KONTOR: A NEXT-GENERATION BITCOIN METAPROTOCOL

January 10, 2026

ADAM KRELLENSTEIN adam@kontor.network

WILFRED DENTON wilfred@kontor.network

OUZIEL SLAMA ouziel@kontor.network

ALEXEY GRIBOV alexey@kontor.network

Contents

1 Introduction

Bitcoin, by design, has a number of significant limitations which make it ill-suited for many use cases beyond that of a global payment network and settlement layer for digital gold:

1. Transaction confirmations on Bitcoin are determined by the 10-minute block time.
2. Bitcoin's maximum theoretical throughput is limited to around 9 transactions per second because of the 4 MB block limit and the repeated signatures and public key hashes required under both ECDSA and Schnorr.
3. Smart contracts on Bitcoin are limited to what can be expressed in Script, which is stateless and not Turing-complete.
4. Bitcoin cannot efficiently persist significant quantities of data because of the 4 MB block size limit.
5. Bitcoin's tokenomics do not support a gas model for smart contracts; BTC is ultimately deflationary, and fees are based on transaction weight only.

Immense economic pressure to unlock the value of the Bitcoin network has produced numerous projects and protocols that have attempted to grow the capabilities of Bitcoin, with mixed success and doubtful prospects. Going back to the invention of sidechains,[1] the predominant strategy has been to scale Bitcoin *horizontally*, as if Bitcoin were a database. But the limit to Bitcoin's scalability isn't in the rate at which it can write transactions to

disk; it's the rate at which the whole network can come to consensus on *which transactions to persist*, so each new blockchain network introduces new bottlenecks in state synchronization.

Sidechains and ZK-rollups, while often marketed as ways of scaling Bitcoin, are best understood as separate execution systems with their own consensus and security assumptions. They can add functionality and throughput, but they also shift economic activity and security-critical execution away from Bitcoin. More generally, blockchains are naturally self-contained ecosystems and economic markets; liquidity and developer attention tend to fragment across execution venues. The ethos behind Bitcoin Maximalism is rooted in the recognition of this fact.

The alternative is to scale Bitcoin *vertically*, by building directly on top of Bitcoin. This is the only way to grow the Bitcoin ecosystem without compromising its security model or competing with Bitcoin as a store of value. The **metaprotocol architecture** extends Bitcoin's state machine with additional logic: data in the blockchain that are ignored by Bitcoin nodes are parsed by the metaprotocol implementation and additional state is derived deterministically. This takes the architectural pattern of a blockchain to its logical conclusion, treating the Bitcoin blockchain as a log-structured store for protocol messages. Metaprotocols are fundamentally Bitcoin-native: all transaction data is on Bitcoin, no new op codes or additional security assumptions are required, and it's natural to use existing Bitcoin wallets and addresses. Unlike sidechains and rollups, metaprotocols strengthen Bitcoin by increasing on-chain activity and miner revenue. In fact, the design for metaprotocols was first described by Hal Finney in 2010.[2] The very first versions of Ethereum were metaprotocols on Bitcoin.[3]

Because metaprotocols are Bitcoin-native, they have enjoyed widespread adoption and are now an integral part of the Bitcoin ecosystem. Ordinals inscriptions exceeded 100 million by mid-2025[4], and the Runes metaprotocol accounted for a majority of Bitcoin transactions in the period following its launch.[5] Integral to the success of these projects is the fact that they do not require an explicit onboarding process.

1.1 Security model and threat model

A Bitcoin metaprotocol inherits Bitcoin's security model for data availability and transaction finality: every metaprotocol transaction maps to a Bitcoin transaction, and the authoritative transaction log is secured by Bitcoin consensus. The primary additional trust surface is software: metaprotocol users must run an implementation that deterministically interprets the Bitcoin log.

Canonical interpretation. Kontor indexers derive state deterministically from (1) the Bitcoin transaction log and (2) a versioned ruleset. If someone runs non-compliant Kontor software, they derive a different *metaprotocol* state from the same Bitcoin log. This is a metaprotocol software fork, not a Bitcoin consensus fork: Bitcoin finality still determines the authoritative transaction log, but metaprotocol users must coordinate on a shared interpretation of that log.

Threat model (high level). The statements in this paper assume:

- Bitcoin consensus determines the canonical transaction log (reorg risk is bounded by chosen confirmation depth).
- Kontor indexers apply the same ruleset version when interpreting that log.
- Participants may be adversarial; safety is enforced either by Bitcoin Script (on-chain enforcement), or by economically secured mechanisms with explicit residual risks.

In scope:

- Invalid metaprotocol messages included on Bitcoin (deterministic rejection by indexers).
- Double-spends and conflicting Bitcoin transactions (handled via ordering rollback rules).
- Storage non-compliance (detected by probabilistic challenges; penalized via slashing).

Out of scope:

- Software monoculture failures (a bug in widely used indexer software).
- Global Bitcoin censorship or catastrophic consensus failure.

In 2025, the question is no longer *whether* Bitcoin will be used for more than just payments. The question is, What does the future of Bitcoin look like? Will it be helpful or harmful to the ecosystem? How will DeFi on Bitcoin be different from DeFi on Ethereum and Solana?

Kontor is a next-generation Bitcoin metaprotocol with the following distinctive capabilities and a native cryptocurrency, KOR:

- **High Throughput:** Kontor achieves over 1,000 transactions per second using BLS signature aggregation, an address ID registry, WAVE binary encoding, and payload compression.[1] (Section 2.1.1.)
- **Low-Latency Confirmations:** Kontor Optimistic Consensus[7] allows for sub-block transaction confirmations—on the order of one to two seconds—while relying on Bitcoin mining for transaction finality.[2] (Section 2.2.)
- **Rich Smart Contracts:** Sigil is a next-generation WebAssembly-based smart contract framework that allows developers to write smart contracts that run directly on the Bitcoin network. The Sigil contract framework uses the Wasm Component Model to make writing smart contracts feel just like writing regular software. (Section 2.3.)
- **Scalable File Storage:** Kontor provides Bitcoin users with scalable, permanent, off-chain file storage where the data storage system itself is entirely on-chain. Users pay once, and the network incentivizes ongoing storage in perpetuity. (Section 2.4.)

These features of Kontor work together to enable a new generation of DeFi applications on Bitcoin, fully respecting Bitcoin's primacy as a store of value and unit of account; scaling Bitcoin's performance and functionality while preserving all of its security guarantees.

2 Protocol

2.1 Transaction Encoding

Kontor transactions are embedded in one or more Bitcoin transactions that, when confirmed in the Bitcoin blockchain, are parsed by Kontor indexers and trigger the execution of smart contract code in a sandboxed, deterministic virtual machine. The smart contract system roughly follows the Ethereum model: users pay "gas" fees in the KOR currency in proportion to the calculated load that this transaction places on Kontor indexers, which must parse the transaction. (As with Ethereum, transaction execution is "optimistic"; if the transaction

[1] Performance figures are derived from benchmarking against the reference implementation and are parameterization-dependent. See the Kontor Scalability specification[6] and the repository benchmarks.

