IPI

Scalability Reimagined: A Dual-Phase Architecture for Global Blockchain Adoption

January 31, 2025

v1.0

www.ipi.io

Abstract

Current blockchain solutions face a fundamental barrier in the form of **limited throughput**, **high transaction fees**, and **long or uncertain finalization times**. These constraints effectively hinder the technology's adoption across many potentially **groundbreaking applications**, including **global financial services**, **micropayments**, and **real-time systems** such as **gaming**, **auctions**, and **social platforms**. Addressing these issues is **essential** if blockchain is to evolve into a **widely**

accessible infrastructure for **diverse services** and **applications** on a **global scale**.

By drawing on techniques from **mature distributed systems**, the proposed architecture seeks to resolve the industry's **long-standing scalability dilemma** without sacrificing **core principles** of **security** and **decentralization**. This approach leads to an **environment ready for widespread use**, enabling **truly advanced blockchain applications** in domains where earlier limitations had been **insurmountable**.

Building on a **novel consensus mechanism** and a fundamentally different approach to **processing smart contracts**, the proposed solution offers **virtually unlimited scalability** for both **transaction processing** and **state storage**, while preserving **decentralization**. As the number of **participating nodes** increases, so does the system's **data-processing capacity**, which translates into **significantly reduced transaction costs** and paves the way for **applications that were previously out of reach**. At the same time, **rapid and reliable transaction finalization** allows for **real-time operations**, while the **decentralized network structure** makes **censorship more difficult**, **mitigates manipulative practices** (including **Miner Extracted Value**), and **incentivizes broad participation** in **securing the chain**.

This document provides a **detailed overview** of the **consensus mechanisms** underpinning this concept, outlining a **vision of an ecosystem** where **speed**, **scalability**, and **low costs** go hand in hand with **reliability** and **full decentralization**.

Typographic Conventions

To enhance readability, the following typographic conventions are used throughout this publication:

- `Constant` – A value that can be adjusted through optimization based on experiments and may change as a result of governance decisions. Descriptions and proposed values of constants can be found in the **Constants** section at the end of this publication.
- **[2]** – A bibliographic reference. The full list of references is available at the end of this publication.
- ***Concept*** – Terms introduced or used specifically for this publication. Their definitions and explanations are provided in the **Terminology** section.

Smart-contract's scalability problem

Imagine a blockchain network that becomes faster and more efficient simply by adding new nodes. This is the promise of *horizontal scalability*. In practice, though, achieving a horizontally scalable system—where transaction verification, processing, and the entire smart contract state are split across many nodes, and no single participant needs to store the entire dataset—is remarkably difficult. Below is an overview of why this problem is so complex and what researchers are doing to solve it.

What Does "Horizontally Scalable" Mean?

In a traditional blockchain, each full node stores every transaction ever made and tracks the entire ***global state*** (e.g., account balances, smart contract variables, etc.). Every transaction is validated by every node, which places a hard cap on how many transactions a blockchain can process. If each node has to do *all* the work, then adding more nodes won't effectively boost throughput—it mostly just makes the network more robust to failures.

A ***horizontally scalable*** design, however, distributes the work so that *each node handles only a portion* of the total data and transactions. This approach, often called ***sharding***, promises the best of both worlds: more nodes result in more capacity, letting the network handle more transactions without requiring every node to process everything.

Why Sharding Is So Complicated

Splitting State and Data

Sharding a blockchain's state means distributing account balances, smart contract data, and transaction histories across multiple ***shards*** (sub-networks). The challenge lies in deciding how to partition this data. Because transactions depend on the outcomes of earlier transactions, a contract on ***Shard A*** might suddenly need state from ***Shard B***. Retrieving or passing execution across the network can become a bottleneck, making it difficult to maintain high throughput. There is no straightforward way to ensure that these cross-shard queries remain rare, and approaches designed to optimize them quickly grow in complexity.

Cross-Shard Transactions and Smart Contract Calls

When a transaction affects data within just one shard, processing is relatively straightforward: that shard's validators come to consensus, update their local state, and move on. However, many transactions need to work across shards. For example, a user might transfer tokens from ***Shard A** to ***Shard B***, or a DeFi contract on ***Shard A*** might call a lending protocol on ***Shard B***. In these cases, multiple ***shards*** must coordinate. One common way to keep everything in sync is through an *atomic commit* or a *two-phase commit* process, which ensures either all ***shards*** are

updated, or none are. This process adds extra overhead, slows throughput, and introduces more points of failure.

Smart contract calls across ***shards*** can be particularly tricky. Today's blockchain applications often rely on ***composability*** — the ability for one contract to instantly call another contract and use its output. ***Sharding*** forces these interactions to become *asynchronous*, which can make it harder to build complex, interdependent protocols. If a contract needs a real-time response from another contract on a different ***shard***, it has to wait for the other ***shard*** to finalize that transaction, adding latency and creating potential bottlenecks.

Network Bandwidth and Propagation

Spreading the work across ***shards*** doesn't just require the network to handle more transactions in parallel—it also demands that each ***shard*** constantly communicates with the others. Nodes need to exchange proofs, transaction receipts, and final states to ensure that if one shard updates its records, other shards know about it when they need that information. As the number of ***shards*** grows, so does the amount of cross-shard traffic. If not carefully managed, this overhead can become a new bottleneck, canceling out the benefits of ***sharding***.

The problem of *data availability* also looms large. Because no single node stores the entire dataset, the network must have robust mechanisms to guarantee that critical data is not lost. Various protocols employ techniques like *erasure coding* (splitting data into many encoded fragments) or *data availability sampling* (randomly checking small chunks of data) to ensure that even partial storage can reconstruct the full state when needed.

