# Atlas Protocol: A High-Throughput DePIN Storage Network Built on CometBFT

**Subay A. Moussa**

Technical Whitepaper

March 14, 2026
Rev. –

**Abstract**

Atlas Protocol is a Layer 1 CometBFT-based blockchain, purpose-built for decentralized storage at planetary scale. The protocol introduces a novel multi-dimensional storage credit mechanism that represents usage in byte-seconds rather than bytes. This enables fair reward distribution proportional to actual storage provided while eliminating complex per-user accounting. A deterministic proof-of-persistence challenge generation mechanism uses on-chain entropy to create unpredictable Merkle proof challenges without oracle dependencies for verifiable randomness. A hybrid Merkle tree construction combines cryptographic Blake3 leaf hashes with high-throughput XXH3 internal nodes, achieving up to 10x faster proof computation and 4x lower memory consumption during validation compared to the fastest single-thread fully cryptographic approaches.

The network separates control plane from data plane: the blockchain indexes files, manages subscriptions, issues challenges, and distributes rewards, while providers handle the raw bytes off-chain. Overall, this design supports millions of files, thousands of providers, and continuous verifiability without congesting the chain. Atlas Protocol demonstrates that decentralized storage need not choose between scalability and trust.

## Contents

# 1 Introduction

## 1.1 Motivation

Decentralized storage represents a fundamental shift in how data is persisted, accessed, and owned across the internet. Unlike traditional cloud storage—where a single entity controls availability, pricing, and access policies, decentralized storage networks distribute data across independently operated nodes, removing central points of failure and censorship. Users retain cryptographic ownership of their files, providers compete openly to offer capacity, and trust is established through verifiable proofs rather than service-level agreements. This model not only enhances data resilience and sovereignty but also creates a more competitive marketplace that can drive down costs over time.

Realizing these benefits at scale requires overcoming significant technical hurdles: proof mechanisms must be efficient enough to handle millions if not billions of files without congesting the network, reward systems must fairly compensate providers without complex accounting systems, and the underlying data structures must enable fast verification on commodity hardware. Atlas Protocol attempts to address these challenges head-on while preserving the core principles of decentralized storage; that data should remain accessible, verifiable, and free from unilateral control. Existing designs face the following limitations that our approach seeks to resolve:

- **Proof congestion**: Proof-of-storage mechanisms can saturate blockspace once the number of stored files reaches hundreds of thousands or millions.
- **Ineffective reward distribution**: Many systems tie rewards to per-payment or per-user gauges, making reward computation scale poorly with network size.
- **Suboptimal data structures**: Complex proofs, Merkle trees, data serialization, and large proof payloads limit proof throughput on commodity hardware.

## 1.2 Objectives

- **Scalability**: Support millions of stored files without congestion.
- **Verifiability**: Provide proof and confidence that providers continuously store user data.
- **Fairness**: Distribute rewards proportionally to actual storage usage over time, represented as byte-seconds.
- **Simplicity of integration**: Enable clients to integrate via simple gRPC/REST APIs.
- **Maintainability**: Allow protocol components (reward parameters, proof cadence, etc.) to evolve without breaking core invariants.

## 1.3 Design Overview

At its core, Atlas Protocol is a Proof of Stake (PoS) blockchain, which is a distributed ledger secured by economic stake rather than computational work. In PoS systems, a set of validators lock up (or "stake") tokens as collateral to participate in consensus. These validators take turns proposing and voting on blocks, with their voting power weighted by the size of their stake. If validators behave maliciously or go offline when expected to participate their stake is partially "slashed", creating a strong economic incentive to follow protocol rules. This model offers several advantages over Proof of Work: it consumes negligible energy, enables fast block finality, and allows the chain to encode complex application logic directly into the consensus layer. The native staking and fee token is denoted by `ATL`, with its micro-denomination being `uatl`.

Atlas Protocol builds on this foundation using the CometBFT consensus engine and Cosmos SDK framework. CometBFT provides Byzantine Fault Tolerant (BFT) consensus, meaning the network can tolerate up to one-third of validators by voting power behaving maliciously while still producing blocks and finalizing state. The Cosmos SDK provides a modular architecture where application-specific logic lives in separate modules that interact with the underlying chain. Atlas Protocol decomposes its functionalities into several modules (see Section 2.2).

# 2 System Architecture

## 2.1 Network Model

Atlas Protocol consists of four main classes of actors illustrated in Fig. 1:

- **Validators**: Consensus nodes that secure the Byzantine-fault tolerant blockchain network by processing transactions, finalizing blocks, and enforcing protocol rules.
- **Providers**: Storage nodes that commit capacity, store and provide files, respond to proof challenges, and receive rewards. Files are replicated accross several providers to ensure file security and redundancy.
- **Clients**: End-users or applications that download files from storage providers and use the blockchain as an index.
- **Users**: Clients that purchase storage, upload or delete their files, manage file permissions, and participate in protocol governance.

The control plane (subscriptions, file indices, challenges, rewards) lives entirely on-chain. The data plane (file bytes) is stored off-chain by providers. It is important to note that a "file" does not need to be uploaded. In this document, a "file" represents any collection of data.