[2] Latency depends on network conditions and the staker network's participation and message propagation.

execution requires more gas than is allotted, the execution is simply halted and the side-effects are rolled back.)

Kontor state transitions are defined entirely by the combination of the Bitcoin log and a versioned interpretation of Kontor payloads. Indexers are required to apply the same ruleset version to preserve a single canonical metaprotocol state; ruleset upgrades are consensus changes for Kontor and must be activated at an agreed point (e.g., a Bitcoin block height), analogous to a software upgrade rather than a Bitcoin consensus change.

Kontor encodes its protocol messages in Bitcoin transactions principally using the Taproot commit-reveal pattern that has also been adopted by Ordinals Inscriptions, Counterparty, and other Bitcoin metaprotocols. This method uses unexecuted script "envelopes" (`OP_FALSE OP_IF ... OP_ENDIF`) for embedding data entirely within Bitcoin's witness, [8] allowing for storing up to ~400 kB of data with the witness discount in a transaction that will be relayed according to the default Bitcoin standardness rules. [9] The Kontor SDK abstracts away the complexities of this data encoding method, returning to the user two (or more) chained Bitcoin transactions that may be signed and broadcast together.

The first Bitcoin transaction is the **Commit** transaction, with a unique Pay-to-Taproot (P2TR) ScriptPubKey and a tweaked public key, which is derived from the user's public key and the Merkle root of the Tapscript in the subsequent **Reveal** transaction:

```
/// ScriptPubKey

OP_1
OP_PUSHBYTES_32
<tweaked public key>
```

The Reveal transaction spends the relevant output of the Commit transaction using a Taproot Script Path Spend. The script begins with an `OP_CHECKSIG` which consumes a signature from the stack, verifies it against the 32-byte x-only public key (used in Schnorr signatures), and pushes a true (`1`) or false (`0`) to the stack based on the signature's validity, determining if the input is correctly spent. Next, the Taproot Envelope embeds the actual data in the Reveal transaction in such a way that the data are completely ignored during evaluation by the Bitcoin protocol.

```
/// Tapscript

// Signature Verification
<x-only public key>         // user identity
OP_CHECKSIG

// Taproot Envelope
OP_FALSE
OP_IF
  OP_PUSHBYTES b"kor"       // identifies Kontor metaprotocol
  OP_0
  OP_PUSHBYTES <serialized data part 1> // max 520 bytes
  OP_PUSHBYTES <serialized data part 2> // max 520 bytes
```

```
  ...
OP_ENDIF
```

During block parsing, each Kontor indexer detects these scripts efficiently through pattern matching: a node first checks and extracts the x-only public key [10], [11] that represents the user's identity; it then verifies that each operation matches until `OP_0`. Finally, it extracts all data until it encounters an `OP_ENDIF`. Kontor message data, embedded in the witness data of the Taproot Reveal transaction, is serialized using Postcard. [12]

There are three principal types of Kontor transactions, all derived from the above scheme:

1. **`Call`** A contract call is represented by a commit-reveal transaction pair using Taproot Envelope to serialize the contract call payload. The funding input's change flows to the Commit transaction output's value, which then covers the Reveal transaction's fee. These two transactions are then published together as a unit.

2. **`Attach`** Attaching an asset balance to a UTXO follows the same composition as a `call`, but it includes an additional chained `detach` Commit whose ScriptPubKey receives the asset balance transfer. For an `attach` operation to be valid, Kontor requires a chained `detach` commit in the reveal transaction. Kontor can recreate the `detach` payload from the x-only public key and asset balance arguments. This allows it then to regenerate the ScriptPubKey of the detach commit and verify that it matches, which protects both the user performing the `attach`—ensuring they can detach their asset balance—and the buyers in swap transactions who want to purchase the asset balance for BTC.

3. **`Detach`** The `detach` operation is a Reveal of the `attach`'s `detach` commit. It optionally includes an `OP_RETURN` output containing the x-only public key where the asset balance should be sent, which is used for swaps. If no `OP_RETURN` is included, then the asset balance returns to the attacher's x-only public key.

Atomic swaps between Kontor assets and BTC (or other UTXO-based assets) are constructed by combining `attach`/`detach` with partially signed Bitcoin transactions (PSBTs). [13] With this feature, Kontor offers deep, native integration with Bitcoin and other Bitcoin metaprotocols.[3] This enables non-custodial, single-confirmation, atomic swaps between Kontor assets and both Bitcoin and Bitcoin UTXO–based metaprotocols, including Ordinals, Runes and Counterparty.

2.1.1 Scalability

The metaprotocol architecture inherits Bitcoin's binding constraint: a block weight limit of 4,000,000 weight units, with blocks mined approximately every ten minutes. A naïve encoding—one Bitcoin transaction per Kontor operation—yields throughput comparable to vanilla Bitcoin payments: roughly 8–10 transactions per second. The fixed overhead of Bitcoin transaction structure (inputs, outputs, witness data, signatures) dominates; the Kontor payload itself is a minority of total weight. The optimizations described here allow Kontor to achieve over 1,000 transactions per second via BLS signature aggregation, compact registry IDs, binary encoding, and Zstd compression.[4]

[3] The authors of this paper notably implemented a similar protocol for Counterparty in October, 2024.

[4] See the benchmarking methodology and parameter choices in the Kontor Scalability specification[6] and the repository benchmarks.

Batching multiple Kontor operations into a single Bitcoin transaction amortizes fixed overhead, but each operation still needs an authorization from its signer. Without aggregation, you would include N per-operation signatures in the payload—negating much of the benefit. BLS (Boneh–Lynn–Shacham) signatures solve this problem at the metaprotocol layer. The key property of BLS is that N signatures on N distinct messages combine into a single 48-byte aggregate, verifiable in one operation. Users sign their Kontor operations with BLS private keys (derived from their seed phrases via a Kontor-specific derivation path); an aggregator combines these into one aggregate signature and broadcasts a single Bitcoin transaction containing all N operations plus the aggregate signature. Bitcoin does not validate the BLS signature; Kontor indexers do. Aggregation is trustless: any party with access to the signed operations can produce a valid aggregate. There is no privileged aggregator role; any node can aggregate signed operations and broadcast a batch. Aggregators compete to bundle operations efficiently; users benefit from lower per-operation fees.