Consensus and Finality

To finalize transactions, each ***shard*** typically runs its own mini-consensus protocol. But these local consensus processes must come together under a global mechanism that ensures consistency across all ***shards***. If a blockchain "reorganization" (or reorg) happens on one shard —meaning the order of recent blocks changes—this can impact related transactions on other ***shards***. Coordinating these potential rollbacks or state updates across many ***shards*** is a significant challenge.

Ongoing Efforts and Current solutions

Various blockchain projects have attempted to tackle the state ***sharding*** problem, each with its own trade-offs:

- **Data Availability Sharding** (e.g., Ethereum's Danksharding/Proto-Danksharding) focuses on splitting the storage of transaction data rather than truly dividing ***global state***. While it boosts throughput and lowers storage costs, developers still operate within a single monolithic state, so global transactions remain sequential.
- **Object-Based Sharding** (e.g., Aptos, Sui) replaces account-based models with independent "objects." Transactions touching separate objects can run in parallel, improving performance.

However, frequent interactions among multiple objects reduce these gains, and developers must carefully design their apps to avoid cross-object bottlenecks.

- **Parachains/Sidechains** (e.g., Polkadot, Cosmos) let each chain maintain its own state, communicating via protocols like XCMP or IBC[1]. This effectively scales at an ecosystem level, but applications spread across multiple chains face added complexity in bridging and cross-chain communication.
- **Adaptive or Dynamic State Sharding** (e.g., NEAR, Elrond/MultiversX, Shardeum) attempts to truly partition a single global state[2] [3]. Although this can increase throughput, cross-shard transactions become more complex, and developers must handle asynchronous communication, shard-balancing, and potential "hot shards."

In all these approaches, "real" unlimited scalability remains elusive. Cross-shard/cross-chain interactions still introduce overhead and complexity, placing additional burdens on developers to manage asynchronous execution and architectural constraints.

Conclusion

Building a horizontally scalable blockchain that supports smart contracts without requiring every node to maintain the full state is one of the hardest engineering problems in the blockchain space. The fundamental tension arises from needing to divide data and workloads across ***shards** while still maintaining secure, atomic transactions and preserving the ***composability*** that has made blockchains like Ethereum so powerful.

Every potential solution—whether it's sharding, rollups[4], or a hybrid of both—comes with its own set of trade-offs. Increased network overhead, complex cross-shard coordination, and heightened security considerations are just a few of the challenges involved. Nonetheless, ongoing research and development are pushing these boundaries, making it increasingly likely that future blockchains will achieve a much higher degree of scalability without sacrificing the core values of decentralization and trustlessness.

New solution

As transaction and smart contract processing in ***shards*** is notoriously difficult—yet crucial to achieving unbounded scalability—an innovative strategy is essential: namely, breaking down the problem into two smaller, more manageable tasks.

A key insight is that transactions in blockchain are executed deterministically. If we know the input data and the precise order of execution, we can always compute the resulting state of the entire network. Moreover, because each transaction depends only on a limited set of previous transactions, it becomes feasible to compute outcomes for only part of the transaction set and still arrive at the same result as if we had processed them all. In other words, as long as we preserve the transaction order, the consistency of results remains intact.

Therefore, by first using a ***consensus mechanism*** to establish the order of transactions and then focusing on their execution, we effectively split a large, complex challenge into two solvable problems. This separation of concerns makes it possible to employ a scalable consensus layer dedicated solely to finalizing the global transaction sequence. Once that order is locked in, the actual processing and state updates can happen in a distributed manner. By decoupling transaction ordering from execution, we avoid the bottlenecks that plague systems where these two steps are tightly coupled, ultimately paving the way for truly limitless scalability.

Hereby, I present a new algorithm for scalable blockchain solution: **Order Now, Execute Later**.

Order Now Execute Later

The ***ONEL*** algorithm consists of two stages that together enable correct execution of arbitrarily large numbers of transactions:

Order Now
This phase establishes the transaction execution order, resulting in a list of transactions in a fixed sequence. It is run in a **scalable**, **distributed** manner under consensus, and its output is also ***sharded***. Once this ordering is determined, the outcome of those transactions—even if not yet computed—is already guaranteed. Because transactions are deterministic and their order is fixed, there can be only one possible result.

Execute Later
In this step, network participants actually execute the transactions to learn their outcomes. Since execution is fully deterministic, validators do not need to know the results of transactions before starting to compute the next block. ***Distributed execution*** is feasible because the outcome of any single transaction depends only on the earlier transactions that affect it, not on the entire ***global state***.

Overall Network State

Research ruled out the possibility of making transaction outcomes a direct part of the network state subject to consensus. The detailed reasoning behind this is presented in the **Late Execution Problem** section. However, this does not prevent the fully computed state from being accessible to users; it is simply not essential for the network's core operation.

How Validators Collect Transaction Fees

Executing a transaction yields a reward for the validators responsible for including that transaction in a block. It is therefore crucial for a validator to know the account balance of the user who pays for the transaction; otherwise, the validator cannot collect the fee, and the transaction will remain unexecuted. Consequently, the block proposer has an incentive to determine the user's balance with some level of confidence (though not necessarily with absolute certainty if execution results are delayed). How the validator obtains this information—

whether by computing transactions directly or by retrieving it from the network—is at the validator's discretion. This is explored in more detail in the **Validator Perspective** section.

Distributed Ordering

In addition to producing blocks within **shards**, the network also generates blocks that confirm results from all ***shards***. It is possible to have more than two levels, forming a tree-like network structure, which is described in detail in the **Multi-level Sharding** section.

Distributed transaction ordering is achieved by dividing all transactions into batches handled by individual ***shards***, determining the order of transactions within each ***shard***, and then finalizing the order of ***shard*** outputs. This process yields the overall ordering of transactions across the entire network.