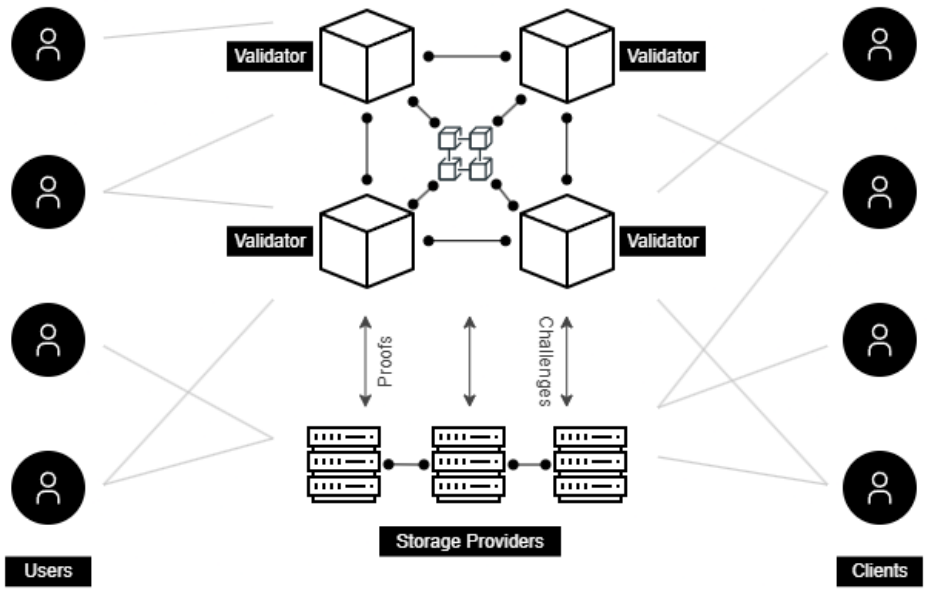


Figure 1: x/storage MsgBuyStorage top-level overview

## 2.2 Module Overview

### 2.2.1 `x/storage`

The `x/storage` module is responsible for:

- Storage accounting using byte-second credits.
- Tracking user storage subscriptions.
- Registering and managing storage providers.
- Indexing files and their replication across providers.
- Issuing and verifying proofs-of-persistence.
- Computing and distributing provider rewards.

### 2.2.2 `x/filetree`

The `x/filetree` module provides an optional logical file system abstraction over the lower-level file identifiers tracked by `x/storage`. It enables:

- Hierarchical paths and directories.
- Mapping from human-readable paths to underlying file IDs.
- Application-friendly queries for navigation and discovery.

### 2.2.3 `x/oracle`

The `x/oracle` module provides a mechanism for determining the value of the ATL utility token, using data from markets external to the protocol.

- Provides the ATL price used to parameterize storage costs and other economic functions.
- Is designed to be extensible for potential data-oriented use-cases.

# 3 Storage Credits and Subscriptions

## 3.1 Byte-Second Representation

Atlas Protocol represents storage usage in units of byte-seconds. One storage credit corresponds to storing one byte for one second. This model has several advantages:

- **Granular billing**: directly couples data size and retention duration.
- **Reward fairness**: provider rewards are proportional to accumulated byte-seconds provided.
- **Pooling**: credits can be aggregated in a single on-chain pool instead of many per-user gauges.

Let $S$ be the size of a file, $R$ be the number of file replicas, and $T$ the duration in seconds. The total credits minted are:

$$C = R \times S \times T.$$

### 3.2 Subscription Model

Users reserve storage capacity through subscriptions. A subscription reserves a given amount of capacity for a fixed period of time. A user can also purchase subscriptions for other users. Lastly, a user can optionally specify a default subscription to use for file uploads. This cannot be done on the behalf of another user. Every subscription can have one of 3 states:

- **ACTIVE**: The subscription is usable; new files can be posted against it.
- **PENDING**: The subscription was purchased for another user, but not yet accepted. This state exists to prevent harassment from a malicious actor mass-creating subscriptions of the smallest denomination for an unwilling user.
- **EXPIRED**: The subscription's end time has passed; it remains in the appstore for historical indexing, but no new usage is allowed.

A new subscription is purchased by broadcasting a transaction with a `BuyStorage` message. The standard replication factor is 3, hence the real allocated capacity is three times the capacity purchased by the client. At the time of the transaction, the buyer must have sufficient balance for the desired capacity and duration. At the time of writing, a subscription costs $12/TB/mo, though this can be changed through governance proposals. An overview of how the `x/storage` module handles a `BuyStorage` message is illustrated in Fig. 2.