With signature aggregation enabling efficient batching, additional optimizations reduce payload size further:

- **Registry**: A deterministic mapping from long identifiers (64-character hex public keys, contract addresses) to compact 4-byte numeric IDs. When a new public key first signs a Kontor operation, the indexer automatically assigns it the next sequential ID. This reduces identifier overhead by 16×.
- **Serialization**: Kontor uses Postcard to serialize each operation into a compact struct: signer ID, contract ID, function index, and arguments. The WIT/WAVE type system defines the schema; Postcard is the uniform binary encoding for all on-chain payloads.
- **Compression**: After serialization, Zstd compression exploits redundancy across batched operations. Repeated contract IDs, common function indices, and similar argument patterns compress well.

At a batch size of 100 operations, the full optimization stack achieves 15–18 weight units per operation, yielding 366–443 transactions per second (35–42× improvement over vanilla Bitcoin). At batch size 1,000, throughput reaches 683–986 TPS (65–93×); at batch size 10,000, throughput reaches 771–1,295 TPS (73–123×).[5] These figures position Kontor to support sophisticated DeFi applications—order books, AMMs, lending protocols—at throughput levels competitive with dedicated Layer-1 chains, while maintaining the security and finality guarantees of Bitcoin settlement.

The optimizations are opt-in: users who want to minimize fees can participate in batched transactions; users who prefer simplicity or need immediate confirmation can submit individual transactions. The full specification is available in the Kontor Scalability specification. [6]

2.2 Optimistic Consensus

Bitcoin blocks are mined approximately every ten minutes. Users who require faster confirmation than Bitcoin provides—such as those making payments at point of sale or trading in fast-moving markets—historically have had to choose between trusting a centralized intermediary, re-using established payment channels with liquidity, or simply waiting.

Kontor Optimistic Consensus[7] offers an alternative: sub-block transaction confirmation with latency on the order of one to two seconds, while preserving Bitcoin's position as

[5]These figures are derived from benchmarking against the production Kontor codebase.

the ultimate source of truth for transaction finality. The mechanism is a virtual prediction market: KOR holders stake tokens to participate in a BFT consensus protocol that produces signed batches of transactions, assigning deterministic positions to Kontor transactions before Bitcoin confirmation. When stakers sign a batch containing a transaction, they are asserting that the transaction is valid, that its fee is sufficient for timely confirmation, and that no conflicting transaction will confirm on Bitcoin. A batch is valid only when signed by stakers representing more than two-thirds of total stake. Users can accept optimistic confirmations upon receiving a signed batch—the stakers' capital backs the guarantee. If Bitcoin fails to confirm a transaction that has been batched, the transactions are rolled back as if a blockchain reorganization had occurred.

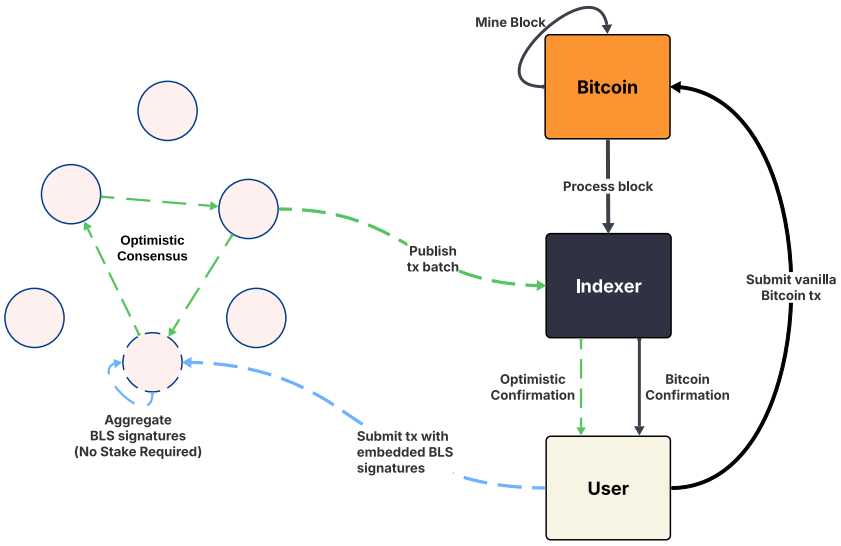

FIGURE 1. Optimistic consensus network flow (high level).

Bitcoin remains authoritative. When a batched transaction confirms on Bitcoin, it is **batch-confirmed**: it executes at its assigned position in the global ordering. If, however, a conflicting transaction confirms instead—whether due to a double-spend attempt or network conditions—the batched transaction and all subsequent transactions in the ordering are rolled back. State changes are reversed, and the system proceeds from the authoritative Bitcoin state.

Batches form a single ordered log: the staker network continuously produces a sequence of signed batches, often faster than Bitcoin finalizes earlier positions. As a result, users may receive confirmations for transactions in later batches while earlier batches are still unresolved on Bitcoin. These later batch proofs still attest to ordering, but their economic meaning is conditional: Kontor executes transactions strictly in position order, and a rollback at an earlier position (caused by a conflicting Bitcoin-confirmed transaction) cascades and invalidates all subsequent positions.

This cascading rollback is the key tradeoff: optimistic confirmations are probabilistic guarantees backed by staker capital, not the absolute finality of Bitcoin confirmation. Users who cannot tolerate rollback risk should wait for Bitcoin confirmation; users who value speed can accept optimistic confirmations with the understanding that stakers are economically liable for conflicts. This strictly dominates vanilla Nakamoto pre-confirmation as an acceptance signal: for stateful metaprotocols the within-block order is arbitrary anyway, so an

explicit ordering rule is required, and batches provide a deterministic ordering *before* Bitcoin confirmation. Economic finality is market-priced: each transaction specifies a minimum total bonded stake requirement $K_{\min}(t)$, and stakers are paid for that transaction only when the enclosing batch's total bond meets the requirement. Bitcoin already admits reorganizations and conflict confirmations; optimistic consensus makes pre-confirmation assurance explicit, measurable, and economically accountable.

2.2.1 Evidence and slashing (high level)

In Kontor, slashable offenses are those for which any observer can later produce unambiguous evidence from signed batch data and Bitcoin-confirmed transactions.[7]

Bitcoin's Replace-By-Fee (RBF) mechanism presents an attack vector: a user could submit a transaction, receive optimistic confirmation, then broadcast an RBF replacement. Kontor mitigates this by treating batched UTXOs as spent: indexers reject any Kontor transaction that spends inputs already claimed by a pending batch. If a user broadcasts an RBF replacement on Bitcoin after batching, the batched transaction expires and the user loses their ordering fee (KOR). Stakers are not slashed in this case if they can submit cryptographic evidence of the RBF replacement on Bitcoin; conflict slashing applies only when a conflicting transaction confirms without such an RBF explanation.

