Irys: A Programmable Datachain for Verifiable Storage and Onchain Computation

Josh Benaron, Dan MacDonald, and Jesse Cruz Wright

Abstract

This paper presents Irys, a Layer-1 Datachain that makes data programmable by unifying high-performance storage, data availability, and an EVM-compatible execution environment so smart contracts can read and act on onchain bytes at hot-access latency. A hybrid Proof-of-Work and Stake consensus ties block production to Hard Disk Drive-bound, Verifiable Delay Function-paced efficient sampling; miners perform deterministic 200 MiB sequential reads each second and are slashable, creating continual proofs of storage with low messaging overhead and fast inclusion. Matrix packing encodes a miner's address into each 256 KiB chunk, enforcing unique replicas and closing remote-mining attack surfaces. A multi-ledger architecture supports arbitrary retention windows; data enters short-term ledgers and is promoted to permanent once ingress proofs reach threshold, decoupling fee markets and stabilizing costs. IrysVM extends the EVM with precompiles that stream chunk ranges into contracts, enabling in-protocol licensing, royalties, and verifiable AI workflows where datasets are first-class program inputs. On commodity 1 Gbps hardware, throughput scales with the number of miners and write latency is drive-limited; pricing targets at-cost HDD economics. Together these primitives provide a single chain that proves availability, ordering, and durable storage while turning data from a static payload into an executable asset.

1 Introduction

A datachain is an implementation of a fault-tolerant replicated data and execution machine. Current storage blockchains treat data as static payloads and make weak assumptions about data availability, verifiability, and usability within computation. Each node typically verifies data in isolation, without any unified mechanism to prove that the data exists, is actively stored, or can be acted upon deterministically by smart contracts. The absence of a verifiable data layer means that while storage can be decentralized, execution remains disconnected, preventing true onchain applications from operating on large-scale datasets.

Irys introduces a verifiable data architecture designed to encode proof of storage, access, and availability directly into the consensus process. Each block not only orders transactions but also cryptographically proves that every partition of data has been sampled and remains accessible. This transforms the ledger into a programmable data substrate, where both state transitions and data proofs are verified within the same trustless environment. It is anticipated that every node in the Irys network can rely on the verifiable persistence and accessibility of data recorded onchain without external assumptions or intermediaries.

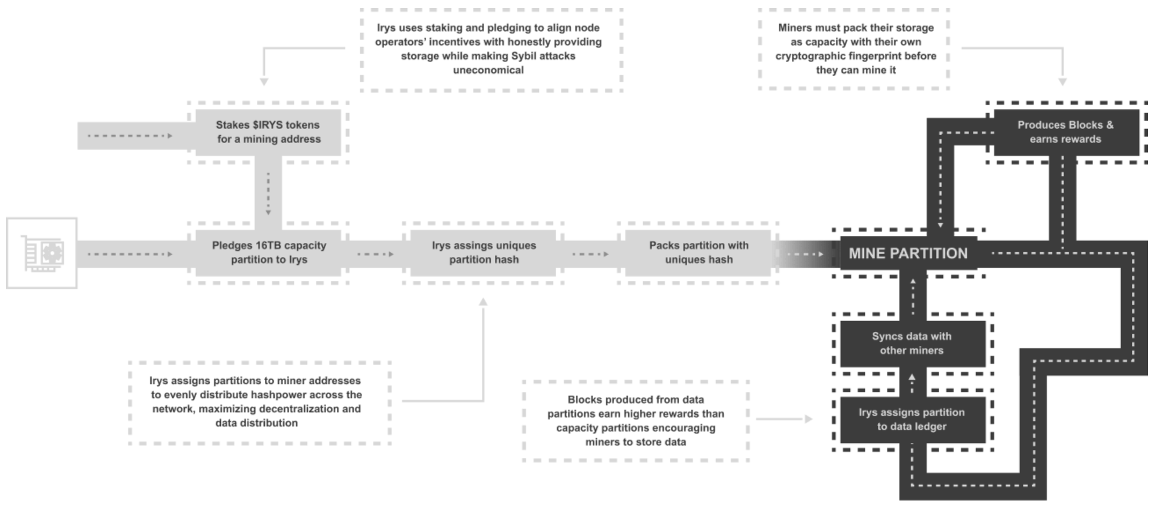
2 Outline

The remainder of this paper is organized as follows. Section 3 presents the high-level architecture and vertical integration model. Section 4 defines blocks, transaction lanes, and the multi-ledger partition lifecycle. Section 5 introduces PoW/S consensus, highlighting efficient sampling and the VDF read limiter as core subsections. Section 6 describes the data transaction lifecycle. Section 7 specifies IrysVM and Programmable Data. Section 8 details tokenomics. Section 9 formalizes the fee model for term and permanent storage. Section 10 defines epoch-boundary processing. Section 11 outlines future work on programmable-data L2s and faster blocks/finality.

3 High-level architecture

Irys is a fully integrated layer-1 protocol that unifies data storage and execution within a single architecture. By combining these traditionally separate layers, Irys enables users to store data directly onchain at a fraction of conventional costs while executing application logic natively on that same data. The design philosophy behind Irys centers on vertical integration: eliminating the fragmented tooling and multiple dependencies that developers currently face across disparate infrastructures. Instead of relying on a patchwork of external services for storage, compute, and verification, Irys provides a cohesive environment where every layer of the stack operates within one trustless, verifiable system.

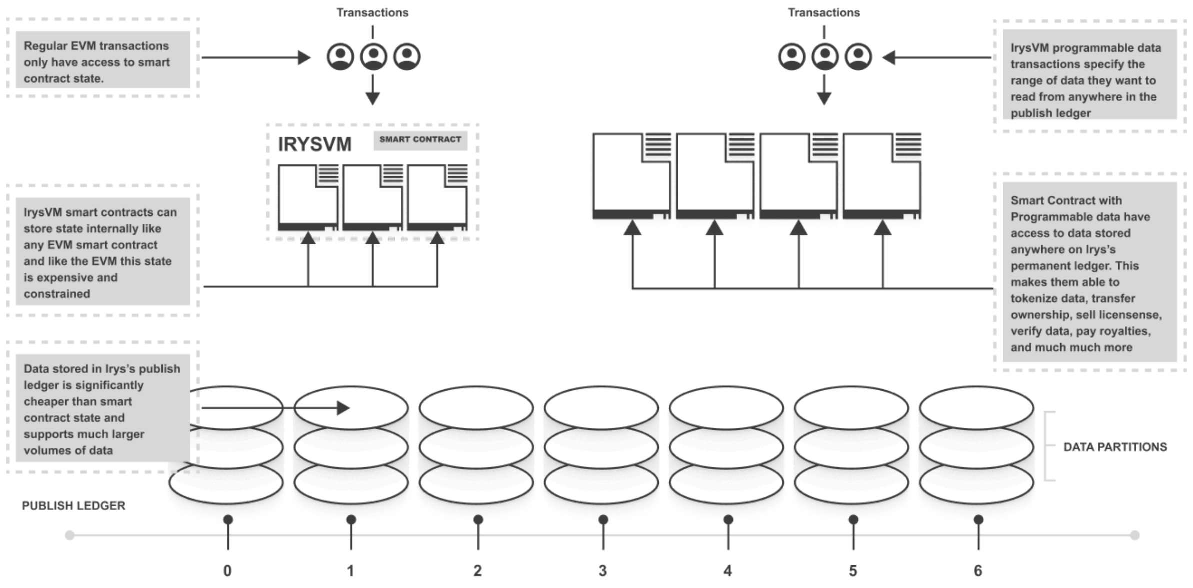
The below diagram outlines the processes within Irys that enable it to have its unique properties:



Irys uses staking and pledging to align node operators' incentives with honestly providing storage while making Sybil attacks uneconomical
Miners must pack their storage as capacity with their own cryptographic fingerprint before they can mine it
Stakes $IRYS tokens for a mining address
Produces Blocks & earns rewards
Pledges 16TB capacity partition to Irys
Irys assings uniques partition hash
Packs partition with uniques hash
MINE PARTITION
Syncs data with other miners
Irys assigns partitions to miner addresses to evenly distribute hashpower across the network, maximizing decentralization and data distribution
Blocks produced from data partitions earn higher rewards than capacity partitions encouraging miners to store data
Irys assigns partition to data ledger

Figure 1: Mining Process

Irys combines high-performance data with robust verification, enabling a native execution environment for onchain data. Traditional storage protocols struggle to balance these two aspects, making it difficult to manage execution and storage in the same chain, meaning building onchain apps on datachains has been impossible to date. Irys overcomes this tradeoff by uniting Proof of Work (PoW) with staking and slashing mechanisms. This enables Irys to efficiently handle all forms of data alongside having robust security for fully onchain applications. We call this integration of secure storage and native execution in a single protocol "Programmable Data."



Transactions
Transactions
Regular EVM transactions only have access to smart contract state.
IrysVM programmable data transactions specify the range of data they want to read from anywhere in the publish ledger
IRYSVM
SMART CONTRACT
IrysVM smart contracts can store state internally like any EVM smart contract and like the EVM this state is expensive and constrained
Smart Contract with Programmable data have access to data stored anywhere on Irys's permanent ledger. This makes them able to tokenize data, transfer ownership, sell licensense, verify data, pay royalties, and much much more
Data stored in Irys's publish ledger is significantly cheaper than smart contract state and supports much larger volumes of data
DATA PARTITIONS
PUBLISH LEDGER
0
1
2
3
4
5
6

Figure 2: Programmable Data

IrysVM enables the capability for smart contracts to directly access, compute on, and derive value from large volumes of onchain data stored within the protocol. This transforms data from a passive, static commodity into an active, composable asset class that can be queried, processed, and monetized within the execution layer itself. Through its extended EVM++ architecture and native data access primitives, IrysVM allows developers to build applications where datasets are not merely referenced but programmatically integrated into logic, enabling new classes of onchain AI, analytics, and automation that operate directly on verifiable data.