For example, consider the following transactions labeled:

```
314, 023, 281, 239, 131, 101,
246, 311, 248, 123, 280, 318
```

We assign these to ***shards*** 1, 2, 3, and 4 in order:

```
shard 1: 314, 023, 281
shard 2: 239, 131, 101
shard 3: 246, 311, 248
shard 4: 123, 280, 318
```

Within each ***shard***, the ***consensus mechanism*** determines the transaction sequence. As an example (here using simple sorting for illustration):

```
shard 1: 023, 281, 314
shard 2: 101, 131, 239
shard 3: 246, 248, 311
shard 4: 123, 280, 318
```

These ordered transactions are placed into blocks in their respective ***shards***. Information about these blocks is then considered in the overall network consensus. At the global level, the final order of these ***shard*** blocks is decided based on their block hashes, for instance sorted by hash. Suppose in the ***Root*** block, the resulting order of ***shard*** block hashes is:

```
Shard 2 block hash, Shard 3 block hash,
Shard 1 block hash, Shard 4 block hash
```

From this, any network participant can reconstruct the global transaction order:

```
101, 131, 239, 246, 248, 311
023, 281, 314, 123, 280, 318
```

Once the order is set, the outcomes—though not yet computed—are guaranteed, meaning the consensus is achieved at this point. A detailed description of the algorithm can be found in the **Multi-level Sharding** section.

Distributed execution

After determining the global transaction order, the next step is execution. This is non-trivial because processing a large number of transactions on a single node would take too long to be practical.

The key is to allow execution of only a chosen subset of transactions, so a user can verify their own transaction without computing the entire network state. The network state is stored in a ***K->K->V database*** (Key->Key->Value) because such databases (e.g., Cassandra-like[5]) scale well and are straightforward to implement. In this specific case, it is ***Address*** -> Key -> Value. A detailed description of this database structure is in the **Blockchain state** section.

All transactions carry information in their headers about which values (address and key) they depend on and which they modify. The structure of these transactions is described in the **Transaction structure** section. These header details, independent of the transaction's actual execution, can be computed before publication and make it possible to index transactions by their input and output keys in the underlying ***K->K->V*** storage.

To execute a single transaction, one must know the state on which it operates. To do so, for each input value, one locates the past transactions that affected that value by searching the output index. Recursively, one can reconstruct the entire transaction graph all the way back to the genesis transaction. However, reconstructing the entire graph is not always necessary; one can stop where the state is already known. For a "thin client," such state data can be fetched from the network.

Validators can store states resulting from historical transaction execution in their databases or obtain them from external servers specializing in transaction processing (see **Information market** for more details). Servers that perform large-scale transaction execution can reverse the process: given a set of already-known values, they can look for additional transactions that can now be processed quickly. Although the algorithm for large-scale batch transaction execution has been explored, its detailed description is beyond the scope of this publication.

Multi-level Sharding

Since the goal is **unbounded** scalability—not just "high" scalability—the ***sharding*** algorithm is multi-level. With a small number of transactions, there may be only one level (the main, or ***Root*** blockchain). As the number of transactions and validators grows, the system first splits into ***shards*** (creating a second level), and then, when each ***shard*** reaches `MaxShardsPerLevel`,

those ***shards*** can themselves split further on the same principle, effectively creating a hierarchy of ***shards***.

Because validators need access to account states from which they collect transaction fees, the network is split into ***shards*** based on addresses. Specifically, each ***shard*** is determined by a certain number of leading bits in the address. ***Addresses*** are assumed to be roughly uniformly distributed, so this method balances load across ***shards***. Even if distribution is not perfect, the shard-splitting logic (described below) addresses any imbalance. Though range partitioning by entire addresses is theoretically possible, it is deemed unnecessarily complex for this design.

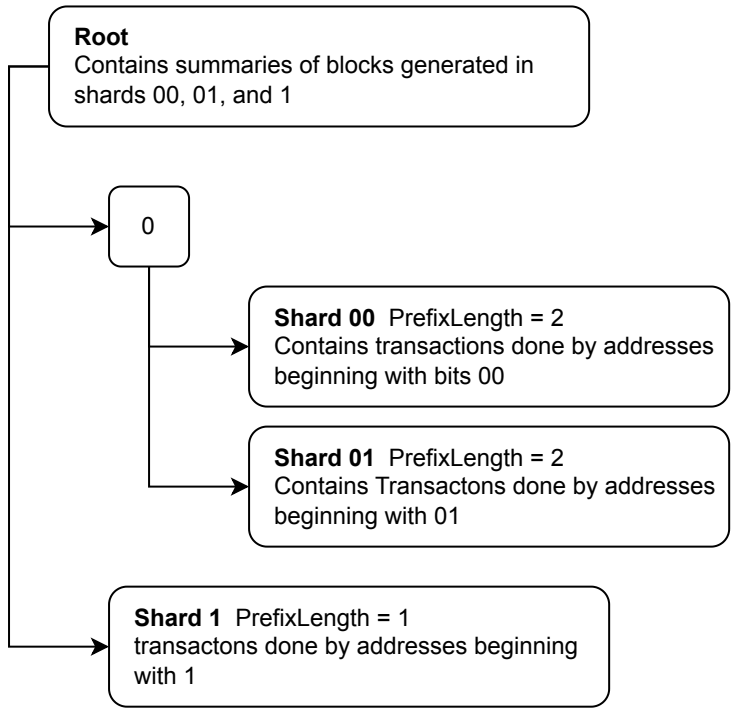
Distributed architecture

The network uses a binary-tree-based approach to ***sharding***. Each ***shard*** is labeled by the **number** of leading bits in the address, referred to as **`PrefixLength`**. In other words, a ***shard*** with **`PrefixLength = k`** encompasses all addresses whose first **`k`** bits match that ***shard's*** prefix. At the start, **`PrefixLength = 0`**, meaning **all** transactions go to the **`Root`** ***shard***.

