text{new}}_j, W_j,$ BlacklistedWeights(j))
  $C_A \leftarrow$ MerkleRoot(H(Assign$_1$), $\ldots, H$(Assign$_g$))
  return $(C_A, \{\text{Assign}_j\}_{j=1}^{g},$ Proofs(C_A))

procedure Migrate($O^{\text{old}}, T, W, N_{e+1}$)
  $\prec_{e+1} \leftarrow$ DeterministicNodeOrder(N_{e+1})
  $O^{\text{new}} \leftarrow$ RetainCompatible($O^{\text{old}}, T, \prec_{e+1}$)
  for $j \in [1, g]$:
    $U \leftarrow$ MustTake(O^{new}, T, j)
    $O^{\text{new}}_j \leftarrow$ MakeRoom(O^{new}_j, U)
    $O^{\text{new}}_j \leftarrow$ FillByTargetDeficit($O^{\text{new}}_j, T, \prec_{e+1}$)
  require UniquePerGroup(O^{new}) $\wedge$ WithinCapacity(O^{new}, W)
  return O^{new}

procedure ApproveAndSynchronize($C_A, \{\text{Assign}_j, \text{proof}_j\}_{j=1}^{g}, v$)
  for each current G_j, *await* VoteCount(G_j, C_A) $\geq f + 1$
  RecordAssignment(C_A)
  for $j \in [1, g]$, require VerifyMerkle(C_A, Assign$_j$, proof$_j$); Materialize(Assign$_j$)
  for each target j, *await* Materialized(j); AdvanceEpoch()
  for $p \in$ AssignedSpools(v), Synchronize(p)
  for each G_j, *await* ReadyCount(G_j) $\geq$ max($k,\ f + 1$); CompleteSync()

Appendix B: Parameter Relationships

Let n be the number of positions in a Clay profile, k its reconstruction threshold and data-symbol count, d the helper count for repairing one position, f the maximum Byzantine positions in a group, and q the agreement threshold. These integer parameters satisfy

$$0 \le f < n, \qquad 1 \le k \le q \le n, \qquad k \le d \le n-1.$$

Honest-only progress and recovery from a weakest certificate require

$$q \le n - f \qquad \text{and} \qquad k \le q - f.$$

The first condition permits progress using honest owners alone. The second guarantees at least k honest acknowledgements in every certificate. After complete distribution, the code tolerates $n - k$ unavailable positions. Eviction and invalidation rely on intersecting quorums and additionally require

$$2q > n + f.$$

Any two quorums then share an honest signer, and honest owners do not sign conflicting results for the same vote, so no two conflicting results can both be certified. Assignment and snapshot approval attest a value every honest owner recomputes from the same finalized state; a wrong value gathers at most f signatures, so the attestation threshold $f+1$ suffices and tolerates $n-f-1$ silent owners. A group completes Sync after $\max(k,\ f+1)$ readiness reports, enough that a handoff whose missing owners crashed leaves a reconstructable group, while owners that will never report cannot block the reassignment that replaces them.

Before metadata and padding, the coded-byte multiplier is

$$\frac{n}{k}.$$

For ideal minimum-storage regenerating repair, including Clay, the traffic needed to restore one position is

$$\frac{d}{d-k+1} \quad \text{slice-equivalents.}$$

At fixed n, q, and f, the boundary $k = q - f$ spends the complete certificate margin; raising k further makes reconstruction depend on availability beyond the certificate guarantee. Selecting k and d therefore trades coded bytes, certificate margin, loss tolerance, repair traffic, subpacketization, and measured coding performance.

Appendix C: Legal Disclaimer

Nothing in this paper is an offer to sell, or the solicitation of an offer to buy, TAPE or any other token, security, or financial instrument, in any jurisdiction. This paper is not a prospectus, offering memorandum, or disclosure document of any kind. We publish it to describe the design of TAPEDRIVE and to receive feedback and comments from the public. The paper describes the intended function of TAPE within the protocol. It does not describe the initial creation, allocation, offering, or sale of any token, and no decision has been made about whether, when, how, or by what entity any of that would occur. Nothing in this paper commits the authors, or any entity affiliated with them, to create, distribute, or sell any token, or to conduct any offering. Any offering that ever does occur would be governed by its own definitive documents and by the laws that apply to it at that time.

Nothing in this paper should be treated or read as a guarantee or promise of how TAPEDRIVE will develop, of the utility or value of TAPE, or of the performance, availability, durability, cost, or rewards of the network. Where this paper uses the word guarantee, it refers to a mathematical property that holds under the stated assumptions, not to a warranty of service. This paper describes a protocol design that may change during development or through applicable protocol governance, and whose success depends on many factors outside our control, including the actions of independent node operators, conditions on Solana, and conditions in the broader data-storage and cryptocurrency industries. Storage prices, storage payments, staking rewards, and other economic quantities described here depend on protocol parameters, operator votes, and market conditions; they may differ from any figure or formula shown and may not materialize at all. Statements about future events reflect our analysis of the issues described in this paper, and that analysis may prove incorrect.

The measurements in this paper record what we observed at the time of writing, on specific hardware, software versions, and network conditions. The end-to-end results were taken on a development network; a production deployment may behave differently. We collected and reported these figures in good faith, and they may nonetheless contain mistakes in measurement, methodology, or transcription. We do not claim that they are accurate or complete, and none of them is a promise of the performance any user will experience. Comparisons with other systems and providers rely on their published documentation, the releases we measured, and list prices at the time of writing. Those projects and providers continue to change, so a comparison that was accurate when made may not describe their current software, performance, or pricing. Third-party names and marks belong to their respective owners; their use here is for identification, and no affiliation or endorsement is implied. We intend to make a reasonable effort to correct material errors that come to our attention. We undertake no duty to do so, or to update this paper as our own results or those of others change.

This paper is not investment advice, and it is not legal, financial, accounting, or tax advice.