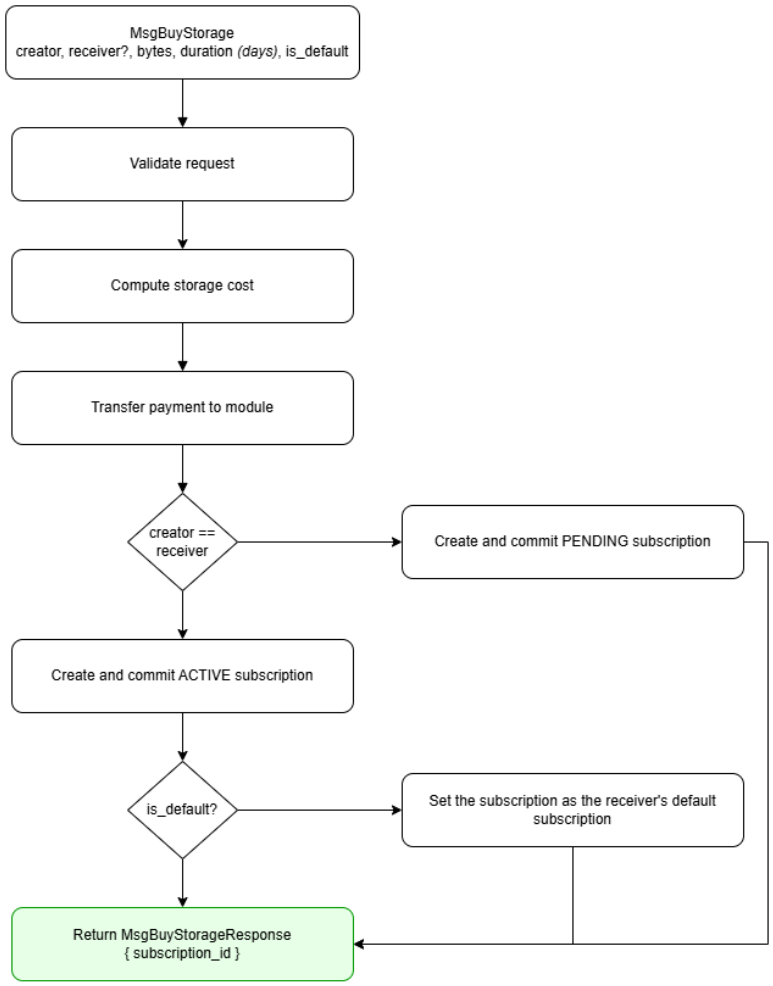


Figure 2: x/storage MsgBuyStorage top-level overview

### 3.3 Storage Credit Mechanism

The `x/storage` module maintains a storage credit pool. When a new subscription is created, storage credits representing the purchased capacity and duration are "minted" in the pool. The buyer's `$ATL` tokens are transfered to a token reward pool that mirrors the storage credit pool. As illustrated in Fig. 3, storage credits are never distributed from the pool, but they are allocated. Atlas Protocol keeps track of each subscription's maximum allocation of the pool and how many credits they have used of their allocation. This is analogous to partitioning a disk drive, but in both physical and time domains.

When a file is uploaded, its corresponding subscription's available credits is decremented by `file_size` $\times$ `replicas` $\times \Delta t$, where $\Delta t$ is the remaining subscription duration in seconds. When a file is deleted, the available credits is incremented by the same formula. Residual storage credits from an expired subscription are translated into tokens that are moved into protocol-owned liquidity (POL). This liquidity is governed by the protocol's users. Storage provider reward distribution via storage credits is discussed in Section 8.

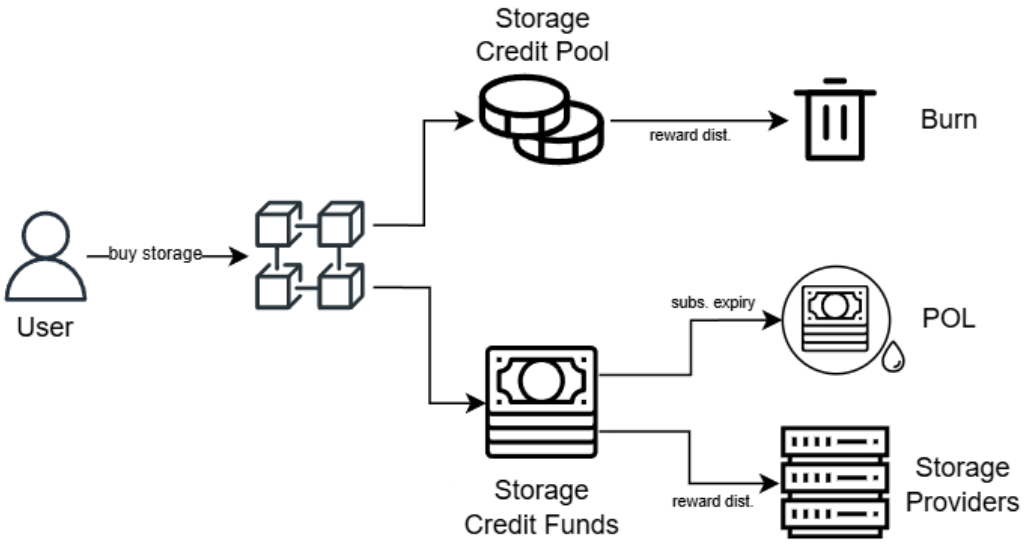


Figure 3: Storage credit flow

### 3.4 Future Developments

- **Subscription expansion**: Increasing the capacity or duration of an existing subscription.
- **On-demand subscriptions**: On-demand credit-based subscriptions with no expiry.

# 4 Storage Providers

## 4.1 Provider Registration

To join the network, providers must register capacity and bond a deposit by broadcasting a `RegisterProvider` message. The deposit is specified by chain paramaters and can be modified through governance. Fig. 4 presents an overview of the `RegisterProvider` message on-chain. A provider may update its details or maximum capacity on-chain by broadcasting an `UpdateProvider` message. Fig. 5 presents an overview of the `UpdateProvider` message on-chain.