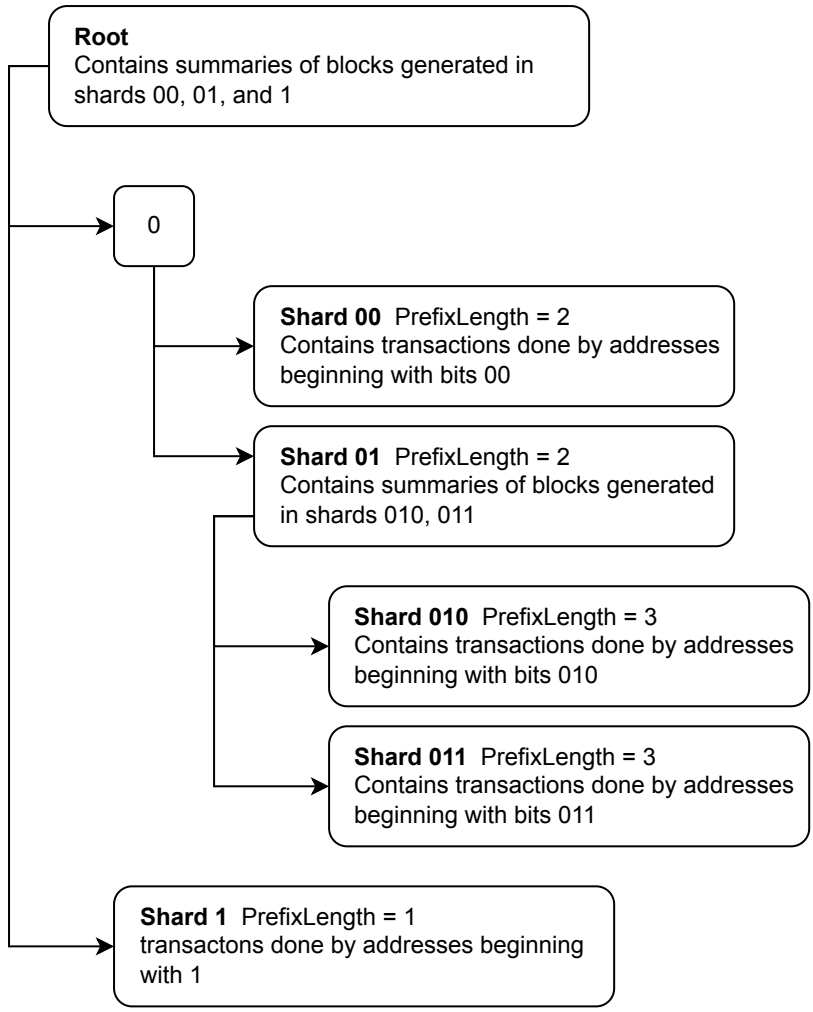
If a ***shard's*** block size grows beyond **`HighWatermark`** and there are at least **`EnoughValidatorsPerShard`** validators available in each of the would-be new ***shards*** (validators are assigned by address), **`PrefixLength`** is incremented by 1 for that shard, splitting it into two. This division occurs after **`ShardDivisionDelay`** blocks at the ***Root*** level, giving validators time to prepare.

Splits continue until **`PrefixLength`** reaches **`MaxPrefixLengthPerLevel`**, at which point additional splitting in that branch is disallowed to avoid the ***Root*** block size exceeding its limit. The ***shard*** becomes a ***SubRoot****, under which further ***shards*** can be created if needed.

Example network structure:



More advanced example (with **`MaxPrefixLengthPerLevel = 2`**):



If the total block size in two adjacent ***shards*** falls below **`LowWatermark`**, or the number of validators in either ***shard*** drops below **`LowValidatorsPerLevel`**, the shards must be merged after **`ShardingConcatentionDelay`** ***Root*** blocks. In an extreme case, if the number of validators is below **`MinValidatorsPerLevel`**, ***shard*** block creation may be impossible; after merging, their transactions must be relocated into a new connected shard blocks.

Additional conditions—like minimum stake levels in a ***shard***—are possible but not mandatory. Validators only handle transaction ordering, which does not affect actual execution. Validators can also migrate among shards to maximize their rewards, as described in a later section.

Validator Stake Set

It is necessary to maintain a list of validators in order to determine when enough votes have been reached in each ***shard's*** consensus.

Operators of full nodes who wish to participate in block creation register as validators by locking a specified amount of coins as their stake. This validator list is kept in a specialized, scalable smart contract that stores it in a form that can be divided between ***shards***.

Validators submit a stake request for a specified range of ***Root*** blocks in the future, along with a designated stake amount. Each request includes:

- A starting ***Root*** block at least `ValidatorEnterDelay` blocks ahead of the current ***Root*** block.
- A stake that will be locked in the smart contract.

Besides participating in block creation in their specific ***shard***, validators also partake in consensus for the ***Root*** block and any relevant ***SubRoot*** blocks. This design is natural since validators have a vested interest in seeing these blocks published; their rewards depend on the ***Root*** block referencing their ***shard's*** block hash.
For example, if there are ***shards*** `00`, `01`, and `1`, and a validator's address starts with the bits `010111`, that validator takes part in creating blocks in the `01` ***shard*** and in the ***Root***.

Shard level consensus

Block proposers and ***shard validators*** are selected using a Verifiable Random Function (***VRF***), in much the same way as in Algorand's *Pure Proof of Stake* protocol[6]. Each validator locally combines their private key with the hash of the ***shard's*** previous block to produce a pseudorandom value, accompanied by a cryptographic proof. This proof allows anyone in the network to verify that the proposer or validator was chosen fairly, without requiring every node to coordinate when drawing "lots."