4 Blocks and transactions

There are two types of transactions:

Data transaction: a transaction that stores data on the network.

Execution transaction: a transaction that interacts with a smart contract through IrysVM. This includes programmable data transactions.

Each block contains two sets of transactions in separate block lanes. The block structure can be seen below:

Field Name	**Description**
block_hash	The block identifier
height	The block height
diff	Difficulty threshold used to produce the current block
cumulative_diff	The sum of the average number of hashes computed by the network to produce the past blocks, including this one
last_diff_timestamp	Timestamp (in milliseconds) since UNIX_EPOCH of the last difficulty adjustment
solution_hash	The solution hash for the block
previous_solution_hash	The solution hash of the previous block in the chain
last_epoch_hash	The solution hash of the last epoch block
chunk_hash	SHA-256 hash of the PoA chunk (unencoded) bytes
previous_block_hash	Previous block identifier
previous_cumulative_diff	The previous block's cumulative difficulty
poa	The recall chunk-proof

Field Name	Description
reward_address	The address that the block reward should be sent to
miner_address	The address of the block producer - used to validate the block hash/signature & the PoA chunk
signature	The block signature
timestamp	Timestamp (in milliseconds) since UNIX_EPOCH of when the block was discovered/produced
ledgers	A list of transaction ledgers, one for each active data ledger
evm_block_hash	The Ethereum Virtual Machine block hash
vdf_limiter_info	Information about the Verifiable Delay Function limiter

Irys uses block lanes to ensure that users can always submit a data transaction, even when the execution lane is congested. This allows data transactions to have stable pricing.

4.1 Partition Lifecycle (Capacity & Data, Multi Ledger)

Irys introduces a multi-ledger system where data can seamlessly transition between different terms. This effectively enables users to store data for any period of time (e.g., 1 day, 1 week, 1 year, or forever). There are also inbuilt mechanisms for "promoting" data to longer-term ledgers––e.g,. when a user requests to store data on the permanent ledger, it enters the 5-day ledger and transitions to the permanent ledger once it's been verified by the network.

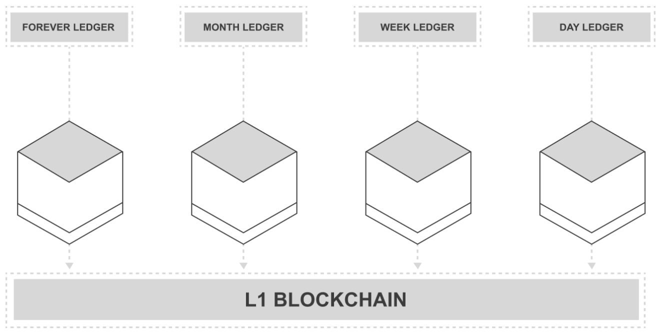


Figure 3: Multi-Ledger System

A ledger is a collection of data with a shared property (typically duration), i.e., all data in ledger 0 is stored for 5 days.

4.2 Partition Lifecycle

Ledgers on Irys are split up into 16TB partitions. This allows miners to affordably use HDDs and not be outcompeted by SSDs.

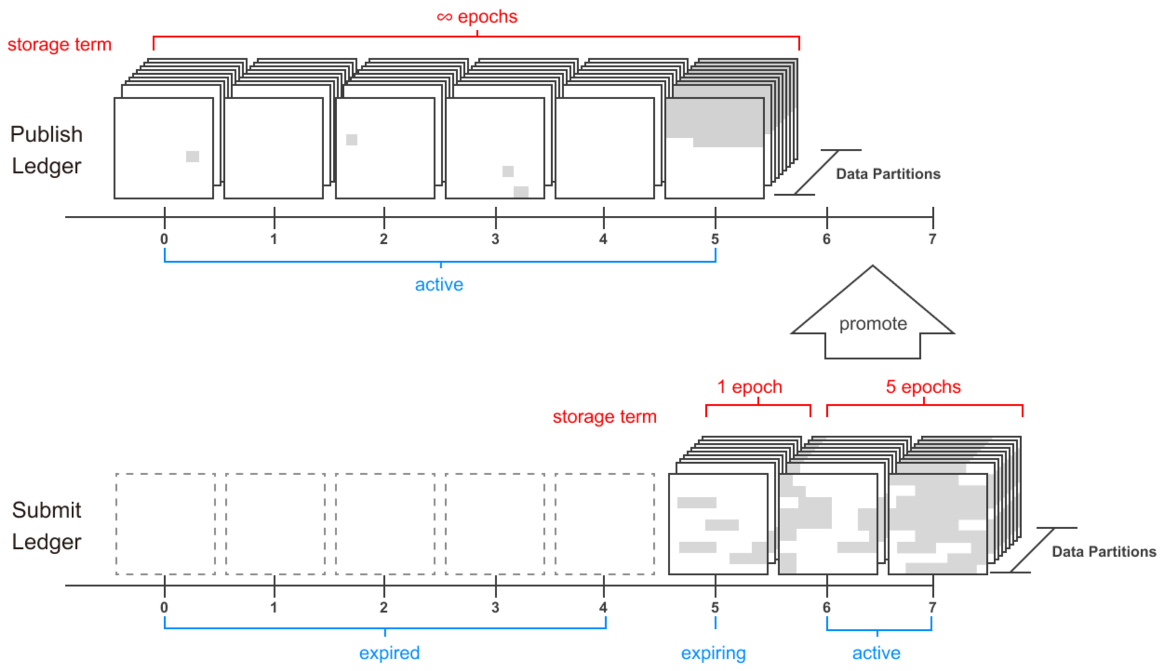


Figure 4: Ledger Promotion

Irys measures the amount of data uploaded and then uses it to project the number of partitions needed on standby. These standby partitions do not contain any data (yet) and are referred to as Capacity Partitions.

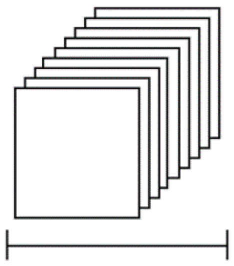


Figure 5: Capacity Partitions

Step 1: Partition Pledging

Miners post a pledge transaction to the network indicating their willingness to bring a new partition online. The protocol then randomly assigns unclaimed capacity partitions to the pledged miners; any miners who don't receive capacity partition assignments have their pledges refunded.

Step 2: Partition Packing

Once assigned a Capacity Partition, the miner packs it using Irys' matrix packing scheme. This process fully encodes the miner's fingerprint into the partition.

Step 3: Partition Mining

Once a capacity partition is packed, the protocol assigns a random 200MB sequential read to the miner every second. The miner takes these 200MB reads, splits them into 800 256KiB chunks, and proceeds to hash each chunk, looking for a mining solution. If the miner is lucky enough to find a mining solution, it wins the right to produce a new block and announce it to the network, thus earning rewards.

Step 4: Ledger Assignment

When it comes time to add more storage capacity to one of Irys' data ledgers, the protocol will randomly select a Capacity Partition that is actively being mined. The randomness is weighted towards partitions used to mine blocks, rewarding miners who have been participating longer and have been effective in mining.

Being assigned to a data ledger and mining it has higher rewards than mining empty capacity partitions, so miners are incentivized to demonstrate their capability by mining capacity efficiently.

Step 5: Partition Departure

There are two ways partitions leave the network: orderly departures and disorderly departures.

4.2.1 Orderly Departure

1. The miner posts a transaction to un-pledge their partition and recover the commitment they staked initially.
2. A timeout period begins when the protocol assigns another Capacity Partition to synchronize the data being taken offline.
3. Once the timeout has passed, the departing miner can recover their staked commitment and remove their partition.
4. If the miner goes offline before the timeout has passed, they risk losing their pledged bond.

4.2.2 Disorderly Departure

1. A miner engages in adversarial behavior toward the network (double signing blocks, for example).
2. The protocol takes their staked commitment.
3. The protocol assigns a new Capacity Partition to fill gaps left by the adversarial miner.

This is an undesirable departure as the network will have to assign new capacity partitions quickly. Because of this, Irys requires miners to make a significant upfront investment by staking tokens to their mining address in order to participate in the protocol and earn rewards.

4.3 Matrix packing

Packing describes the process of miners encoding data with their staking address. This proves they're storing a unique replica of the data, as opposed to mining off a single remote copy (a common attack vector on datachains).

Irys introduces Matrix Packing, which uses a VDF to encode the miner's address into each 256KiB chunk they store. This provides sufficient computational cost for each bit of data, making any adversarial mining pattern unprofitable. The process is outlined in detail below:

4.3.1 Phase 1 - Sequential Hashing

1. **mining_address**, **partition_hash**, and **chunk_offset** are SHA-256 hashed into **seed_hash**.
2. **seed_hash** is SHA-256'd to form the initial **segment**.
3. The **segment** bytes are appended at the start of the empty chunk.
4. The **segment** is also used as input for the next SHA-256.
5. Resulting **segment** is appended following the previous segment's bytes.
6. Steps **4** and **5** are repeated using the previous segment until the empty chunk is filled with segments.