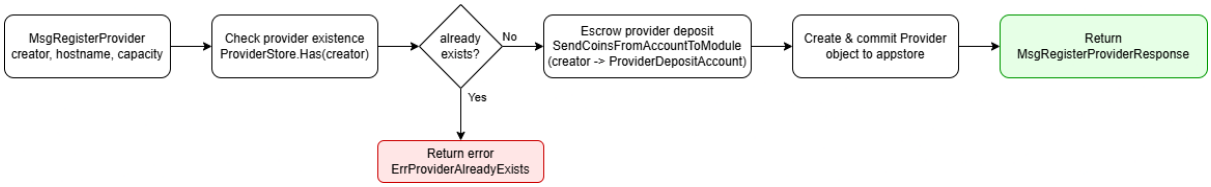


Figure 4: x/storage MsgRegisterProvider top-level overview

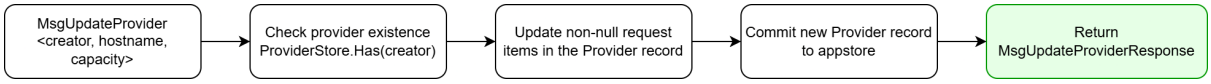


Figure 5: x/storage MsgUpdateProvider top-level overview

## 4.2 Provider Lifecycle

A provider has three main functions: commiting files, serving files, and responding to proof-of-persistence (PoP) challenges. Upon file upload, the provider verifies that the file was previously posted on-chain, caches the metadata, and tells the blockchain it has recieved the file (see Section 5.2). Providers serve files by responding to GET requests on its `/upload` endpoint, provided the file id.

The provider is always listening for new challenges from the blockchain demanding that it proves it has the file specified by the challenge request. If a provider fails to prove it has a file, its future rewards may be slashed. The provider is also always listening for files that get deleted. If a provider misses a file delete event (e.g. due to connection issues, temporarily offline), it would eventually detect the file was deleted by identifying which files have not been challenged for an extended period of time.

To sunset operations, a provider must broadcast an `UnregisterProvider` message. The initial provider deposit will be return to the provider's address, and the provider will be stripped from the records of all the files it was serving.

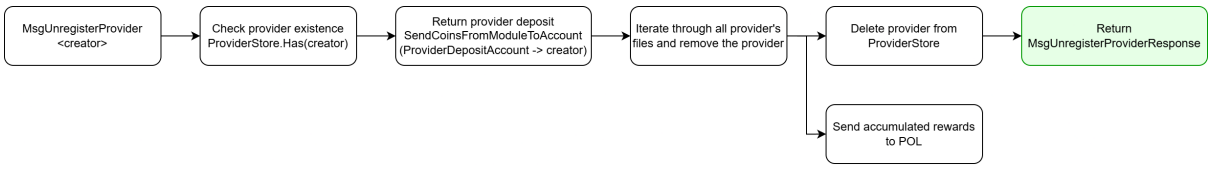


Figure 6: x/storage MsgUnregisterProvider top-level overview

# 5 File Management

## 5.1 File Identification and Indexing

Files can be referenced on both the control plane (blockchain) and data plane (providers) using a unique file identifier (FID). The standard construction of an FID is:

$$\texttt{fid} = \text{SHA256}(\text{merkle\_root} \,\|\, \text{creator} \,\|\, \text{nonce}),$$

where:

- `merkle_root` is the root hash of a Merkle tree representing the file (see Section 7),
- `creator` is the address that uploaded the file,
- `nonce` disambiguates multiple uploads of the same content.

On-chain, files are keyed by a composite tuple (creator, subscription_id, `fid`), with a secondary index mapping FID to its full key for efficient lookup. The total active file count is also tracked by the storage module.

## 5.2 File Upload Workflow

The high-level workflow of a file upload is:

1. **Merkle tree preparation**: A user splits the file into fixed-size chunks and constructs a Merkle tree using a hybrid hash scheme (Section 7.1).
2. **On-chain posting**: The user broadcasts `MsgPostFile`. The `x/storage` module updates the corresponding subscription's space used and creates and stores a STAGED file record.
3. **Off-chain upload**: The user uploads the file to one or more providers.
4. **Provider attestation**: A select provider recomputes the Merkle tree and submits a proof (i.e. chunk data and path) using a `MsgProveFile` message, referencing the file's FID.
5. **Activation**: On a valid proof, the `x/storage` module appends the provider to the file's list of providers, sets file status to `ACTIVE`, and increments the total file count index.

If a file has fewer providers than its desired replication factor, it is considered a stray, and additional providers may adopt it by proving they hold the data.

## 5.3 File Delete Workflow

The high-level workflow of a file delete is:

1. **Initiate delete**: A user broadcasts `MsgDeleteFile`.
2. **Remove file**: If the file exists, the indexed record representing the file on-chain is removed.
3. **Modify storage stats**: Decrement each provider's used space by the file size. Update subscription credit usage accordingly.
4. **Apply credit delta**: The amount of credits consumed during the active reward window needs to be added to the providers' rewards at the end of the window (see Section 8).
5. **File delete**: The file's providers' receive a blockchain event indicating that the file has been deleted. The providers delete the file data from their local storage.