Below is a brief outline of the consensus process, showing how it parallels standard *Pure Proof of Stake*:

Proposal:
After calculating the ***VRF*** result, one or more **validators** who "win the lottery" become eligible to propose a block. A chosen proposer compiles the ***shard's*** transactions into a new block and broadcasts it alongside the ***VRF*** proof of selection.

Validation (Soft Vote):
Another random subset of validators (also selected by their ***VRF*** values) then reviews the proposed block. These validators verify the block's validity and, if it meets all criteria, cast votes in favor. If the proposal collects sufficient votes (usually a 2/3 threshold within the committee), the network reaches an initial agreement on this block.

Certification (Certify Vote):
Finally, a subsequent step of voting (often by the same or a newly selected committee) certifies the block. This additional phase provides finality, ensuring that once a block is certified, it cannot be revoked under normal circumstances.

Adaptation to Sharding and Merging
When a new shard is formed—splitting off from an existing one—the ***VRF*** uses the hash of the block immediately preceding the split, combined with the new shard's identifier. This ensures

that each shard's consensus process starts from a unique yet verifiable reference point. In the case of merging shards, the "previous block hash" becomes a function of the final blocks from each shard involved, preserving a continuous chain of trust without interruption.

Differences in Rewards
While the core mechanics of proposing and validating blocks mirror *Pure Proof of Stake*, this system uses a specialized reward structure tailored for a sharded environment. Validators earn rewards in a manner that reflects their shard-level participation, ensuring incentives remain aligned as the network scales.

Through this design, each shard maintains fast ***block finality*** and strong Byzantine fault tolerance, leveraging the proven concepts of *Pure Proof of Stake* while supporting a scalable, tree-like structure of multiple shard blockchains and a root (or coordinating) chain.

Validator perspective

For the network to operate efficiently and securely, it is essential to align incentives so that validators are motivated to perform the necessary tasks correctly.

Incentives

To ensure validators remain motivated to produce blocks, there are two kinds of rewards:

Block reward – A fixed number of coins (subject to halving).
Validator fees specified in transactions – Consisting of a fixed reward plus "burned gas" fees.

Because account states are not directly part of the consensus, validators have no strict obligation to verify whether there are enough funds to pay the fee. They could include transactions in blocks with no validation or even invalid transactions. However, it is in their interest to claim these rewards, so they will likely check whether the transaction can pay them.

With sharding according to address, a validator only needs to store states for the addresses covered by its shards (and for relevant validator stake data), rather than the entire network state.

Although there is no requirement for validator rewards to be in a specific currency or token, they must be in a token that the validators accept. Naturally, the validator stake itself must be in the network's base currency.

Transaction fees are shared among validators across all levels, meaning fees from a block in ***shard*** `11` also reach validators of the ***Root*** block. The fee split is percentage-based, with **`HigherLevelTransactionCostShare`**% flowing to higher-level validators.

Block rewards are shared among ***Root*** validators and the validators of its ***shards***. Specifically, **`LowerLevelBlockPrizeShare`**% of the reward is allocated to ***shards*** according to their address space size — for instance, a ***shard*** with prefix `0` might receive half, while a ***shard*** with

prefix `10` receives a quarter. Within a ***shard***, rewards are divided among the block proposer (who gets **`ProposerShare`**%) and all other validators, proportionate to their voting power in that ***shard***.

This reward split provides proper incentives for validators to propose and validate blocks.

Information market

Because validators and other parties have an interest in real-time account states—both to collect transaction fees and for other reasons—servers that execute transactions can operate independently of validators, creating opportunities for data exchange against token payments. Transaction-execution servers benefit by earning tokens, and validators benefit from higher rewards, as they can more easily discard transactions lacking sufficient funds.

Transaction processing servers may, in turn, rely on other indexing servers, and so on, creating a fully distributed system where each participant profits from their chosen role. Payments can be facilitated by a smart contract creating a payment channel, with data exchange accounted for off-chain and final settlement occurring upon channel closure. Although this protocol is straightforward, a detailed explanation goes beyond the scope of this publication.

Additionally, a validator can acquire the same piece of information from multiple sources and compare it for consistency to detect fraud. Computing and indexing servers must digitally sign all data they provide; if they supply misleading or incorrect information, the validator can propagate that signed evidence across the network as proof of misconduct, reducing the offending server's reputation. In a fully automated information-exchange market, a decentralized reputation system would be indispensable to prevent such abuses; however, the details of such a mechanism lie beyond the scope of this publication.

Late execution problem

During research, a particular concern arose: What happens when a poorly written, non-scalable smart contract is flooded with a large volume of transactions? For example, consider a contract that returns a "next random number" dependent on its previous result, effectively a linked pseudo-random number generator. A large influx of queries could create a chain of dependent transactions that cannot be processed quickly on a single machine, as each call relies on the previous result and thus cannot be efficiently parallelized.

This challenge explains why the entire network state at a specific point in time cannot simply be subject to consensus. Although it is possible in theory, there is no guarantee that the ***global state*** can ever be fully computed. Even if the transactions are included in blocks and the consensus is finalized, the actual results of these transactions might take a very long time to materialize. If such transactions continue flooding the network, in the worst case, the final outcome might never be computed.

While such an issue may seem minor in the case of random numbers, consider a contract representing a rudimentary roulette wheel that pays out based on the generated number. A large backlog of dependent transactions would leave the user's final balance unknown for an extended period. In the worst case, they could not execute further transactions since their actual account balance remains indeterminate.

The next section details mechanisms to mitigate this challenge.

Blockchain state

The network state is stored in a ***K->K->V*** (Key -> Key -> Value) database structured as ***Address*** -> Key -> Value, where the address always refers to a specific smart contract. Only the smart contract code that controls those values can modify them, though any contract may read them. The second key may itself be an address, for example representing an account.