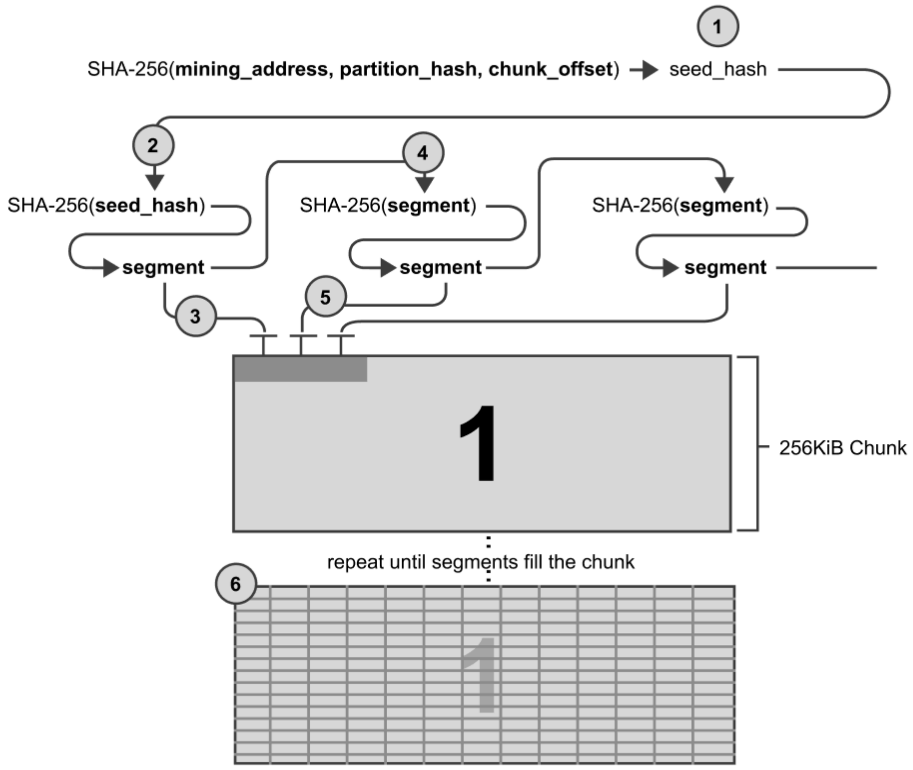


Figure 6: Sequential Hashing

Sequential hashing ensures single-core chunk packing since each segment depends on the one before it, establishing a minimum packing time. This method allows mining of data-less "capacity" chunks that are verifiable by other miners deterministically (by recreating the packing locally and comparing it to the provided packed chunk).

With 32-byte segments and a 256KiB chunk, one pass of sequential hashing requires 8,192 hashes. The challenge is that a Ryzen 5900X can compute ~15 million SHA-256 hashes per second, making this amount of packing delay take a tiny fraction of a second. To deter adversarial miners from packing "on demand" we'll have to increase the packing length, something on the order of ~1500ms.

4.3.2 Phase 1 - Sequential Hashing + Parametrized Packing Time

To lengthen chunk packing time, sequential hashing could loop from the chunk's last to first segment, overwriting it. This process could repeat for multiple passes until reaching a desired packing duration.

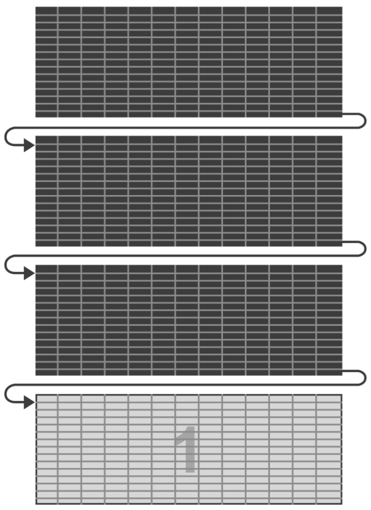
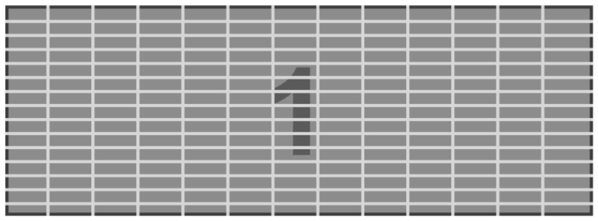


Figure 7: Parametrized Packing Time

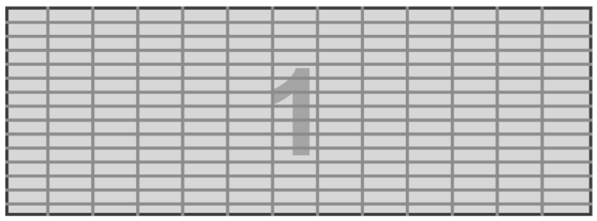
Repeated sequential hashing of the chunk creates layers of segments, with the final layer stored in the chunk. This process succeeds in realizing a specific packing time but it also exposes a potential vulnerability to adversarial miners.

4.3.3 Fast Packing Vulnerability

Miner then stores only the first segment of the resulting packing layer and discards the rest.



When they need to pack a chunk, they can use these segments to reproduce the final packing layer without computing all previous packing layers (in parallel).



If a miner wanted to stake multiple mining addresses for replicas of the same partition they could use this strategy to avoid storing a unique partition for each staked address. Instead they could store an unpacked partition and quickly pack chunks "on demand" for each mining address they staked to that partition.

This is an example of using compute instead of storage to satisfy the consensus rules of the protocol and is considered a degenerate mining strategy (not accomplishing the protocol's goal of provably replicating partition data).

4.3.4 Adversarial Mining

Why would a miner want to use "on demand" packing?

Miners are economically incentivized to use the most profitable mining strategy. For miners with access to cheap energy and pools of compute (GPU or CPU) it may be more profitable to employ a computational strategy to mining than to pay the storage costs of a storage-based strategy. They'd employ this degenerate strategy (in that it does not reinforce or support the protocol's utility) — a miner will maintain a single unpacked replica of a partition.

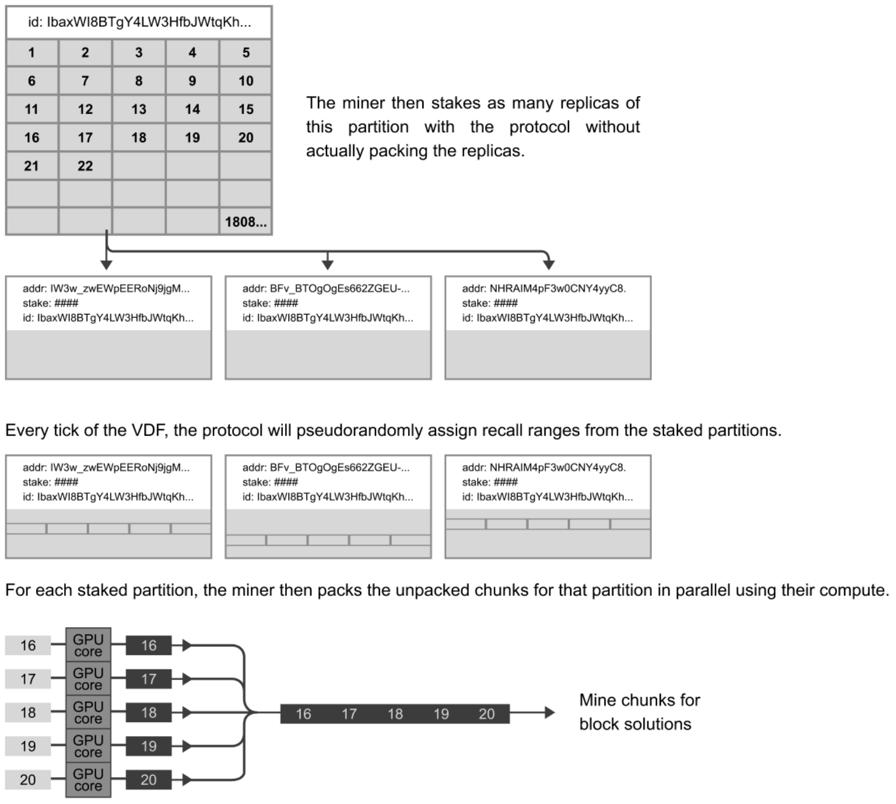

id: IbaxWI8BTgY4LW3HfbJWtqKh...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 1808...
The miner then stakes as many replicas of this partition with the protocol without actually packing the replicas.
addr: IW3w_zwEWpEERoNj9jgM...
stake: ####
id: IbaxWI8BTgY4LW3HfbJWtqKh...
addr: BFv_BTOgOgEs662ZGEU-...
stake: ####
id: IbaxWI8BTgY4LW3HfbJWtqKh...
addr: NHRAIM4pF3w0CNY4yyC8.
stake: ####
id: IbaxWI8BTgY4LW3HfbJWtqKh...
Every tick of the VDF, the protocol will pseudorandomly assign recall ranges from the staked partitions.
For each staked partition, the miner then packs the unpacked chunks for that partition in parallel using their compute.
16 17 18 19 20
GPU core
16 17 18 19 20
Mine chunks for block solutions

Figure 8: "On Demand Packing" Example (Adversarial)

If this process is fast enough to win block rewards, the miner may choose it instead of packing all the chunks of a partition replica like the protocol expects.