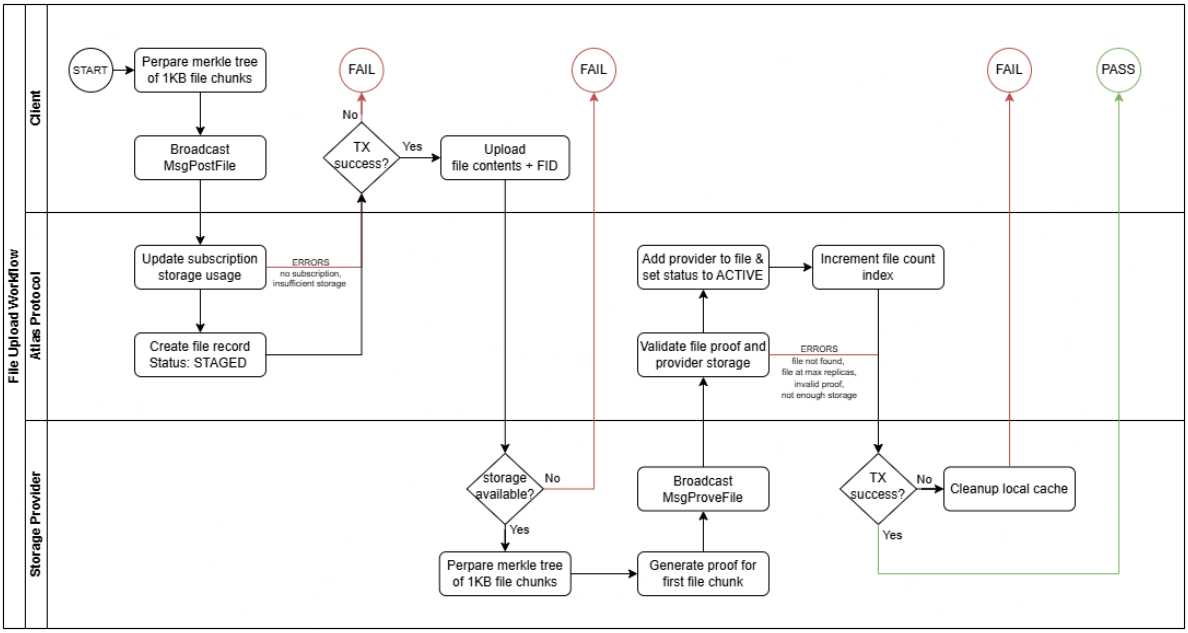


Figure 7: File upload workflow.

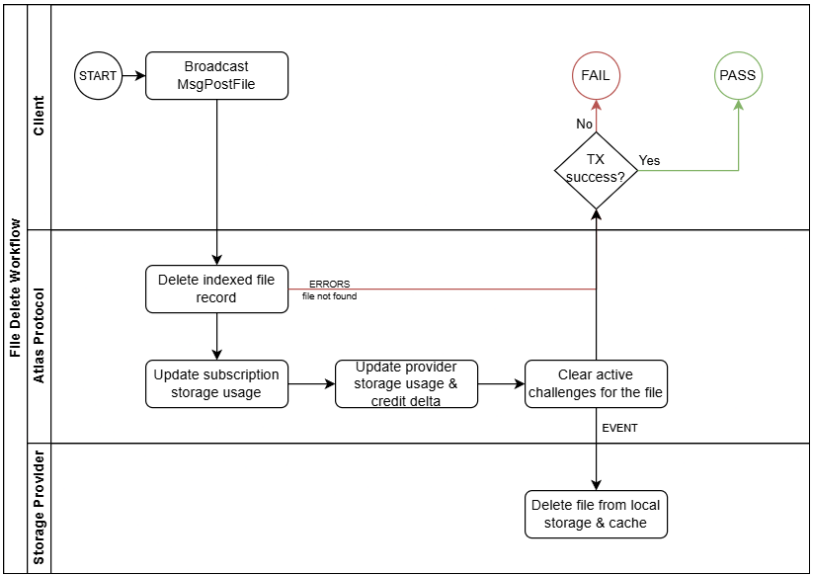


Figure 8: File delete workflow.

# 6 Filetree Module

While the `x/storage` module is concerned with the physical layout of files across providers and subscriptions, most applications and end-users think in terms of logical paths, directories, and shared workspaces. The `x/filetree` module bridges this gap by providing a hierarchical namespace that sits on top of the file identifiers (FIDs) managed by `x/storage`.

## 6.1 Design Goals

The `x/filetree` module is designed with the following objectives:

- **Human-friendly organization**: expose files via paths such as `/users/alice/photos/2025/trip.jpg` instead of opaque FIDs.

- **Separation of concerns**: decouple logical naming, sharing, and access semantics from the underlying storage and proof-of-persistence machinery.
- **Efficient lookups**: enable constant or logarithmic-time resolution from path to FID, even as the number of files grows to millions.
- **Minimal on-chain footprint**: store only the metadata required to reconstruct namespaces and enforce permissions, leaving content and heavy indices off-chain where possible.

### 6.2 Data Model

At a high level, the `x/filetree` module maintains a tree of nodes, where each node represents either a directory or a leaf file reference:

- **Directory nodes**: contain a parent reference and a mapping from child names to child node identifiers.
- **File nodes**: hold a pointer to the underlying FID in `x/storage`, along with ownership and permission metadata.
- **Roots and workspaces**: multiple independent roots can coexist (e.g. per-user roots, shared team spaces), each identified by a stable namespace identifier.