All account states and transactions are managed by smart contracts. To facilitate basic network usage, the genesis block contains contracts for fundamental operations (e.g., staking, transferring). All values are stored in binary of arbitrary length up to **`MaxValueLength`**, and keys are also binary up to **`MaxKeyLength`**.

Due to the late execution problem, besides a simple "write" operation, this design incorporates additional operations to help manage dependencies. Specifically, there are **Add** and **Subtract** operations that works with integers, which are commutative. If a transaction that adds to an accumulator (a numeric field) is not fully computed, it can be skipped for the moment when checking if enough funds exist, as long as it is an additive credit. You cannot skip a subtraction result, because that would allow double spending.

For example, consider a contract similar to ERC20 tokens and a second overloaded contract ("roulette"). If a user who has previously sent tokens to the roulette contract tries to pay for another transaction, a validator wants to know whether the user's account can cover the fee. The user's exact balance might hinge on the unresolved roulette winnings. However, the validator does know how many tokens the user initially sent to the roulette contract; all known payments are accounted for, but not all potential gains. If the sum of all known gains exceeds the user's known losses, the validator may decide to publish the transaction, hoping it will indeed be paid. Smart contracts themselves can use similar logic.

Hence, five operations are possible on any value in this database:

- **Get** – Read a value as binary data
- **Set** – Write a value as binary data
- **Add** – Add to a value
- **Subtract** – Subtract from a value (must not result in negative values)
- **IsGreater** – Check if a value is greater than another

Additionally, smart contracts can perform range queries in the database. This structure is flexible enough to allow implementation of most standard IT solutions in a scalable manner.

Transaction structure

A transaction in a block consists of:

A list of database values to be read (including whether it uses **Get** or **IsGreater**).
A list of database values to be modified (including whether it uses **Set** or **Add** or **Subtract**).
The address of the smart contract the transaction will call.
A procedure number identifying the method to invoke on that contract.
Parameters passed to the smart contract.
A validator fee, which includes a fixed component and a gas price.
A memo field.
A signature.

These value lists include the specific keys in the database that identify them. Optionally, they may also carry the expected values themselves; in that case, the transaction only executes if the contract's resulting state matches the transaction's requirements. This both helps protect the user and provides an additional channel for passing data to the contract.

For instance, let's look at a transaction that transfers main blockchain currency from address AAA to address BBB on a contract with address `0000`:

```
IN: 0000/AAA IsGreater 2310
OUT: 0000/BBB Add 1000
OUT: 0000/AAA Subtract 2310
CONTRACT: 0000
PROCEDURE: 1 (transfer)
PARAMETERS: ---
FEE: 10
MEMO: "funds transfer"
```

Here, `0000` is repeated for clarity, but in actual serialization it would only appear once. The address references are still required in case multiple contracts are involved—for instance, if a swap contract `8686` is exchanging token `0000` for token `2323`:

```
1. IN: 0000/AAA IsGreater 2000
2. IN: 2323/8686 IsGreater 30000
3. OUT: 0000/AAA Subtract 2000
4. OUT: 0000/8686 Add 2000
5. OUT: 2323/8686 Subtract
6. OUT: 2323/AAA Add
7. CONTRACT: 0000
```

8. PROCEDURE: 2 (call another contract)
9. PARAMETERS:
    1. CONTRACT: 8686
    2. TRANSFER: 2000
    3. PROCEDURE: 4 (sell operation)
    4. PARAMETERS: 2000 (amount)

Execution proceeds so that AAA first calls contract 0000, which then calls contract 8686, transferring 2000 tokens. Contract 8686 in turn calls 2323, giving tokens to AAA. The above is illustrative; the actual implementation may differ.

Example transaction data flow

Consider a transaction transferring coins from address X (starting with 1100) to address Y (starting with 0010), in a network split into ***shards*** 00, 01, 10, 11:

The user broadcasts the signed transaction to the appropriate full-node(s).
The transaction is received by validators of ***shard*** 11 and placed into their mempool.
The proposer chosen by ***VRF*** includes the transaction in ***shard*** 11's block and produces the block.
The proposed block is validated and confirmed by enough validators with matching ***VRF*** results.
Once voting results in ***shard*** 11 are confirmed, the block hash is sent to the ***Root shard*** mempool.
The ***Root*** block proposer includes the ***shard*** 11 block hash (as well as hashes for ***shards*** 00, 01, 10).
After ***Root*** block validation (via VRF-based voting), the block is confirmed.
With the ***Root*** block containing the ***shard's*** block hash, the transaction is considered published.
Any network participant can validate the transaction and compute the new balance for address 0010 (and also for 1100).
At this point, validators can account for the transaction's outcomes when deciding on future transactions from address 0010.

P2P information routing

Because the goal is unbounded scalability, we cannot rely solely on traditional broadcast mechanisms. A specialized information-routing protocol is required.

The solution is a distributed pub-sub model. Transactions submitted to a node are not relayed to every peer but only to those subscribing to a specific portion of the network and its topics. Such a subscription includes a prefix (the initial address bits or ***shard*** prefix) and a topic. Possible topics include:

- Mempool transactions by source address—required by validators
- ***Block*** proposals by ***shard***.
- ***Block*** votes by ***shard***.
- ***Blocks*** by ***shard***—e.g., for prefix `1`, blocks of ***shards*** `10` and `11` are forwarded
- Confirmed transactions by address, key set, and block range
- Transaction status by transaction hash—whether it is in a mempool, confirmed, and in which block(s)
- The latest transaction by address and key

Similar to GraphQL, the protocol allows queries to contain templates for further sub-queries, populated by data returned from the initial query. This design permits fetching a transaction along with the transactions on which it depends, up to a specified graph depth or block range.