4.3.5 Phase 2 - 2D Matrix Packing

To stop "on-demand" packing, the protocol requires miners to store all chunk segments of the previous layer when computing a new one. This makes computing the final layer possible only by recomputing all prior layers through two-dimensional sequential packing: breadth and depth. Each layer depends on the previous one, starting with Phase 1 - Sequential Packing.

1. a) The last segment of the previous layer becomes an input to the first sequential hash of the new layer.
1. b) The segment on the previous layer at the same location of the segment we are computing becomes input entropy to the current segment.
2. The segment produced by hashing the previous segment and the entropy segment is appended to the new layer.
3. a) As in step 1a, the segment computed in step 2 is used as input to the next sequential SHA-256 hash.
3. b) As in step 1b, the segment from the previous packing layer at the same location as the segment currently being computed is used as entropy.
4. The resulting segment is then appended to the chunk after the previous segment.
5. Repeat until the required number of packing layers have been achieved to satisfy the packing duration requirement.

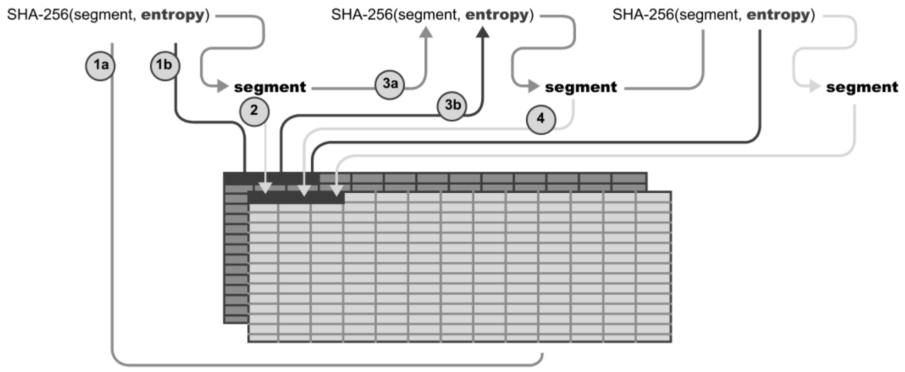

SHA-256(segment, **entropy**)
SHA-256(segment, **entropy**)
SHA-256(segment, **entropy**)
1a
1b
segment
2
3a
3b
segment
4
segment

Figure 9: Sequential Packing Process

The result of packing in two dimensions this way is that every segment of a new packing layer requires the segments of the packing layer immediately beneath it. There is no way to compute the final packing layer without first computing every packing layer beneath it. This ensures that adversarial miners wishing to pack "on demand" will have to pack for the full packing duration every time they pack a chunk.

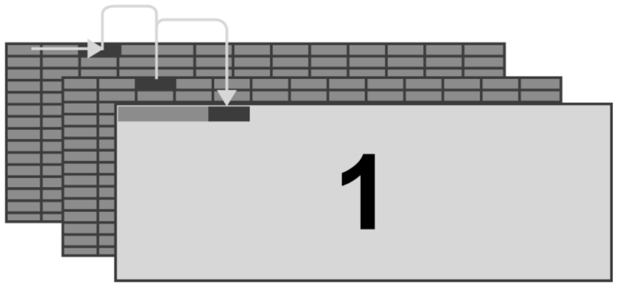


Figure 10: 2D Matrix Packing

5 PoW/S consensus

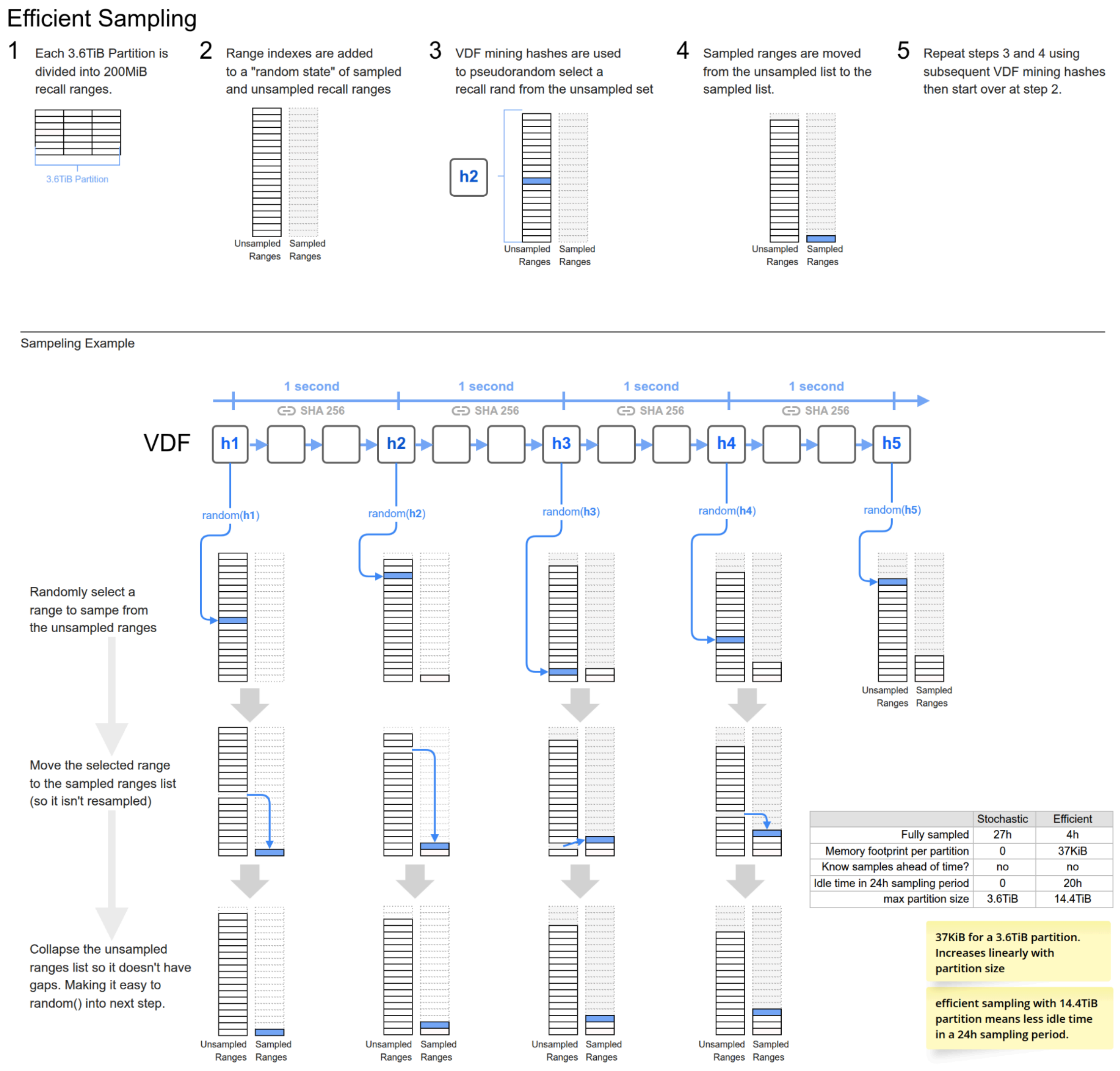
Irys introduces a hybrid Proof-of-Work-Stake consensus algorithm. This was done to achieve the following set of requirements:

- Sustainable economics for permanent data
- Minimize infflationary pressure long-term
- Create accountability for miners such that they can be punished for malicious behavior
- Must reliably sample all data stored once per day

A hybrid consensus enables us to adopt PoW economics with security mechanisms like slashing to provide maximally reliable onchain data with performant execution.

5.1 Consensus algorithm

Irys uses a process known as efficient sampling, where the chain guarantees every partition is sampled entirely once per day.


Efficient Sampling

1 Each 3.6TiB Partition is divided into 200MiB recall ranges.

2 Range indexes are added to a "random state" of sampled and unsampled recall ranges

3 VDF mining hashes are used to pseudorandom select a recall rand from the unsampled set

4 Sampled ranges are moved from the unsampled list to the sampled list.

5 Repeat steps 3 and 4 using subsequent VDF mining hashes then start over at step 2.

Sampeling Example

Randomly select a range to sampe from the unsampled ranges

Move the selected range to the sampled ranges list (so it isn't resampled)

Collapse the unsampled ranges list so it doesn't have gaps. Making it easy to random() into next step.

	Stochastic	Efficient
Fully sampled	27h	4h
Memory footprint per partition	0	37KiB
Know samples ahead of time?	no	no
Idle time in 24h sampling period	0	20h
max partition size	3.6TiB	14.4TiB

37KiB for a 3.6TiB partition. Increases linearly with partition size

efficient sampling with 14.4TiB partition means less idle time in a 24h sampling period.

Figure 11: Efficient Recall Range Sampling

The algorithm can be summarized as so:

1. There is a VDF running “ticks” every 1 second, generating a seed
2. For every partition a miner stores, use the seed in a deterministic random function to generate the start position of a range
3. Read 200MiB worth of chunks (800)
4. For each chunk, calculate a solution
    a. Convert into a number
5. If the solution is greater than the current network difficulty, then publish the block. If it’s below the difficulty then go back to step 1

The idea here being that miners are continuously sampling data to participate in consensus creating a strong guarantee around data being sampled. If miners lose data then they can be challenged and ultimately slashed if proven malicious.