Formally, resolving a path proceeds as a sequence of map lookups starting from a root:

$$\texttt{resolve(root, [s\_1, ..., s\_k])} \rightarrow \texttt{fid}$$

where $[s_1, \ldots, s_k]$ are the path segments and `fid` is the final file identifier in `x/storage`. If the last node is a directory, the same procedure is used to support directory listing and tree traversal APIs.

### 6.3 Access Control and Sharing

The `x/filetree` module is also the natural place to attach access control semantics. While Atlas Protocol does not prescribe a single global ACL model, the module is structured to support node-level ownership, delegated access, and shared namespaces within a unified framework. Each node may specify an owner address that has full control over renames, moves, and deletions, enabling clear authority over the evolution of a subtree. On top of this, applications can layer optional reader or writer lists, role-based policies, or capability tokens that are interpreted by off-chain clients to decide who can see or modify a given path. Finally, team or application roots can be modeled as shared namespaces where multiple principals share administrative privileges over a subtree, allowing collaborative workspaces to be expressed as first-class parts of the file tree.

These semantics are enforced at the control plane level: providers are agnostic to logical paths and only see FIDs, while users and applications interact almost entirely with paths and high-level APIs. This design cleanly separates cryptographic storage guarantees from user experience and collaboration concerns.

# 7 Proof-of-Persistence Mechanism

## 7.1 Merkle Tree Construction

Atlas Protocol uses a two-layer hybrid hash scheme for Merkle trees:

- Leaf hashes: a Blake3 cryptographic hash of each 1KiB data chunk of a file, providing pre-image resistance.
- Internal nodes: high-throughput XXH3 hashes, which is non-cryptographic but safe when leaves are already cryptographically hashed.

File chunk sizes are limited to 1024 bytes (1KiB) to prevent saturation of the 1MB block-space used by Atlas Protocol.

## 7.2 Blake-XXH Hybrid Merkle Hash Scheme

XXH3-128bit hashing of Merkle tree nodes supports scalability for hundreds of millions of files and maintains reasonable upload speeds across a wide range of hardware. Following the birthday paradox, the 50% collision probability of this hash algorithm is $2^{64}$. This hash strategy on its own is not considered cryptographically secure; however, using cryptographically secure Blake3 hashes for the Merkle leaf nodes introduces entropy into the Merkle tree, making preimage attacks unrealistic for proofs. To further enhance security, domain separation between leaf nodes and internal nodes is implemented using a domain identification byte prefix on every node. Each leaf node is also associated with a leaf number. For an odd number of leaves, the last leaf is duplicated to prevent second-preimage proof attacks. As a result, the worst-case proof forging resistance for the smallest file unit is $2^{128}$, which is equivalent to the collision resistance of a single SHA-3-256bit hash. The real effort to forge a proof without having the raw data at hand scales exponentially with the Merkle tree depth. The worst-case real effort to required to forge a proof becomes $2^{(d+1)\times 64}$, where `d` is the Merkle tree depth. The depth can be calculated using Eq. (1):

$$d = \left\lceil \log_2 \left( \lceil \frac{S}{1024} \rceil \right) \right\rceil \tag{1}$$

where `S` is the file size in bytes.

This design provides:

- **Security**: The Merkle root remains binding to the original data thanks to cryptographic leaves.
- **Performance**: Internal node computation can be up to $\sim 10\times$ faster than using a full cryptographic hash tree, yielding very high proof throughput. Testing has also unexpectedly revealed that the memory consumption during proof validation is up to 4x less than a standard Blake3 Merkle proof.

## 7.3 Challenge Windows and Rounds

All files are proven within a challenge window with a length defined by the `ProofWindowBlocks` storage module parameter. The proof window is divided into challenge rounds, each with a length defined by the `ProofRoundBlocks` storage module parameter. A challenge is a request for a provider to submit a proof for a file using a specified chunk. At the end of every `ProofRoundBlocks` blocks, Atlas Protocol generates new challenges for a different subset of files. File challenges are evenly distributed in challenge rounds over the duration of a challenge

window to prevent uneven chain congestion. The distribution is enforced by Eq. 2, which specifies how many files should be challenged at the specified block.

$$f(x) = \text{ceil}\left(\frac{\left(\text{floor}\left(\frac{x}{R}\right)+1\right)\cdot N\cdot R}{G}\right) - \text{ceil}\left(\frac{\text{floor}\left(\frac{x}{R}\right)\cdot N\cdot R}{G}\right) \tag{2}$$

$$G = \texttt{ProofWindowBlocks} \text{ — } R = \texttt{ProofRoundBlocks} \text{ — } N = \texttt{\# Files}$$

Files are selected using from iteration cursor preserved accross rounds, with iteration direction alternating (ascending/descending) between windows to avoid bias.

### 7.4 Proof Chunk Randomness

The security of any proof-of-storage system hinges on unpredictable challenge generation. If providers could anticipate which chunks would be challenged, they could pre-compute the proofs of every upcoming challenge round without having to store the file, defeating the purpose of verifiable storage. Atlas Protocol generates deterministic, verifiable randomness for each provider-file combination using only on-chain data, eliminating the need for interactive VRF protocols or oracle dependencies while maintaining unpredictability until challenge rounds finalize.