Staker behavior is governed by economic incentives. Honest stakers earn a share of protocol emissions (e.g., 10% of total KOR emissions) plus ordering fees (KOR) from batched transactions. Dishonest or negligent behavior triggers slashing:

- **Equivocation**—signing conflicting batches—results in complete stake loss. This is provable malice: cryptographic evidence (two signatures from the same key on conflicting batches) triggers immediate and total slashing.
- **Conflict confirmation**—when a conflicting transaction confirms on Bitcoin instead of the batched one and stakers cannot demonstrate that the batch transaction was displaced by an external RBF replacement—results in partial slashing.
- **Batch expiry**—when batched transactions fail to confirm within the expiry window—results in graduated penalties that scale with the number of expired transactions.

Slashed funds are split between burning (reducing supply) and rewarding the evidence submitter (incentivizing monitoring).

If staker ordering fails entirely—whether due to network partition, coordinated attack, or insufficient participation—the protocol degrades gracefully. Users can still transact by broadcasting directly to Bitcoin; their transactions confirm on-chain and are appended at "block-end" after all batched transactions. The system never blocks Bitcoin activity. Optimistic ordering is a performance enhancement, not a requirement for operation.

2.3 Sigil Smart Contracts

Sigil is a *next-generation WebAssembly-based framework* that enables a user-driven ecosystem where developers can publish their own contracts for execution on Bitcoin with Kontor. The functionality of the Kontor protocol is then determined dynamically by the publication of these smart contract code modules. All inputs to the smart contracts are pulled from the Bitcoin ledger, and once a smart contract code module has been published to a given network, its execution across that network may be triggered by any member of the network with the broadcast of an appropriately signed transaction with funds allocated for the gas costs. Sigil offers stronger compile-time type safety, powerful language built-ins, an expressive

storage interface, rich native types, native cross-contract calls, and better ergonomics in general than other smart contract languages.

2.3.1 Runtime

WebAssembly is an increasingly popular runtime for smart contract platforms because it provides a secure, sandboxed environment for contract execution, as well as superior performance and easy integration of a gas metering system with Wasmtime. However, the naïve adoption of the WebAssembly runtime leads other frameworks that target WebAssembly to treat contracts as standalone Wasm modules that are isolated from the host and from each other. This in turn necessitates the introduction of a custom Application Binary Interface (ABI) such as CosmWasm's `cosmwasm_std` and Substrate's `ink_env`, which are often specified in JSON and cumbersome to develop and use. CosmWasm defines JSON messages with a prefix string, NEAR uses JSON/Borsh for input/output and state, and Alkanes uses numeric opcodes in Runestone-style cellpacks to select actions and forward raw inputs to a contract. [14] These broad host environments entangle application logic with runtime details and enlarge the VM–contract boundary that must be reviewed and versioned. Most importantly, it disallows all runtime linking and requires developers to manually manage dependencies. By contrast, Sigil adopts the WebAssembly Component Model end-to-end, [15] describing all contract interfaces using WebAssembly Interface Types (WIT), a language-agnostic IDL that enables typed, standardized function signatures across contract modules and with its own runtime. [16], [17] Sigil deliberately constrains the host surface area to three primitive capabilities—signer access, cross-contract calls, and storage operations—while lifting higher-level facilities (authorization, ORMs, governance) into libraries. This narrow host API reduces the trusted computing base, simplifies determinism analysis, and limits the attack surface, enabling faster iteration in userspace without compromising runtime stability. Many smart contract bugs are serialization/ABI mismatches (wrong type, wrong selector, wrong encoding)—Sigil's use of WIT and typed interfaces helps catch these integration bugs at compile time.

Along the same lines, Sigil uses WIT and WAVE (WebAssembly Value Encoding) to describe the types of all values that cross component boundaries, and Postcard as the concrete binary encoding, allowing complex Rust structs and enums to be encoded and decoded in a type-safe manner. Other Wasm-based platforms use a variety of serialization formats that do not preserve the necessary type annotations (NEAR uses JSON for function I/O and Borsh for contract state; [18] Polkadot uses the SCALE codec; [19] CosmWasm uses JSON messages; [20] EOSIO uses a custom binary format; Ethereum uses Contract ABI encoding with a 4-byte function selector for contract calls). [21] These formats lack a shared schema between the contract and the host, requiring the caller and callee to agree on the data format by convention. Kontor's use of WIT and WAVE means the data schema is explicitly shared as part of the interface, which can catch type mismatches at interface boundaries. Each Sigil contract includes both a WIT interface and an implementation of that interface. At a high level, WIT serves as the interface description language—defining types and function signatures for contracts and the host—while Postcard provides the schema-aware binary encoding used for arguments, return values, and on-chain payloads. The combination ensures that every value crossing a component boundary is both typed (by WIT) and compactly serialized (by Postcard).

2.3.2 Contract Structure

Contracts can be written in any language targeting WebAssembly, but the Sigil SDK offers first-class Rust support via a `contract!` macro that generates WIT bindings and wires up component model interfaces at compile time.

2.3.2.1 Execution Contexts

Sigil encodes execution semantics directly in the types of exported functions. Every exported function receives a **context** as its first parameter, and the context type determines what operations are permitted:

- **`ProcContext`** ("procedure"): Enables state-modifying operations. Provides `signer()` to identify the transaction sender, `contract_signer()` for the contract's own identity (used for custodial patterns where contracts hold assets on behalf of users), and full read-write access to storage. Used for functions like `mint` or `transfer` that update blockchain state.
- **`ViewContext`**: Supports read-only queries for inspecting contract state. Restricts access to read-only storage operations—no signer access, no mutations. Used for functions like `balance` that retrieve data without modifying the blockchain. These functions are callable via the API exposed by Kontor nodes.
- **`FallContext`**: Used exclusively for the `fallback` hook. Can be converted to `ViewContext` for read-only access, or to `Option<ProcContext>` when a signer is present. Enables proxy patterns and generic delegation.

These distinctions are visible in the WIT interface and enforced at both compile time and runtime, preventing accidental mutation along read-only paths and clarifying the call graph for tools and auditors. Additionally, an `init` hook is called once when a contract is first deployed for initialization.

2.3.2.2 Storage

Sigil's storage system utilizes a hierarchical key space with path-based accessors that provide compile-time type checking for stored data, with fine-grained access to nested fields that allows contracts to read or update a sub-field without materializing an entire aggregate. Contracts define their storage structure using derive macros:

- **`#[derive(StorageRoot)]`**: Marks the root storage type. Every contract has exactly one `StorageRoot`, which provides an `.init(ctx)` method and is accessible via `ctx.model()`.
- **`#[derive(Storage)]`**: Used for nested structures within the storage hierarchy.
- **`Map<K, V>`**: A key-value collection type for storing mappings (e.g., account balances).

The resource model closely tracks the operations performed: updating an allowance within a nested map touches only the relevant path and charges gas accordingly.