5.2 VDF to synchronize read speed

One key requirement for Irys was to force people to use HDDs, or in other words, don't allow people to outcompete by using SSDs. A VDF is used to achieve this by only allowing 200MB to be read every second per partition a miner stores, essentially limiting the read speed of each partition at 200MB (approximately the max read speed of an HDD).

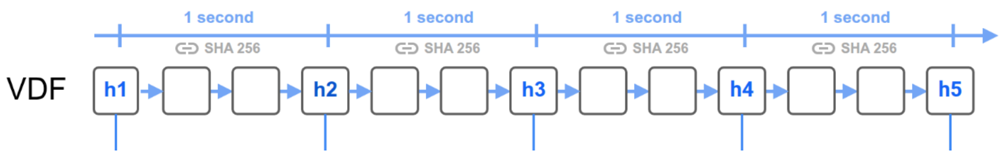


Figure 12: Read Speed per Partition

If the difficulty is greater than the network difficulty and the miner has sufficient stake, the block is accepted.

6 Data transaction lifecycle

Each data transaction specifies a ledger the data should be stored in, and each ledger represents a duration the data should be stored for; for example, a transaction could represent "I want to store this 1 GB of data in ledger 0, which stores data for 2 weeks". Once a user has posted a transaction and all ingress proofs have been verified, the data is XOR'd into a partition where it can be used in consensus proofs.

The only special case is when the transaction specifies the permanent ledger. In this case, the transaction enters the submit ledger and then migrates to the permanent ledger once 10 miners have proved they're storing the data.

6.1 Ingress proofs

An ingress proof is a Merkle root that can only be generated by accessing the chunks of a transaction's data. Ingress proofs are used to prove a miner has some data. This is a required step before miners can store data, providing evidence that they store a unique replica of the data.

The data chunks are hashed together with the miner's address to create a proof unique to that mining address. This process makes the proofs easy to generate and validate by someone with the data. To prevent false claims, these proofs must also be signed by the miner's private key. Without this signature, any miner with access to the data could generate proofs for another mining address, falsely claiming that they had downloaded the data.

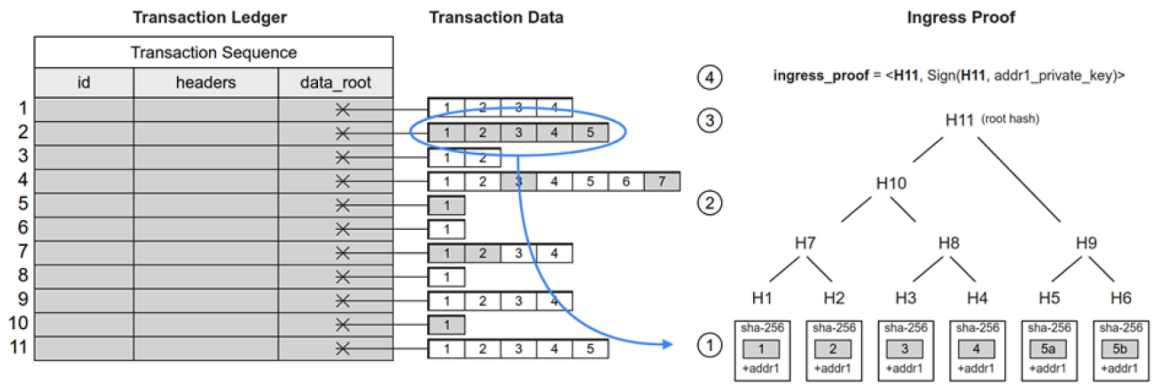


Figure 13: Ingress Proofs

Identify recently added transactions in the submit ledger that are waiting to be promoted to the publish ledger. Obtain the data associated with these transactions, either directly from a user or through gossip with other miners.

Once all the data chunks are collected, create an ingress proof as follows:

1. Split the data into chunks and hash each chunk together with the miner's address. (note: chunk 5 is split into 5a and 5b to build a more balanced Merkle tree.)
2. Continue hashing these hashes to build the branches of a Merkle tree.
3. Compute the root hash of the Merkle tree.
4. Sign the root hash with the private key associated with the mining address used in step 1.

7 IrysVM and programmable data

Irys introduces IrysVM, an EVM++ implementation, which adds new opcodes. The main upgrade made is to enable programmable data: the use of stored data within smart contracts. Essentially this means you could store 1EB of data and use portions of the data within a secure environment.

To fully implement programmable data, we must integrate the feature across multiple layers in the tech stack. Starting with transactions that allow the caller to specify the range of chunks they wish to access with their PD SmartContract call. Next, gossip the transaction and request the range of chunks to verify their availability. Finally, include the transaction in a block and execute the state transition.

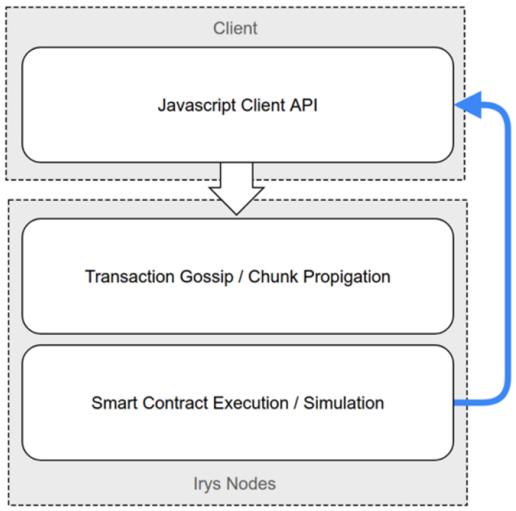

Client
Javascript Client API
Transaction Gossip / Chunk Propigation
Smart Contract Execution / Simulation
Irys Nodes

Figure 14: Execution of State Transaction

7.1 Posting Transactions

To post a PD transaction the transaction must specify the range of chunks it wishes to have available during the SmartContract invocation. To maintain compatibility with EVM toolchains this is done by including the range of chunks to reference as elements using EIP-2930 access lists, where the address is the address of the PD precompile, and the data is one or more range specifications.

7.2 Range Specification

Chunk range specification follows a simple format.

```
<partition_index>:<offset>:<chunk_count>
```

partition_index: The partition index in the publish ledger of the partition containing the first chunk.
offset: The chunk offset in the partition to begin reading chunks
chunk_count: The number of sequential chunks to read after offset
partition_index: 26 bytes - 0 to a lot
offset: 4 bytes - 0 to ~80,000 (num chunks in a partition)
chunk_count: 2 bytes - 0 to 65,535 (PD is constrained to about 7,000 chunks per block, so this is safe by an order of magnitude)
Total Binary Bytes to represent a range: 26+4+2= 32 bytes
Constraining the range to 32 bytes makes it easily storable in the EVM smart contract state.

Note: Ranges are always 32 bytes, and the values are unpacked by slicing the bytes at the correct offsets for each value.

7.3 Irys Client SDK

The default way of referencing data on Irys is by using the transaction ID, putting the burden on the developer to locate the partition and chunk offset of the data they are interested in for PD is high friction and requires a lot of knowledge of the internal data model of the chain.

To simplify the process of posting PD transactions, the client SDK implements some utility methods to abstract away the complexity of specifying chunk ranges.

```
const chunkRange = getTxChunkRange(txid);
```

The implementation of this function is provided by the gateway the client is using.

1. It looks up the bundle_txid if the txid is actually a DataItem ID
2. It verifies the txid is in the Publish Ledger
3. It looks up the chunk_offset in the Publish Ledger
4. It looks up the partition_index in the Publish ledger.
5. It uses 3 & 4 to compute the partition_offset.
6. It uses the Publish Ledger txid to look up the size of the transaction data.
7. It computes the number of chunks required to read the transaction.
8. It builds a Range Specification and returns it to the caller.

7.4 Estimating Transaction Price

The price of PD is determined by the same transaction simulation that estimates gas prices for posting transactions to the EVM. Because the chunk range is included in the call data of the transaction, when the simulation is run the simulation mechanism includes the pricing for the number of chunks needed to be retrieved along with the computational budget of the transaction.

In order to properly determine the cost of Programmable Data in Irys, key parameters that impact network performance must be defined:

Propagation Delay (D): The maximum time allowed for data to travel across the network. Choosing an upper bound allows us to extrapolate blocktimes and probability of forks in the network.

Minimum Connection Speed: With a propagation delay established, we can identify a minimum connection speed to describe the expected data throughput of the network.

Peer Connections in a Gossip Network: Each peer connects to a limited number of other peers (e.g. 20). The number of these connections allows us to maximize utilization of an individual peer's network connection while minimizing the number of "hops" between any two peers on the network.

7.4.1 Gossip Networks

Calculating how long it takes to propagate data across the network requires some understanding of the underlying gossip protocol. Gossip protocols exist because they scale better and have higher throughput than connecting every peer with every other peer.

In a network of 1000 peers, each peer maintains 999 connections, consuming memory and CPU. When a peer produces a block, it sends a copy to each peer. With a 1 Gbps connection (125 MBps) and a 500 KB block, the peer can broadcast to 250 peers/second. It would take 4 seconds to reach all peers, fully using the connection.