For each active file and each provider storing that file, the protocol derives a challenge chunk index using a multi-source seed that combines several entropy sources:

$$\text{chunk} = \text{XXH3}(\text{prev\_block\_hash} \,\|\, \text{fid} \,\|\, \text{round\_number} \,\|\, \text{provider\_array\_index}) \bmod \text{total\_chunks}$$

The seed components serve distinct purposes:

- Previous block hash: Provides unpredictability, as the exact hash is unknown until the previous block is finalized. This prevents providers from pre-computing challenges far in advance.
- File identifier (FID): Ensures challenges are file-specific. Without this, all files would share the same challenge pattern per round.
- Round number: Guarantees that challenges change each round, even for the same file and provider.
- Provider position: Creates provider-specific indices for the same file. Since files are replicated across multiple providers, each must prove a different chunk to prevent collusive storage sharing.

### 7.5 Challenge Response and Verification

Once challenges are generated at the beginning of a proof round, providers must respond with cryptographic proofs that demonstrate continued possession of the challenged data. The response and verification workflow is designed to minimize on-chain computation while maintaining strong security guarantees. Providers respond to challenges by broadcasting a `MsgProveFile` transaction containing the following components:

- **Challenge identifier**: A reference to the specific challenge being answered, comprising the file FID, round number, and provider address.
- **Chunk index**: The index of the challenged chunk within the file (must match the index generated during challenge creation).
- **Raw chunk data**: The actual bytes of the challenged chunk (1KiB maximum).

- **Merkle proof path**: The sibling hashes required to reconstruct the path from the challenged leaf to the Merkle root.

Upon receiving a `MsgProveFile` transaction, validators perform a series of checks to verify the proof's validity. The verification algorithm proceeds as follows:

1. **Challenge validity check**: Verify that a challenge for this provider-file-round combination exists and has not expired. Challenges expire at the end of their proof round; any response received after the round's final block is rejected.

2. **Chunk index verification**: Confirm that the provided `chunk_index` matches the index generated during challenge creation. This prevents providers from responding with a different chunk than the one challenged.

3. **Leaf hash computation**: Compute the cryptographic leaf hash by applying Blake3 to the provided chunk data:

$$\text{leaf\_hash} = \text{Blake3}(\text{chunk\_data})$$

4. **Merkle root reconstruction**: Using the leaf hash and the provided sibling hashes, reconstruct the Merkle root by iteratively hashing with XXH3 at each level. For a leaf at index $i$ with proof path $[h_0, h_1, ..., h_{d-1}]$ where $d$ is the tree depth, the reconstruction follows:

$$\begin{gathered}\text{current} = \text{leaf\_hash} \\ \text{for } j = 0 \text{ to } d-1: \\ \text{if } (i \gg j)\&1 == 0: \\ \text{current} = \text{XXH3}(\text{current} \parallel h_j) \\ \text{else:} \\ \text{current} = \text{XXH3}(h_j \parallel \text{current})\end{gathered}$$

5. **Root comparison**: Compare the reconstructed root against the file's on-chain Merkle root stored during upload. If they match, the proof is valid.

6. **State update**: On successful verification:

   - Remove the challenge to complete it
   - Update the file's last proven record
   - Emit a `ProofValidated` event for off-chain consumers

Missed or failed proofs are accounted for after the last block of a proof round, where the `EndBlocker` scans for challenges whose deadlines have passed and triggers penalties (Section 8).

# 8 Rewards and Slashing

## 8.1 Reward Distribution

At the end of each challenge window, the `x/storage` module distributes rewards to providers based on their contributed storage over the window period. The distribution mechanism uses the storage credit pool as a global accounting primitive, eliminating the need for per-user or per-file reward tracking. The distribution algorithm proceeds as follows:

1. **Time elapsed computation**: Calculate the duration since the last distribution:

$$\Delta t = t_{\text{current}} - t_{\text{last\_distribution}}$$

   where $t$ is measured in seconds using block timestamps.

2. **Provider credit calculation**: For each provider $p$, compute the total byte-seconds contributed during the window:

$$C_p = \Delta t \cdot \text{space\_used}_p + \text{credit\_delta}_p$$

   where $\text{space\_used}_p$ is the provider's currently stored bytes (sum of all file sizes stored), and $\text{credit\_delta}_p$ is an adjustment factor described in Section 7.2.

3. **Token reward computation**: Allocate tokens from the reward pool proportionally to each provider's share of total credits:

$$R_p = T_{\text{pool}} \cdot \frac{C_p}{C_{\text{total}}}$$

   where $T_{\text{pool}}$ is the module's `uatl` balance reserved for rewards.

4. **Distribution and cleanup**: Transfer $R_p$ from the module account to each respective provider's address, reset all $\text{credit\_delta}_p$ values to zero, and burn $\sum C_p$ credits from the global storage credit pool.

This design ensures that provider income is strictly proportional to actual storage provided over time. A provider storing 1 TB for an entire window receives twice the rewards of a provider storing 500 GB for the same window, and four times the rewards of a provider storing 1 TB for only one-quarter of the window.