Once a node has relayed a piece of information, seeing the same information again from another source does not trigger repeated distribution. Only new, previously unseen data is forwarded, reducing redundant bandwidth usage. In a range query, any locally cached data is sent immediately in the response. If a node already has the relevant block or transaction, it can signal the sender to stop, further reducing unnecessary network traffic.

Illustrative Transaction Flow (Shards and Root-Level Events)

Consider a transaction transferring coins from address `X` (`1100`) to address `Y` (`0010`) in a system split into ***shards*** labeled `00`, `01`, `10`, and `11`:

Transaction Creation
A client creates and signs the transaction, then sends it to one or more full nodes that can publish it to the network.

Initial Routing
Upon receiving the transaction, a full node examines the signer's address to determine which ***shard*** should handle it—here, that is ***shard*** `11` for address `1100`.

- If the node itself does not belong to ***shard*** `11`, it routes the transaction toward the appropriate validators.
- This routing follows a logic similar to a Distributed Hash Table (DHT), where each node knows enough about nearby ***shards*** to forward transactions correctly.

Shard Mempool Dissemination
Once the transaction arrives at a validator in ***shard*** `11`, it propagates among all validators in that ***shard's*** mempool. ***Shard*** validators strive to connect with peers in the same ***shard*** to ensure they can propose and validate blocks effectively.

***Block* Proposal**

A validator that includes this transaction in its mempool might “win” the ***VRF*** lottery, qualifying it to propose the next shard `11` block. If chosen, it packages pending transactions—including ours—into a new block and broadcasts it, along with the ***VRF*** proof of selection.

***Block* Broadcast**

The proposed block is relayed to other validators interested in shard `11`. These nodes propagate the block further within the ***shard***.

Voting (Soft + Certify)

- **Soft Vote**: A separate, randomly selected committee of validators (again chosen by ***VRF***) quickly reviews the proposed block. If it is valid and meets consensus requirements, they cast votes approving it.
- **Certify Vote**: The same or another ***VRF***-chosen committee then issues a final certification vote. This final step ensures the block achieves true Byzantine fault tolerance, making it effectively irreversible under normal conditions.

  Both sets of votes are distributed through the shard similarly, ensuring all relevant nodes receive them before the round closes.

Forwarding to *Root*

After the block in ***shard*** `11` is fully certified, its **block hash** is sent to the ***Root shard’s*** mempool. This forwarding uses the same type of DHT-like routing mentioned before, ensuring the ***Root*** chain validators learn about new shard blocks.

- At the ***Root*** level, a similar ***VRF***-based process (proposal and voting) occurs. Once the ***Root*** chain finalizes the reference to shard `11`’s block, the transaction’s ordering is recognized globally.

Completion

When the ***Root*** chain includes the reference to ***shard*** `11`’s certified block in one of its own blocks, the transaction’s ordering phase is complete. At this stage, the result of the transaction is locked in, and any interested party can deterministically compute the updated state.

This flow shows how each ***shard*** handles its own consensus process—very similar to *Pure Proof of Stake*—while still integrating seamlessly into a higher-level, ***Root shard*** consensus. The transaction is thus confirmed in two layers:

Shard Finality for local ordering of transactions and block creation.
Root Finality for integrating shard blocks into the global ledger view.

In doing so, the network balances local scalability (multiple ***shards***) with a strong global ordering guarantee (***Root*** chain).

This concludes the translated segment, maintaining the original structure and level of detail.

Decentralisation and Censorship Prevention

Thanks to the use of ***sharding***, we can maintain a relatively small block size, making it possible to process blocks on modest hardware. Because validators only handle transaction ordering—while actual transaction execution is distributed across the network—they do not need to perform heavy computation themselves. This lowers the hardware barrier to entry, aiming for the widest possible decentralisation. The design goal is to make it profitable for anyone to propose blocks even with a modest stake, and to enable other network participation without any stake requirement.

Through experimentation, the network parameters will be tuned so that running a full node on hardware as simple as a Raspberry Pi remains feasible. This approach eliminates the need to delegate stake and thus promotes deeper decentralisation.

By employing a pure proof-of-stake mechanism, the protocol also mitigates the risk of transaction censorship. Since the pool of validators is randomly chosen for each block, only the block proposer can decide to omit certain transactions. For validators to collectively block a transaction, they would have to reject the block proposal outright, forfeiting their reward opportunity. On the next proposal round, a different proposer and validator set will be chosen. This significantly reduces any practical ability to censor transactions.

Constants

- **`HighWatermark`** - A block-size (or transaction-count) threshold. When a ***shard*** regularly exceeds this threshold, it signals that the shard may be overloaded and should consider splitting into two smaller ***shards***.
- **`EnoughValidatorsPerShard`** - The minimum required number of validators needed to sustain block creation in a new ***shard*** after splitting. If a shard cannot guarantee at least this many validators, it will not be allowed to split.
- **`ShardDivisionDelay`** - The number of ***Root***-level blocks to wait after deciding a ***shard*** should split. This delay gives validators and the network enough time to prepare for the creation and assignment of the new ***shards***.
- **`MaxPrefixLengthPerLevel`** - A limit on how many “leading bits” of an address can be used to form a ***shard*** prefix at each hierarchical level. Prevents the ***Root*** shard (and any ***SubRoot*** shards) from becoming overly large and ensures the network remains manageable.
- **`LowWatermark`** - A lower limit on block size. If two adjacent shards routinely produce blocks smaller than this threshold, it indicates underuse, triggering a merge procedure to combine those ***shards***.
- **`LowValidatorsPerLevel`** - The minimum acceptable count of validators in any ***shard***. If a ***shard’s*** validator count dips below this number, the shard becomes a candidate for merging

with another shard.