In the same 1000-peer network, each peer connects to only 20 "closest" peers based on ping time. A block is broadcast to 20 peers, who then forward it to their 20 closest peers. With a 1 Gbps connection, broadcasting to 20 peers happens in a small fraction of a second. The block reaches the entire network in a few hops, modeled by:

$$\log_{\text{number\_of\_peer\_connections}}(\text{total\_network\_peers})$$

$$\log_{20}(1{,}000) = \mathbf{2.31 hops}$$

$$\log_{20}(10{,}000) = \mathbf{3.07 hops}$$

As blocks are broadcasted across peers in a few hops, the propagation delay D directly impacts the time to reach consensus, making it a critical factor for ensuring timely block propagation.

7.4.2 Propagation Delay

Propagation Delay (D) is the time for data to travel between peers in the Irys network. In a gossip network, data moves through multiple hops. Assuming an average link speed of 60ms per hop, data can reach 10,000 peers in three hops, giving a D of about 180ms.

Forks may occur when a new block is produced during another's propagation. To maintain a 1% fork chance with a D of 180ms, block times should be around 18 seconds. For the testnet, Irys will implement 30-second blocks to keep the fork chance at 1% with a D of 300ms.

Network consensus requires at least 51% of nodes to confirm a block within the Propagation Delay, allowing for a minority of nodes with slower links.

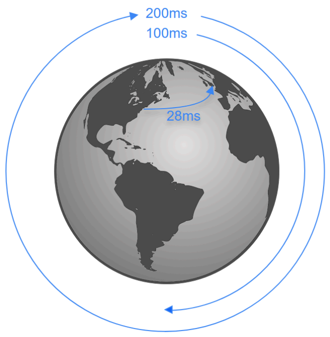


Figure 15: Propagation Delay

This is important because network propagation delay directly impacts core performance and security parameters. Shorter propagation reduces the likelihood of forks by minimizing the window for simultaneous block production, ensuring a single canonical chain forms more reliably. It also defines the lower bound for block time; faster propagation allows shorter intervals between blocks, improving responsiveness and transaction throughput. Additionally, understanding and optimizing propagation delay determines the network's scalability limits, dictating how many nodes can participate in consensus while maintaining efficient data flow and synchronized state across the system.

7.4.3 Modeling Programmable Data

On a 1Gbps connection, a peer can transfer 500 data chunks (256KB) per second, fully utilizing the connection. Since the protocol handles other tasks like chunk ingress and block gossip, we'll assume 50% of bandwidth is available for Programmable Data chunk propagation.

To determine the cost of transferred Programmable Data, consider the following:

Network Capacity:

- Assumed connection: 1Gbps
- Maximum transfer rate: 500 chunks (256KB each) per second
- Available bandwidth for Programmable Data: 50%
- Effective transfer rate: 250 chunks per second

Block Capacity:

- Block time: 30 seconds
- Maximum chunks per block: 7,500 (250 chunks/second * 30 seconds)

Cost Analysis:

- Based on 10Gbps connection at $300/month (retail pricing)
- Transfer cost: $0.10 per 1000MB
- Cost per block (7,500 chunks = 1,875MB): $0.20
- Cost per chunk: $0.00002667 (negligible for individual pricing)

7.4.4 Pricing Programmable Data

Beyond raw bandwidth, unpacking and deserializing data chunks for use within IrysVM introduces computational overhead that must be priced to prevent spam. To account for this, Irys implements a minimum base_fee for Programmable Data transactions. During testnet, the base_fee for 1 MB of Programmable Data is set at $0.01, which also serves as the network-wide minimum cost for such transactions.

To balance throughput and demand, Irys employs a dynamic congestion pricing model where the base_fee adjusts by ±12.5% per block, similar to Ethereum's gas_fee mechanism. If more than 50% of the 7,500 available Programmable Data chunks in a block are used, the base_fee increases linearly up to +12.5%. Conversely, if a block contains no Programmable Data chunks, the base_fee decreases by as much as −12.5%. While there is no upper limit on fee growth during congestion, the system enforces a hard floor of $0.01 to maintain network integrity.

When the base_fee is rising but Programmable Data demand still exceeds available capacity, users can attach priority fees (additional payments above the base rate) to incentivize miners or validators to include their transactions in the next block. This ensures market-driven inclusion for high-value or time-sensitive workloads while preserving predictable baseline costs for general use.

All base_fees from Programmable Data transactions are deposited directly into the network treasury, effectively removing them from circulation and creating deflationary pressure on the native $IRYS token. In contrast, priority fees—the portion users pay above the base and compute fees—are awarded to the block producer. This dual incentive model aligns miner profitability with network efficiency, while progressively strengthening token value through protocol-level burns and treasury accumulation.

7.5 Gossip & Mempool

An important constraint on PD transactions is the maximum capacity for propagating PD chunks between a majority of nodes on the network between blocks.

7.6 Transmission

When a node receives a PD transaction, it broadcasts it to its peers, indicating whether it has the chunks. Receiving peers may request the chunks in their response. The broadcasting node tracks which peers need the chunks and sends them after receiving them. Receiving peers also broadcast the PD transaction, marking peers that have already received it and noting which peers have the chunks. When a peer receives the chunks, it sends them to peers lacking them.

If a peer hasn't received the chunks after some time, it may retrieve them from assigned storage partitions by inspecting the ledger, locating partition owners, and requesting the chunks from a randomly chosen partition.

7.7 Validation

Transaction validation follows the same static validation as other transactions.

Chunk validation is a little more complicated. PD Transaction chunks can be retrieved in a number of ways.

1. The node has the cached unpacked chunks locally.
2. The node has the packed chunks in a partition they mine.
3. The node receives unpacked chunks from a peer.
4. A node requests packed/unpacked chunks from a peer.

Cached Unpacked Chunks: In this case, the node has already validated the chunks with their Merkle roots and can be confident the data in them is correct and ready to be exposed to the VM for PD execution.

Local Packed Chunks: The node happens to mine the partition that contains the PD chunks requested by the transaction, but they are packed.

The Node:

1. Creates the entropy for the chunk range
2. Unpacks the chunks using the computed entropy
3. Builds a merkle-root out of the unpacked chunks
4. Looks up the transaction that posted the chunks from its block index
5. Compares the computed merkle-root with the one in the transaction.
6. If valid, the node posts the unpacked chunks to any peers marked as not having them.

Receives Unpacked Chunks: In this case the node is being sent unpacked chunks by one of their peers. While they do not have to unpack the chunks they do need to verify the chunks contain the correct data or risk proposing an invalid block.

The Node:

Follow steps 4-6 from the Local Packed Chunks path.

Requests Chunks: As a failsafe, if the node is not receiving any of the chunks within time D, where D is the propagation delay of the network (Assume D = 200ms for testnet).

The Node:

1. Looks up the partitions responsible for storing the chunks.
2. Picks a partition at random to request the chunks.
    a. If the partition provides packed chunks, unpack them.
3. Follow steps 4-6 from the Local Packed Chunks path.

7.8 Block Production

After a mining node receives a PD transaction and its chunks and validates them, it can include the transaction in a block. To minimize the chance of producing a block with a PD transaction that the majority haven't received the chunks for, the miner may wait for the propagation delay D before including it. This allows most of the network to retrieve the chunks and validate the PD transaction when it's included in a block.

7.9 Smart Contract Execution

Executing a PD smart contract interaction requires further exploration. There are a few possible approaches, but they will require exploration of the code to evaluate their feasibility.

7.9.1 Exposing Chunk Data

PD transactions include an instruction to a precompiled "system" contract which takes the Range specification as an input. This precompile will bring the chunk data into scope. There are at least two possible approaches.

Return Value: The foreign call to the precompiled "system" contract could return the chunks specified by the range as a buffer.

Global State: Once the precompiled "system" contract has been invoked the chunks become accessible via a global that is exposed to all subsequent instructions in the PD transaction.

7.9.2 Calculating Compute Units

Because the execution of a particular smart contract function may take one code path or another depending on the data read from the chunks, calculating compute units (CUs) can be problematic. There are a few possible approaches

Simulate With Chunks: The only way to deterministically simulate the CUs required to complete the execution of the instruction is to have the unpacked chunks available during the simulation. This would require the simulating node to retrieve the unpacked chunks during the simulation request.

Simulate Compute Upper Bound: In this case, the simulation would evaluate all code paths and return the cost of the most compute-intensive code path. This way the user always pays enough gas for any possible computational resources.

7.10 Programmable data roadmap

Blob Data - MVP: The first version of PD transactions will expose chunks as buffers or blob data to the contract and leave the interpretation of these bytes up to the caller. This will allow PD chunks to have any structure or format the caller can imagine.

Bundle Format v.1 - Bundles: Once the blob chunks are working, the next layer of functionality will be a DX upgrade that allows PD contracts to load a bundle and data items from the chunks. The IrysVM will parse the chunks in the range specification as a v1 bundle format.

Bundle Format v.2 - DataItems: Once the v2 (merkelized) bundle format exists, a DX upgrade will allow parsing of specific data items from a larger bundle or retrieve smaller nested bundles (or their data items) by loading only the chunks that store the specific data items the caller is interested in.

Programmable data L2s: In the future, we expect users will develop programmable data L2s that expand the compute capacity beyond a single-state machine. The end goal here is a shared dataset with the ability for anyone to spin up L2s to tap into data, compute, and liquidity resources.

8 Tokenomics

The Irys native token, IRYS, underpins the economic and security architecture of the Irys protocol. Its design integrates five foundational components that collectively align incentives across users, miners, and the broader ecosystem. Its role is to align incentives across all participants, from early adopters and community members to institutional partners and long-term investors.