## 8.2 Credit Delta Mechanism

The `credit_delta` field in each provider's record serves as an adjustment mechanism to account for events that occur mid-window. Because rewards are computed at window boundaries based on current space_used values, events that change a provider's storage obligations during the window must be captured to ensure accurate compensation.

When a file is deleted mid-window, the provider's space_used decreases immediately, but the provider already stored that file for part of the window. Without adjustment, the provider would be undercompensated at the next reward distribution. The protocol handles this by incrementing credit_delta at deletion time:

$$\text{credit\_delta}_p \mathrel{+}= \text{file\_size} \times \Delta t_{\text{stored}}$$

where $\Delta t_{\text{stored}}$ is the duration from the last reward distribution (or file upload, if more recent) to the deletion time. This effectively "credits back" the storage time that would otherwise be lost due to the immediate space reduction.

Conversely, when a provider fails to prove possession of a challenged file, the protocol applies a penalty by decrementing credit_delta:

$$\text{credit\_delta}_p \text{ -= } 2 \times \text{file\_size} \times \Delta t_{\text{window}}$$

The penalty multiplier (2x) serves two purposes:

- **Disincentive**: Makes it economically irrational to skip proofs, as the penalty exceeds any potential savings from not storing the data.
- **Slashing equivalent**: Provides a continuous, non-destructive slashing mechanism that doesn't require bond confiscation for minor infractions.

Repeated missed proofs accumulate, potentially driving credit_delta negative enough that the provider receives no rewards or even "owes the protocol" though rewards cannot go negative, hence providers with negative effective credits simply receive zero until they recover.

# 9 Security Considerations

## 9.1 Data Availability and Store-on-Demand Attacks

In a naive system, a provider might attempt to store data only when it anticipates a challenge, or fetch content from another source just-in-time. Atlas Protocol mitigates such "store-on-demand" attacks by:

- ensuring challenges are distributed uniformly and unpredictably across all files and providers;
- requiring timely responses within a narrow deadline window;
- penalizing missed or invalid proofs.

Because providers do not know which file and chunk will be challenged until block inclusion, they must maintain the full dataset to avoid sustained penalties.

## 9.2 Provider Sybil and Collusion Risks

Atlas Protocol assumes a standard Sybil resistance layer via stake-weighted consensus. Provider-level Sybil attempts (e.g. splitting capacity across many identities) can be mitigated via deposit requirements and reputation over time. Collusion between providers and users, or between providers and validators, is addressed primarily by economic incentives and decentralized validator control.

# 10 Token & Governance

## 10.1 ATL Token Utility

The ATL token fulfills several roles:

- **Staking and security**: Validators bond ATL and are subject to slashing.
- **Fees**: Transactions pay gas fees in `uatl`.

- **Storage payments**: Users purchase storage subscriptions using `uatl`, which flows into the storage reward pool.
- **Governance**: ATL holders can vote on parameter changes, upgrades, and grant allocations.

### 10.2 Protocol Upgrades

There are two types of updates to Atlas Protocol - major and minor. Major updates affect the behavior of the protocol as a whole and imply: adding/removing functionality that tampers with the blockchain state, or any other changes at the architecture level (changing the algorithm for reaching consensus, signature, or number of validators). Minor updates include small non-consensus breaking changes, often to add query endpoints or deliver performance enhancements.

Proposals for major and minor protocol upgrades can be made by any member who holds the ATL tokens. To do this, she needs to send a transaction of the appropriate type, in which the update details are defined. The voting time for improvements is defined as 14 days from the date of the transaction. To make a decision, holders of 51% of the total number of tokens must vote.

### 10.3 Protocol Ownership

There is no central foundation that maintains or owns this protocol. It is an open source protocol, hence it is anticipated that developers and business entities from all around the world participate in the protocol's maintenance and continuous improvement.

### 10.4 A Note from the Author

I am tired of seeing good projects go to waste, just because they're built by a small company that depends on the tokens granted to them by them to stay afloat. With all of these new companies under dubious leadership, the adoption of decentralized technologies continues to be held back. I have seen this time and time again, and will not tolerate this for Atlas Protocol.

## 11 Conclusion

Atlas Protocol introduces a storage-focused Layer 1 that couples economic incentives and verifiable cryptography to make decentralized storage practical at scale. Byte-second accounting ties costs and rewards directly to capacity and time, while a hybrid Blake3/XXH3 Merkle scheme and structured challenge windows enable high-throughput proofs without congesting the chain. At the control plane, the protocol cleanly separates physical storage from logical organization: `x/storage` manages subscriptions, providers, files, and rewards, and `x/filetree` offers a human-centric namespace for paths, workspaces, and sharing. This modular design lets parameters, message types, and access-control models evolve through governance without breaking core security or economic invariants.

Ultimately, Atlas Protocol is intended as a piece of public infrastructure rather than a rent-seeking platform. With a minimal, utility-driven token, on-chain governance, and an open implementation, the protocol invites a broad community of operators, builders, and researchers to refine its mechanisms, extend its tooling, and deploy applications that require durable, verifiable, and censorship-resistant storage. The design sketched in this document is a starting point: its success will be measured not only by theoretical scalability, but by the diversity and resilience of the ecosystems that form around it.

**Note:** This document describes the initial implementation. Details may evolve as the protocol is tested, audited, and refined.