2.3.2.3 Native Types

Sigil provides high-precision numeric types for financial calculations:

- **`Integer`**: 256-bit signed integers. Used for token balances, vote counts, and any value that might exceed 64 bits. Provides both unchecked operators (`+`, `-`, `*`, `/`) that panic on overflow and checked methods (`.add()`, `.sub()`, `.mul()`, `.div()`) that return `Result`.
- **`Decimal`**: High-precision decimals (up to 18 decimal places) for calculations requiring fractional values, such as price calculations or percentages.

2.3.3 Cross-Contract Calls

Sigil uses the WebAssembly Component Model to import foreign contracts via their WIT and to generate typed stubs, so cross-contract calls are checked by the compiler and read like ordinary library calls rather than RPC. When a dependency evolves—such as an enum variant gaining a field or a parameter changing from `u64` to `integer`—callers that have not updated will fail to build, rather than emitting transactions that revert or decode incorrectly on chain. Competing systems model cross-contract interactions as RPC-like message passing (CosmWasm JSON `Execute` messages, NEAR promises), where schema drift is discovered only at runtime; Substrate's ink! can generate strongly typed stubs, but these are not standardized across the chain and generally require both sides to share the same codebase rather than a chain-wide IDL. Sigil's component-level linking and WIT-based stubs make integrations explicit, portable across languages, and resilient to interface evolution, shrinking the cognitive gap between on-chain and off-chain development. [17], [20], [22], [23] Indeed, Sigil's use of Wasmtime locally makes it trivial to develop contracts in the exact same runtime as on-chain.

Sigil provides two macros for importing contract interfaces:

- **`interface!`**: Imports a contract's WIT interface without specifying its address. The contract address is passed as a parameter at runtime. This enables standards patterns similar to Ethereum's ERC-20: define an interface once, and any contract can work with any implementation of that interface. For example, an AMM contract can accept any token address that implements the transfer interface.
- **`import!`**: Imports a specific contract instance at a known address (block height and transaction index). Used when the dependency is fixed at compile time.

For DeFi applications, `interface!` is the more common pattern:

```
interface!(name = "token", path = "token/wit");

fn deposit(ctx: &ProcContext, token: ContractAddress, amount: Integer) ->
Result<(), Error> {
    // Transfer from user to contract's custodial address
    token::transfer(&token, ctx.signer(), &ctx.contract_signer(), amount)?;
    // ... update local state
    Ok(())
}
```

The `contract_signer()` method returns the contract's own identity, enabling custodial patterns where contracts hold assets on behalf of users—essential for AMMs, liquidity pools, and escrow.

2.3.4 Error Handling and Rollback

Sigil provides robust error handling through Rust's `Result` type combined with automatic storage rollback. When a function returns an `Err` or panics, **all storage modifications are rolled back**—not just for the current function, but for the entire call chain across cross-contract calls. If Contract A calls Contract B and B returns an error, all storage changes in both A and B are rolled back; the entire transaction has no effect.

This all-or-nothing semantics simplifies reasoning about contract behavior: a transaction either completes fully with all storage changes persisted, or fails entirely with no storage changes. Combined with Sigil's prohibition on re-entrancy (calling cycles between contracts are disallowed), developers can follow the Checks-Effects-Interactions pattern without the re-entrancy guard boilerplate common in EVM development.

2.3.5 Upgradeability

Sigil supports upgradeability through a `fallback` hook that routes unmatched calls to successor components while preserving strict storage isolation. Each version owns its own state, eliminating the `delegatecall`-style hazards common in EVM proxy patterns—layout coupling, clashing storage slots, and upgrade keys with broad authority. Upgrade authorization can be implemented in userspace, enabling governance mechanisms like DAO-based voting rather than relying on key-based primitives.

2.3.6 Deterministic Execution

Kontor runs contracts in Wasmtime with threading, SIMD, and floating-point non-determinism disabled, targeting `wasm32-unknown-unknown`. All timeouts and resource limits are deterministic, and all runtime primitives are audited for determinism directly.

2.4 File Storage

Storing data directly in Bitcoin transactions externalizes costs to the whole network and is still extremely expensive. Kontor establishes a protocol for cryptographically verifiable, high-availability data storage that complements the state-machine replication of Kontor and Bitcoin.[24] Files stored with Kontor are stored *forever*: the user pays once, and the network incentivizes ongoing storage in perpetuity.

Other decentralized storage projects—Filecoin, Sia, Chia—are designed to compete with centralized services like Amazon S3, [25] providing storage only as long as users continue paying fees. They offer no long-term availability guarantees and require separate blockchains with incompatible tooling. Arweave storage protocol is explicitly not intended to survive indefinitely, and Arweave nodes face no penalties for failing to store committed data. [26]

Kontor uses KOR to incentivize storage nodes to maintain user data indefinitely. The protocol implements a commitment-challenge-response mechanism to verify that nodes are actually storing the data they claim to store, with rewards for participation and economic penalties (slashing) for non-compliance. The cryptographic protocol itself is based on known schemes for compact proofs of retrievability. [27], [28] Importantly, the entire file storage protocol itself resides on-chain—only the static data is not embedded in Bitcoin transactions.

A foundational principle of the protocol is that the economic value of data is independent of its physical size. To align the protocol's incentives with this principle, rewards, user fees, and stake requirements are scaled based on the **natural logarithm** of the file size ($\ln\left(s_f^{\text{bytes}}\right)$), not its linear size. This ensures that the return on capital for storage nodes is roughly constant across file sizes, making both small and large files similarly attractive to store.

2.4.1 Protocol Description

In the Kontor data storage protocol, **users** upload their data to **storage nodes**, which commit to storing their data forever. To store a file, a user pays a one-time fee, which is calculated based on the file's size and the network's current state. This entire fee is burned, creating deflationary pressure.

An initial set of storage nodes are party to a file agreement upon its creation. These nodes are not paid from the user's fee; instead, they (and any nodes that join later) are compensated through ongoing KOR emissions. The protocol uses a pooled stake model: to participate, a node must maintain a single, total KOR stake balance that is sufficient to cover all of its file storage commitments.

With the mining of each Bitcoin block, every Kontor indexer deterministically derives from the block hash a **challenge** that pseudo-randomly identifies a set of previously uploaded files to be audited. For each challenged file, one of its storage nodes is selected to publish a **proof** to the Bitcoin blockchain that it is indeed storing the data. This proof must be submitted within a fixed window of blocks.

If a storage node fails to produce a valid proof in time, a portion of its staked KOR is slashed. A part of the slashed funds is burned, and the remainder is distributed to the other nodes storing that same file. Conversely, nodes in a file agreement are rewarded each block a share of that file's KOR emissions. After an agreement is created, nodes may join or leave it based on their operational costs and expected profits, as long as the file's replication level remains above a minimum threshold. Nodes pay a fee to leave based on (1) the quantity of KOR they escrowed to join the agreement and (2) the number of nodes in the agreement. Storage nodes are thus strongly incentivized to store all files that they have committed to.