8.1 Token distribution

The initial supply of IRYS is 10,000,000,000 tokens. At the Token Generation Event (TGE), the circulating supply will be approximately 2,000,000,000 tokens, representing 20.0% of the maximum supply.

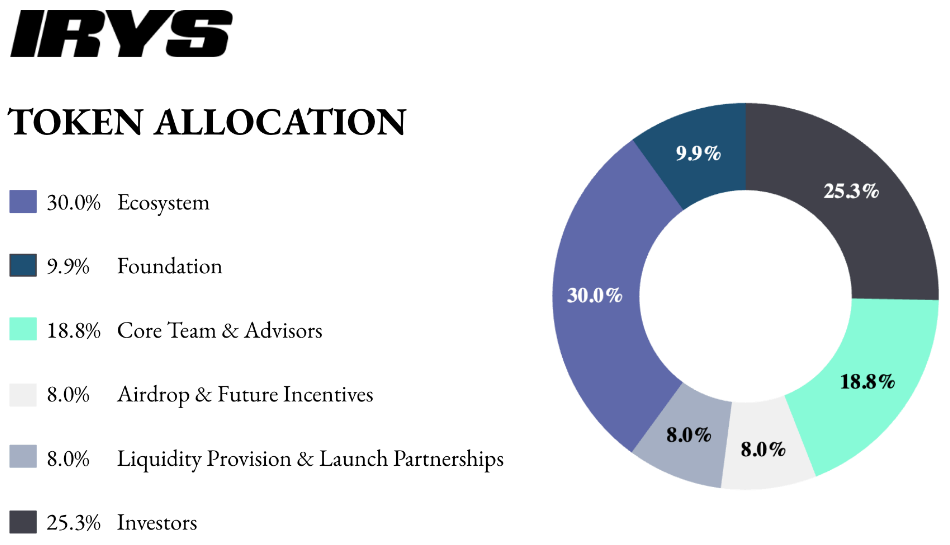


Figure 16: Irys Token Allocation.

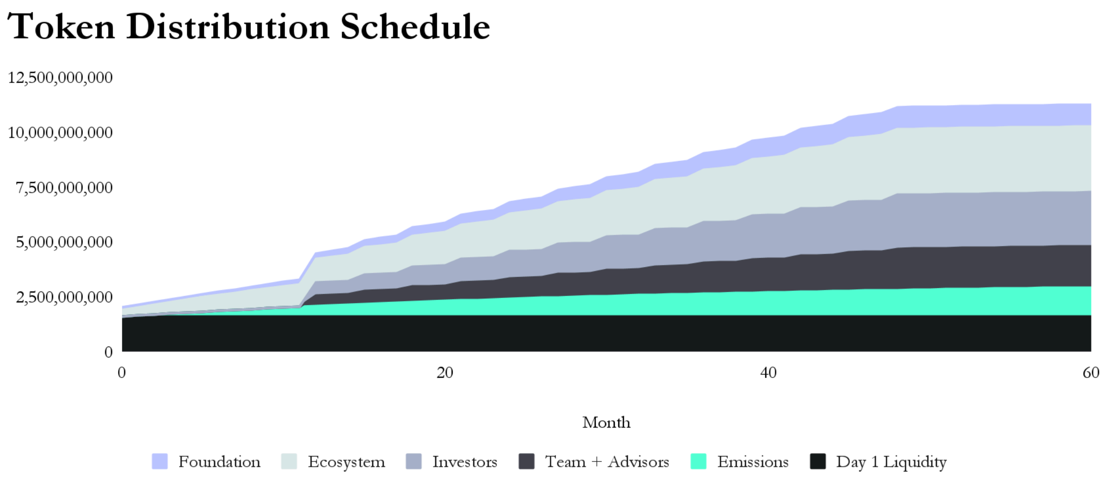


Figure 17: Token Distribution Schedule

Ecosystem (30%): This allocation will be used for various Irys initiatives, including retaining and incentivizing novel decentralized applications, as well as various cross-chain initiatives, partnerships, and more. The allocation will be held by a secure, multisig wallet in collaboration with the top custodians in the space.

Foundation (9.9%): This is used to further initiatives that serve to widen the reach of Irys, reducing crypto's reliance on centralized data storage and fragmented developer experiences. This $IRYS allocation will be used to fund further development, risk assessments, audits and more.

Core Team and Advisors (18.8%): This portion of the $IRYS allocation (18.8%) represents the distribution to the core contributors to the Irys Layer 1 Protocol, as well as advisors who have worked to bring the L1 protocol to market. All core contributors are locked on a 1-year cliff, with 3-year linear monthly vesting thereafter. No core contributor tokens are unlocked prior to the 1-year cliff, and no staking rewards will be unlocked in advance of vesting schedules.

Airdrop and Future Incentives (8%): This allocation is designed to recognize meaningful contributions that helped bring Irys to market. A portion of this allocation is reserved for future incentive programs designed to attract aligned contributors, developers, and users over time. This ensures that Irys continues to grow with the right people, not just the loudest.

Liquidity, Market Stability and Launch Partnerships (8%): This allocation will be deployed to market makers and launch partners to establish robust liquidity profiles from day one. Our goal is to create an efficient trading environment that supports organic volume growth and prevents manipulation. By anchoring this allocation to strategic partners and a mix of CEX and DEX infrastructure, we ensure Irys is accessible, liquid, and trade-ready, without compromising long-term sustainability.

Investors (25.3%): This represents token rights obtained by investors backing the Irys protocol's development, to bootstrap the protocol. All investors are locked on a 1-year cliff, with 3-year linear monthly vesting thereafter. No investor tokens are unlocked prior to the 1-year cliff, and no staking rewards will be unlocked in advance of vesting schedules.

8.2 Token utility

The token will have all of the following utilities at launch:

Native Asset: $IRYS serves as the native utility token of the Irys network, functioning as the primary medium for payments, collateral, and settlement across all protocol-level operations.

Fees: All network actions—including data uploads, programmable data executions, and contract interactions—are denominated in $IRYS. Unlike traditional datachains, Irys employs a USD-pegged pricing model for both temporary and permanent data storage, recalibrated annually to reflect real-world storage costs. This ensures stable, predictable economics for users and developers independent of token volatility.

Security: Token emissions and rewards are allocated to validators and miners who maintain consensus integrity and data availability. This incentivization structure provides economic protection against spam, denial-of-service attacks, and other adversarial behaviors, ensuring that network reliability scales with participation.

Endowment: A portion of $IRYS is directed into a protocol-level endowment that underwrites miners' long-term storage commitments. The endowment functions as a self-sustaining reserve, covering future storage liabilities and reinforcing the permanence guarantees of the network's data layer.

Staking: Miners and validators must stake $IRYS tokens as bonded collateral to participate in consensus and data verification. This mechanism enforces accountability—malicious or negligent behavior results in slashing—and ensures that economic security is directly tied to the integrity of the network. Delegation mechanisms allow token holders to contribute to network security indirectly, earning rewards proportional to their delegated stake.

8.3 Token emissions

Each block, new IRYS tokens are minted as block rewards to incentivize miners for their contributions to network security and data storage. Irys adopts a predictable inflation schedule modeled on a traditional decay curve to ensure long-term sustainability. The initial annualized inflation rate is approximately 2%, halving every four years until it reaches a terminal rate of 0.25%. This gradual reduction aligns token issuance with network maturity; front-loading incentives to bootstrap storage capacity and decentralization, then transitioning toward a deflationary equilibrium as protocol usage and fee-based rewards become the dominant economic drivers.

8.4 Burn mechanism

The $IRYS token economy is designed to exhibit strong deflationary dynamics early in the network's lifecycle, gradually transitioning toward equilibrium as usage demand offsets the decay of issuance. Inflationary rewards distributed through block subsidies are programmed to decline over time, while multiple burn and sink mechanisms continuously remove tokens from circulation, ultimately creating a net deflationary effect as network activity scales.

The primary source of this deflation arises from long-term storage fees. Payments made for data stored in extended-duration or permanent ledgers (e.g., greater than two weeks) are allocated to the protocol's endowment contract, a reserve designed to cover miners' future storage obligations. Because these endowment tokens are effectively locked indefinitely to preserve the network's data permanence guarantees, they function as a one-way economic sink, removing the corresponding $IRYS supply from active circulation. This mechanism ensures that as the total volume of stored data grows, so too does the deflationary pull exerted by the endowment.

A secondary deflationary vector is introduced through the execution fee burn mechanism. For every programmable data or smart contract transaction executed on IrysVM, 50% of the associated execution fees are permanently burned, while the remaining portion compensates the block producer. This design links token supply contraction directly to network utilization, meaning that as onchain activity, data composability, and programmable workloads increase, a proportional share of tokens will be destroyed.

Together, these two mechanisms, endowment locking and execution fee burning, establish a feedback loop between protocol usage and token scarcity. In early phases, declining block rewards amplify deflationary pressure, while in later stages, sustained demand for storage and computation ensures a continuously contracting circulating supply. The result is an economic system where long-term participation, data growth, and application activity all reinforce $IRYS's purchasing power and sustainability.

9 Fees

9.1 Minimum Fee Parameter

Irys implements a minimum fee to mitigate network spam and to ensure that tx fees can be easily denominated with the atomic units. The minimum fee is $0.001 (1/10th of a cent) as determined by Irys' price approximation mechanism. The same minimum fee is paid to the provider of ingress-proofs for publishing permanent data.