- **`ShardingConcatentionDelay`** - The number of ***Root***-level blocks to wait after deciding that two shards should merge. This delay period allows validators to reorganize and transition to the merged ***shard***.
- **`MinValidatorsPerLevel`** - The absolute minimum number of validators required to produce blocks in a ***shard***. If a ***shard's*** validator set falls below this number, it can no longer create blocks and must be merged.
- **`ValidatorEnterDelay`** - The waiting period (in ***Root***-level blocks) after a validator locks their stake and signals intent to join, before they become eligible to participate in block production and voting in their assigned shard.
- **`HigherLevelTransactionCostShare`** - The percentage (or proportion) of transaction fees that flows upward (e.g., from lower-level shards) to validators in higher-level shards, including the ***Root***.
- **`LowerLevelBlockPrizeShare`** - The fraction of the overall block reward allocated to lower-level ***shards***. This ensures that ***shard*** validators, who do most of the transaction-ordering work at their level, receive a fair share of the reward.
- **`ProposerShare`** - The percentage of fees (and possibly block rewards) that goes specifically to the validator who successfully proposes a block. The rest of the reward is divided among the other validators who certify or vote for the block.
- **`MaxValueLength`** - Defines the maximum size (in bytes) allowed for any single stored value in the ***Address*** -> Key -> Value ***(K->K->V) database***. Ensures data entries do not grow unbounded.
- **`MaxKeyLength`** - Defines the maximum size (in bytes) allowed for a database key. Prevents excessively large keys that could degrade performance or complicate storage.

Terminology

Address - A unique identifier associated with an account or a smart contract. In this design, addresses are used as the first key in the **K->K->V** database and help determine the shard (based on leading bits).

Block - A batch of transactions (or references to transactions) proposed and validated by a set of nodes (validators). Once certified, a block is added to the chain, forming part of the immutable ledger. In the ***Root*** or ***SubRoot*** shard, a block may also include references to blocks produced in lower-level shards, thereby finalizing their transaction order on a global scale.

Block Finality - A point in the consensus process where a block becomes effectively irreversible (under normal conditions). Once a block achieves finality, its contents—particularly the transaction order—are deemed permanent. Because that order is now fixed, the results of those transactions are already *deterministically guaranteed*, even if their full execution has not yet been performed.

Composability - The ability of one smart contract to directly call or interact with another in real time, potentially chaining multiple interactions. In sharded architectures, composability may be partially asynchronous, adding complexity to contract design.

Consensus Mechanism - The protocol by which validators agree on the next valid block (and thus on the ordering of transactions). In this paper, a variant of **Pure Proof of Stake** is used.

Distributed Execution - A process where the actual "work" of computing transaction outcomes happens across multiple nodes. Rather than forcing every node to execute every transaction, nodes can selectively execute subsets of transactions or rely on external sources.

Global State - All account balances, contract variables, and other on-chain data at a given point in time. In some blockchains, this entire state is updated via consensus. In this design, the global state can be **reconstructed** from transactions but is not necessarily required to be stored/updated by every validator in real time.

K->K->V Database (Key -> Key -> Value)*** - A scalable data structure used here as ***Address*** -> Key -> Value. It allows storing and retrieving data in a hierarchical way. Smart contracts and accounts are identified by the first key (address), and each address can maintain its own sub-keys and values.

Root (or Root Shard / Root Chain) - The top-level shard in the multi-level sharding hierarchy. It finalizes the block hashes produced in lower-level ***shards***, providing a global ordering for all transactions network-wide.

Shard - A sub-network handling a subset of addresses or transactions. Each ***shard*** runs its own consensus process to order transactions assigned to it. ***Shards*** ultimately feed their block references into the higher-level "Root" for global finality.

SubRoot - A ***shard*** at an intermediate level in the hierarchy, which may coordinate further "child" ***shards***. A ***SubRoot*** is structurally similar to the Root but exists at a lower tier in the tree-like

network.

VRF - Verifiable Random Function A cryptographic function producing pseudo-random outputs alongside a proof that can be publicly verified. Used in this protocol to fairly select block proposers and voting committees without requiring global coordination.

Future work

This paper introduces the design of a scalable ***consensus mechanism*** for a blockchain capable of handling an unlimited number of transactions on smart contracts. Future research and development will focus on the following areas:

Further Theoretical Analysis – Formalizing the consensus algorithm and providing its mathematical specification. Conducting detailed security analyses, including resistance to Sybil attacks, validator-level manipulations, and network partitioning (*network split*).

Simulation and Modeling – Evaluating system performance under various network conditions and different validator set sizes. Assessing the effects of network partitioning and determining the optimal values for system parameters. Testing resilience against various attack vectors and analyzing potential full-node manipulation scenarios.

Virtual Machine for Smart Contracts – Developing and publishing a detailed specification of a distributed virtual machine for smart contract execution. This includes designing an efficient algorithm for processing interconnected smart contracts while ensuring high performance and security.

Formal Academic Publication – Publishing research findings in scientific journals and presenting results at academic conferences.

Wood, Gavin. “Polkadot: Vision for a Heterogeneous Multi-Chain Framework.” Whitepaper (2016). Available at: https://polkadot.com/papers/Polkadot-whitepaper.pdf ↩

NEAR Protocol. “NEAR White Paper: Nightshade Sharding Design.” (2020). Available at: https://near.org/papers ↩

Elrond. “Elrond: A Highly Scalable Public Blockchain via Adaptive State Sharding and Secure Proof of Stake.” Whitepaper (2019). Available at: https://multiversx.com/ ↩

Buterin, Vitalik, et al. “An Incomplete Guide to Rollups.” (2021). Available at: https://vitalik.ca/general/2021/01/05/rollup.html ↩

Avinash Lakshman, Prashant Malik - Cassandra - A Decentralized Structured Storage System - https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf ↩

Micali, Silvio. “Algorand: The Pure Proof-of-Stake Blockchain.” Whitepaper (2017). Available at: https://algorand.co/blog/the-algorand-whitepaper ↩