The Kontor protocol uses Nova recursive SNARKs[29] for proof-of-retrievability, providing a fully transparent setup (no trusted ceremony). A storage node demonstrates possession of data by answering deterministic challenges against a Merkle tree commitment. Proof payloads are constant-size (around 12 kB) and verifier time is approximately 50 milliseconds in the reference parameterization with $s_{\text{chal}} = 100$; both depend only on the number of sampled symbols, not on the underlying file size.[6]

[6]Measured in the reference implementation and dependent on parameterization and hardware. See the Kontor Proof-of-Retrievability specification[28] and implementation notes in the repository.

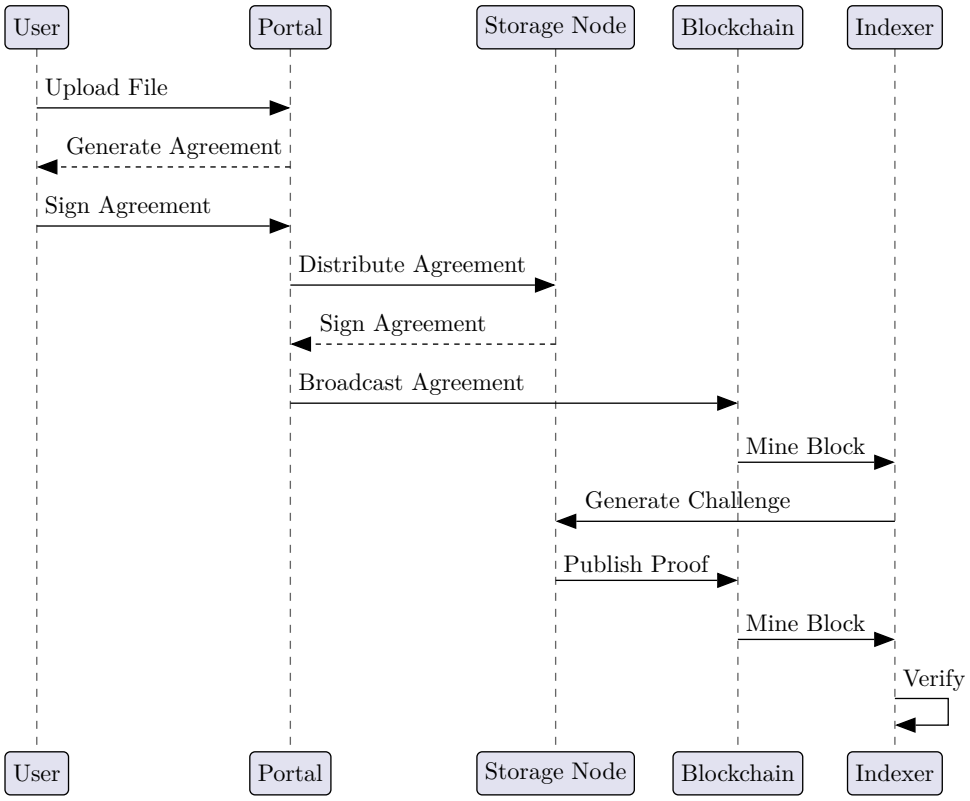

Figure 1: Kontor File Storage Protocol Communication

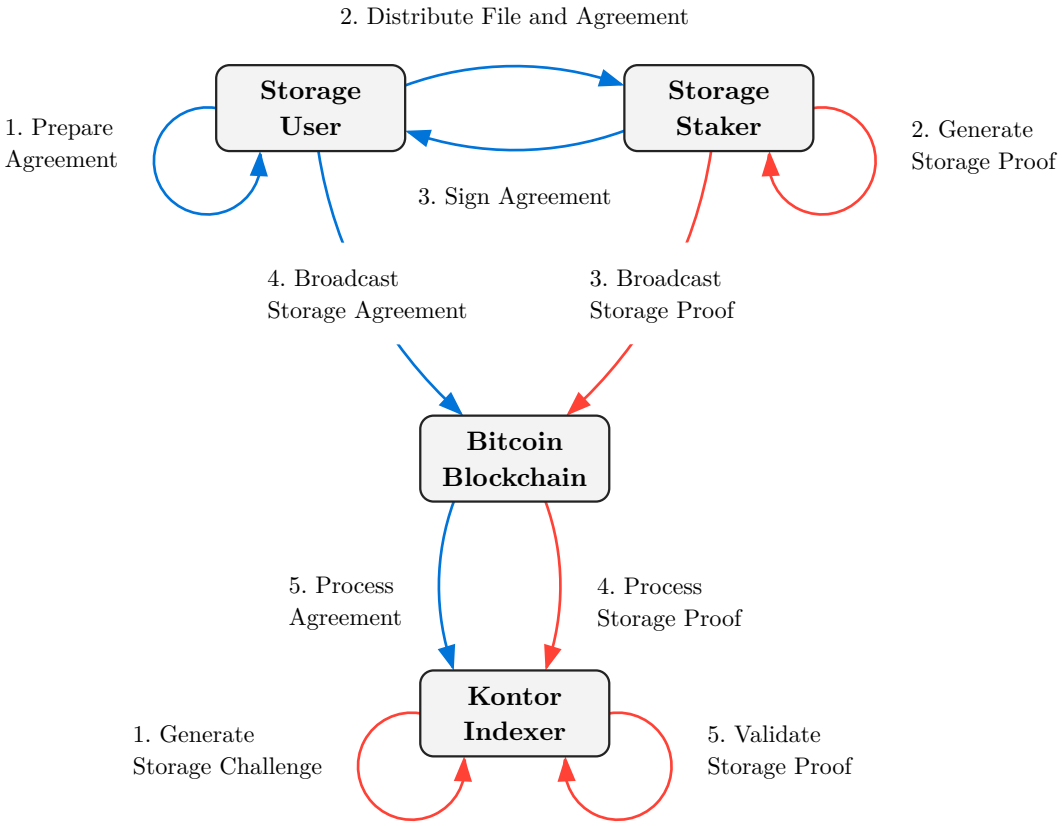

FIGURE 2. File storage flow: agreement creation, proof publication, and indexer challenge processing.

2.4.1.1 File Preparation

The user prepares files locally by partitioning data into fixed-size symbols, applying Reed-Solomon erasure coding (enabling reconstruction from ≥90% of symbols), and building a Poseidon Merkle tree. The Merkle root serves as a compact cryptographic commitment; each symbol is encoded directly as a field element, ensuring that provers cannot generate valid proofs without storing the actual data. The full preparation algorithm is specified in the storage protocol specification.[24]

2.4.1.2 File Agreement Creation

A file storage operation is memorialized in a `FileAgreement` Kontor transaction, signed by the user and an initial set of storage nodes. The user pays a one-time storage fee that is entirely burned. The file's perpetual emission weight, which determines its share of rewards, is based on its size and creation rank—files created when the network is smaller receive higher rewards in perpetuity.