9.2 Term Fees

The pricing model for term storage determines the cost of providing storage for data, 10 replicas for n epochs

Pricing Parameter	**Value**
Annualized Cost of operating 16TB HDD	$44.00
Number of Replicas	10
Calculation	**Value**

Daily Cost per TB	$0.0075	
Daily Cost of 16TB HDD	$0.12	
Total Fee Per Epoch Storage Price (TB)	**$0.0753**	
Total Fee Per Epoch Storage Price (GB)	**$0.00007358**	
Epoch Fee Calculator	**Data Size (TB)**	**Total Fee Per Epoch Storage Price**
1 Epoch	1	**$0.0753**
5 Epochs	1	**$0.3767**

An additional 5% fee is added for inclusion in the block (scales with the size of transaction data)

1TB of Term Data in the Submit ledger (5 epochs)

term_fee = term_cost + 5%

term_fee = $0.3767 + 5% = 0.3955 -> $0.40

1GB of Term Data in the Submit ledger (5 epochs)

term_fee = $0.00039

As $0.00039 is below the network minimum fee of $0.01 the term_fee becomes:

term_fee = $0.01

Note: if repacking term partitions after they expire represents an ongoing expense to miners, this cost will be quantified and included in the term data pricing.

9.3 Perm Fees

The pricing model for permanent data has some additional factors to account for. Because the users are paying for centuries of storage upfront the model has to account for declines in the physical costs of storage (due to technological gains) over that time period. Irys chooses an extremely safe 1% annualized decline in the cost of storage as a factor for pricing permanent data. (Observed declines in storage costs over the last 50 years have been > 25% year on year.)

Pricing Parameter	**Values**
Annualized Cost of operating 16TB HDD	$44.00

Safe annual decline in cost of storage (decay rate)	1.00%
Number of Replicas	10
Years of storage paid for	200
Cost Per TB	**$2,381.54**
Cost Per GB	**$2.33**

Because permanent data must first pass through the submit ledger (term data) on its way to the publish ledger, the fee includes the cost of submit ledger storage as well.

Perm data requires 10 ingress-proofs, ingress-proofs are the same as the 5% immediate reward for including the transaction in a block. (scales with data size, shares the minimum $0.01 fee floor).

1TB of Permanent Data

perm_fee = term_fee + (ingress_fee * 10) + perm_cost

perm_fee = $0.40 + ($0.018835 * 10) + $2,381.56 -> $2,382.14

1GB of Permanent Data

perm_fee = term_fee + (ingress_fee * 10) + perm_cost

perm_fee = $0.01 + ($0.01 * 10) + $2.33 -> $2.44

If a user fails to upload data during the submit ledger term duration or the network fails to achieve the required number of ingress-proofs, the user's ingress_fee's and perm_cost are refunded when the submit ledger transaction expires at the end of 5 epochs (the submit ledger term duration)

9.4 Consensus Pricing Mechanism

The process of promoting data from the submit ledger to the permanent ledger involves multiple phases, resulting in a staged payment model for permanent data. All transactions, whether intended for permanent (perm) or temporary (term) data, initially enter the submit ledger. The payment process for term data is consistent across all transactions, while permanent data incurs additional payments to incentivize the complete publishing process.

9.5 Term Data Payment Distribution

1. User posts a transaction, including the term_fee.
2. Block producer transaction inclusion:
    a. Block producer includes the transaction in a block.
    b. Block producer's balance increases by 5% of the term_fee.
    c. Remaining 95% of term_fee is added to the treasury (tracked in block headers).
3. The user uploads data chunks associated with their transaction.
4. Miners assigned to store chunks gossip them amongst themselves.
5. Term ledger expiration payout:
    a. When the transaction expires from the submit ledger (when the partitions containing its chunks are reset at an epoch boundary), each miner is paid their portion (term_fee / 10) for all assigned chunks expiring in their partition.
    b. For a full 16TB partition, this payout is approximately $0.60 per miner.
    c. Miners continue to earn full inflation/block rewards from any blocks they produce while mining these partitions.

9.6 Additional Incentives

This payment structure creates additional incentives for miners to participate in term ledgers:

- Miners receive a payout when data expires from their partitions.
- Because miners must re-pack the partitions after expiration, this additional fee encourages ongoing participation and maintenance of the network.

9.7 Permanent Data Payment Distribution

Users pay the following fees for permanent data storage:

term_fee: Standard fee for term storage

perm_fee: Fee for permanent storage

5% of term_fee for block inclusion

5% of term_fee for each ingress-proof

9.8 Fee Distribution

term_fee: Processed identically to regular term data transactions.

Block Inclusion Fee: 5% of term_fee paid immediately to the block producer including the transaction.

Ingress-Proof Fees: 5% of term_fee for each ingress-proof provided.

Perm_fee: Prepaid amount covering 200 years x 10 replicas with 1% annual decline in storage costs. Added to the treasury.

10 Submit Ledger Expiry (Epoch Boundary) Processing

Refund Scenario
If transaction data was never uploaded:

Ingress-proof fees and perm_fee are refunded to the uploader.

Promotion Scenario

If data was promoted to permanent storage:

Protocol inspects all permanent transactions with ingress-proofs.

Pays out the ingress-provers.

10.1 Epoch Boundary Payment Distribution Tasks

For each expiring submit ledger transaction:

1. Inspect the transaction to determine if it was intended for the publish ledger.
2. If intended for publish ledger, check if it arrived:
    a. If Published: Reward ingress-proof submitters with their 5% rewards.
    b. If not Published: Refund perm_fee and ingress-proof fees to the address that posted the tx.
3. Tabulate the amount of data posted to the expiring partition.
4. Pay each partition owner the term_fee for storing that amount of data.

11 Roadmap

11.1 Scaling programmable data

Programmable data, at its core, represents a new computational paradigm where datasets are no longer passive records but active, composable building blocks for onchain applications. It establishes a universal data layer: an openly accessible substrate where developers, users, and protocols can read, write, and execute logic directly against verifiable, onchain data. Unlike traditional blockchains that separate storage from execution, programmable data unifies both, creating a shared environment where information, compute, and value flow seamlessly across applications.

A critical evolution of this model will come through Programmable Data Layer-2s (PD L2s): scaling environments that extend IrysVM's compute capacity while maintaining trustless interoperability with Irys's global dataset. PD L2s function as specialized execution domains, enabling parallel computation, lower latency, and domain-specific optimizations without fragmenting state or data availability. They are natively anchored to Irys's data layer through verifiable proofs, ensuring that all reads, writes, and computations performed off-chain remain cryptographically linked to canonical data on the base layer.

Within this architecture, the Irys dataset bifurcates into public and private states. The public state represents permissionless data: open for anyone to access, query, and integrate into their own smart contracts or applications. This enables composability across projects, allowing ecosystems such as AI agents, analytics protocols, or DePIN networks to build atop shared datasets. Licensing primitives at the protocol level can embed economic logic directly into data access, enabling creators to monetize usage through programmable royalties or usage fees.

The private state, by contrast, leverages privacy-preserving compute primitives, such as zero-knowledge proofs or secure enclaves, to enable sensitive or proprietary data to be processed within verifiable but confidential environments. This ensures that organizations or users can run computation over private datasets while maintaining cryptographic guarantees of correctness and auditability.

Ultimately, Programmable Data L2s extend Irys into a multi-tier execution ecosystem where scalability, privacy, and interoperability converge. They transform Irys from a datachain into a universal compute and coordination layer, where any dataset, public or private, can become a living component of the onchain economy.

11.2 Fast blocks and fast finality

Building applications on Irys fundamentally depends on the relationship between block time, finality, and composability. Faster block times reduce the latency between state updates, enabling applications, particularly those reliant on real-time or near-real-time data, to respond to onchain events with minimal delay. This is critical for use cases such as inference caching, AI model coordination, or automated data pipelines, where sub-second responsiveness can meaningfully impact both user experience and system efficiency.

For Programmable Data L2s, fast finality becomes even more essential. These Layer-2 environments rely on rapid confirmation from the base layer to maintain deterministic synchronization with Irys's global dataset. When finality times are short, L2s can frequently checkpoint their state back to the base chain, ensuring consistency and reducing the risk of data divergence or rollbacks. This enables cross-domain composability, where applications deployed on separate L2s can securely interoperate through the shared Irys dataset without introducing excessive latency or trust assumptions.

In practice, faster finality allows developers to design more interactive and modular systems, where data stored on one partition can immediately trigger computation or logic on another. For example, an AI inference engine operating on an L2 could instantly consume new sensor data written to the base chain, compute predictions, and commit results back to Irys in near real time.

To reach these performance targets, Irys will continue exploring Byzantine Fault Tolerant (BFT) consensus variants that can complement or extend its hybrid PoW/S model, with the goal of enabling tighter block confirmation and deterministic finality across partitions. Such mechanisms would allow nodes to agree on block validity within bounded time while preserving the probabilistic security of Proof-of-Work. By integrating BFT principles into the consensus pipeline, Irys aims to minimize propagation delays, enhance network responsiveness, and ensure composable finality between L2s and the base layer. Collectively, these optimizations position Irys as a high-performance execution substrate, capable of supporting complex, data-driven applications with the same responsiveness expected from modern cloud infrastructure, but with the trust guarantees of a fully decentralized network.