The protocol uses a pooled stake model: each node maintains a single stake balance sufficient for all files it has committed to store. The required stake scales with the file's proportional weight and the network's size, amplified by a dynamic factor that disincentivizes Sybil attacks. A node can only join an agreement if its stake meets the new requirement. The

protocol ensures that the number of nodes storing any file never drops below a minimum threshold $n_{\min}$. Detailed formulas are provided in the storage protocol specification.[24]

2.4.1.3 Challenges

With each Bitcoin block, Kontor indexers deterministically derive challenges from the block hash. The number of files challenged scales to maintain a constant target challenge rate per file per year, ensuring consistent security guarantees as the network grows. For each challenged file, one storage node is randomly selected to produce a proof. The challenge consists of randomly selected Merkle leaf indices, derived deterministically from the block hash—unpredictable beforehand but publicly verifiable afterward.

The random sampling provides strong detection guarantees. If a fraction f of a file's symbols is missing and each challenge samples s_{chal} symbols uniformly at random, the probability of detecting the loss in a single challenge is $1 - (1 - f)^{s_{\text{chal}}}$; with 100 symbols sampled per challenge and $f = 0.10$, this exceeds 0.999. Complete data loss is detected within a year with near-certainty (>99.99%), and even partial losses at the erasure coding threshold are caught within months on average. The probability of detecting data loss depends only on the *fraction* of missing data, not file size, ensuring uniform security across all stored files.

2.4.1.4 Proofs and Proof Verification

Storage challenges must be answered within a predefined submission window. The challenged node generates a recursive SNARK proving knowledge of the challenged symbols and their Merkle paths, then compresses it for broadcast to Bitcoin. When included in a block, all indexers verify the proof against the file's Merkle root and challenge seed; invalid or missing proofs trigger immediate slashing.[28]

2.4.1.5 Slashing

If a storage node either lets a challenge expire or produces an invalid storage proof, then that storage node's stake k_n is slashed by an amount equal to $\lambda_{\text{slash}} \cdot k_{f(t)}$, where $k_{f(t)}$ is the dynamic base stake for the file in question and λ_{slash} is a system-wide multiplier. The node is also immediately removed from the file agreement.

A proportion of the slashed funds, β_{slash}, is burned. The remainder is distributed equally among the other storage nodes that are parties to the storage agreement that was broken. This disincentivizes a form of collusion in which only one storage node in the agreement actually stores the file and merely transfers the file data to other nodes that have committed to it when the latter are challenged.

If a slash (or any other event) causes a node's total stake k_n to fall below its required stake $k_{\text{req}}(n, t)$, the protocol automatically removes the node from file agreements with additional penalties until stake sufficiency is restored.

2.4.2 Storage Node Economics

Storage nodes are modeled as rational, profit-seeking agents. Revenue comes from KOR emissions and slashed funds; costs are dominated by the opportunity cost of staked KOR. This capital cost dominance is by design: when staking costs exceed physical storage costs, nodes compare KOR rewards against KOR staking costs—both intrinsic to the protocol. This eliminates the need for external price oracles and prevents the inflation death spirals that have plagued other token-based storage systems.

Files receive a fixed emission weight ω_f at creation time, based on their size and creation rank: $\omega_f = \frac{\ln\left(s_f^{\text{bytes}}\right)}{\ln(1+\text{rank}_f)}$. This creates a tapering effect: files created when the network is young receive higher reward shares in perpetuity, incentivizing early adoption while preventing runaway reward dilution as the network grows.

Market forces naturally stabilize replication: as more nodes join, per-node rewards decrease; as nodes leave, rewards increase. The pooled stake model provides Sybil resistance. Nodes can join file agreements directly or through **sponsored joins**, where an existing storer provides data in exchange for a temporary commission. Sponsorship uses a bond-escrow mechanism designed to prevent both sponsor extortion and entrant griefing. Departure is forbidden if replication is at the minimum threshold; otherwise, departing nodes pay a leave fee that scales as files become more vulnerable. Detailed formulas are provided in the Kontor Storage Protocol specification.[24]

2.4.3 File Retrieval

While the protocol guarantees storage, retrieval occurs off-chain through Bitcoin payment channels. Users discover candidate providers by querying indexer state for $\mathcal{N}_f$, negotiate price quotes and channel coordinates off-chain, select one or more providers, and establish payment channels on Bitcoin.

Retrieval proceeds via atomic symbol-for-payment exchanges: the provider streams symbols together with their Merkle paths; the user verifies each path against the on-chain root ρ and only then signs a payment channel update for that symbol. If verification fails, the user aborts and closes the channel with the current state.

The erasure coding structure solves the "final symbol problem": in any sequential exchange, one party must accept risk on the final transfer.[30] Users request $n_{\text{symbols}} + k$ symbols where $k \geq 1$ represents redundancy beyond the reconstruction threshold (e.g., 235 of the 231 required per codeword), so if a provider withholds the final symbols, the user can still reconstruct the file and the provider simply forfeits the withheld payment.

This mechanism is cryptographically verifiable: Merkle path checks ensure data validity, and redundancy makes completion incentive-compatible. File retrieval is a bilateral off-chain contract settled through Bitcoin payment channels; the Kontor protocol does not track or enforce retrieval on-chain. The full protocol specification is available in the Kontor Storage Protocol specification.[24]

3 Tokenomics

In the development of novel blockchain protocols, too often the tokenomics of the projects are only an afterthought. The emissions schedule of gas tokens especially is critical to the success not only of fundraising efforts but of the long-term success of a protocol, and, furthermore, gas tokens have economics that are fundamentally different from those of Bitcoin *qua* digital gold. In particular, there exist many ill-fated ongoing efforts to attempt to use novel tokens with Bitcoin-like economics—or even BTC itself—as gas for smart contract execution in one form or another. It is, in general, not very difficult to create a new blockchain network; it's much more difficult to create a healthy blockchain ecosystem that is supported by a virtuous cycle of shared value creation. Native currencies align interests across an entire network of disparate users, a minority of which cares about each particular feature of the protocol.

A native cryptocurrency is essential to the success of almost all blockchain protocols because it may be used to incentivize the good behavior amongst anonymous network participants. Additional digital currencies and protocols on the Bitcoin blockchain are not fundamentally dilutive, at least insofar as they don't compete with Bitcoin as digital gold and a unit of account for payments. On the contrary, additional protocols on a shared Bitcoin network support the value of Bitcoin in the long run, by the same mechanism that allows for increases in the value of the Bitcoin token to support the Bitcoin network.

3.1 Monetary Policy

Bitcoin's deflationary economics make it unsuitable as a gas token for smart contract execution. Additionally, incentivizing honest behavior among anonymous network participants requires a token whose supply the protocol controls. KOR, unlike BTC, is fundamentally inflationary.

Kontor distinguishes **issuance**, **fees**, and **burns**. Issuance mints new KOR; fees are denominated in KOR and are assessed on specific protocol actions; burns are the subset of fee and penalty flows that permanently reduce supply. Across the network, the protocol assesses the following fees:

- **Smart contract execution fees**, charged on Sigil contract calls as a function of metered computation (gas) and metered block space (data). This is the only utilization-priced fee class; its pricing is determined by network utilization and congestion.
- **Storage creation fees**, paid when a `FileAgreement` is created and entirely burned.
- **Storage departure fees**, paid by storage nodes that leave file agreements when replication permits.
- **Transaction aggregation fees**, paid to batchers as compensation for producing and broadcasting batched Bitcoin transactions.

Separately, the protocol burns KOR from the following sources:

- Smart contract execution fees (indexer fees), which are entirely burned.
- A fraction β_{fee} of ordering and bundling fees.
- One-time storage creation fees.
- A fraction of slashed stake, as specified by β_{slash} in the Slashing section.

This structure yields variable net inflation:

$$\text{net inflation} = \mu_0 - \text{burn rate} \tag{1}$$

Here "burn rate" denotes the annualized KOR supply reduction rate from burns (fee burns and slashing burns), expressed as a fraction of total supply.

Kontor's monetary policy emits KOR at a fixed annual rate (μ_0, e.g., 10%) relative to total supply:

$$\varepsilon(t) = \text{KOR}_{\text{total}}(t-1) \cdot \left(\frac{\mu_0}{B}\right) \tag{2}$$

where B is blocks per year. Emissions are split between storage $(1-\chi)$ and ordering χ: storage nodes receive emissions proportional to file emission weights; ordering stakers receive emissions plus the non-burned portion of ordering fees, proportional to stake bonded on successful batches.

The burn rate is driven primarily by smart contract activity. Let u be block utilization and let $\beta(u)$ be the congestion multiplier from the Transaction Fees section (near zero when

utilization is low, rising through the transition region). Burns scale roughly as $u \times \beta(u)$, and the system is parameterized to reach break-even (burns equal emissions) near 80% utilization. As activity increases, burns naturally rise and net inflation drops.

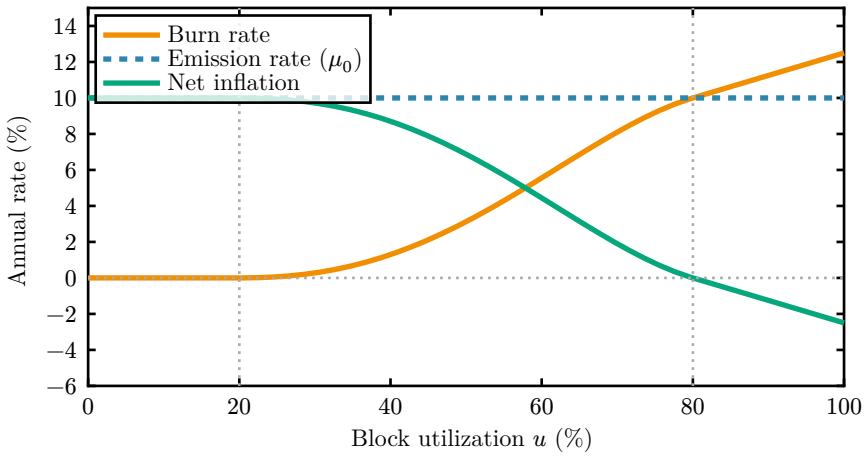

Figure 2: Burns and net inflation vs. block utilization. Below 20% utilization, inflation is the full 10%. Above 80%, the network becomes deflationary.

3.2 Transaction Fees

Kontor transactions incur up to three independent fee components:

Fee	**Settlement**	**Zero?**
f_{ix}	Burned (spam protection)	Yes (low util)
f_{ord}	Escrowed; paid to orderers on Batch-Confirmed, else burned	Yes
f_{bun}	Escrowed; paid to bundler on Bitcoin confirmation	Yes

$$f(t) = f_{\text{ix}}(t) + f_{\text{ord}}(t) + f_{\text{bun}}(t) \tag{3}$$

These fee planes are orthogonal: a transaction may use ordering without bundling, bundling without ordering, both, or neither. When neither service is requested and indexer utilization is low, a transaction can have zero KOR fee.

3.2.1 Indexer Fee (Resource Rent)

The indexer fee prices two scarce resources under a single congestion regime:

$$f_{\text{ix}}(t) = p_{\text{gas}}(t) \cdot g + p_{\text{data}}(t) \cdot d \tag{4}$$

where g is gas consumed and d is data bytes.

Three-region pricing. Both p_{gas} and p_{data} share a common congestion multiplier $\beta(t)$:

$$\beta(t) = \begin{cases} \beta(t-1) \cdot \lambda_{\text{decay}} & \text{if } u(t-1) < u_{\text{low}} \\ \max\left(s\left(\frac{u(t-1)-u_{\text{low}}}{u_{\text{high}}-u_{\text{low}}}\right), \beta(t-1) \cdot \lambda_{\text{decay}}\right) & \text{if } u_{\text{low}} \le u(t-1) < u_{\text{high}} \\ \max(\beta(t-1), 1) \cdot \left(1 + \kappa_{\text{cong}} \cdot \left(u(t-1) - u_{\text{high}}\right)\right) & \text{if } u(t-1) \ge u_{\text{high}} \end{cases} \tag{5}$$

In practice, each price is updated as $p_{\text{gas}}(t) = \varphi_{\text{base,gas}} \cdot \beta(t)$ and $p_{\text{data}}(t) = \varphi_{\text{base,data}} \cdot \beta(t)$, so $\beta(t)$ can be viewed as a dimensionless congestion multiplier applied to separate base prices for computation and data. When $\beta(t) = 1$, prices equal their base levels $\varphi_{\text{base,gas}}$ and $\varphi_{\text{base,data}}$.

Define per-block utilization for each resource as $u_{\text{gas}}(t) = \frac{G(t)}{G_{\max}}$ and $u_{\text{data}}(t) = \frac{D(t)}{D_{\max}}$. The congestion multiplier is driven by a single aggregate utilization $u(t) = \max\big(u_{\text{gas}}(t), u_{\text{data}}(t)\big)$. Here $s(x) = 3x^2 - 2x^3$ is the smoothstep function. **Initial condition:** $\beta(0) = 0$ (free at genesis).

Figure 3 shows the **instantaneous** mapping from utilization to the congestion multiplier, ignoring state. In Region 3 the curve is flat at $\beta = 1$ because the **multiplicative** behavior comes from repeatedly applying the recurrence over time via the $\beta(t-1)$ term when u remains above u_{high} (Figure 4).

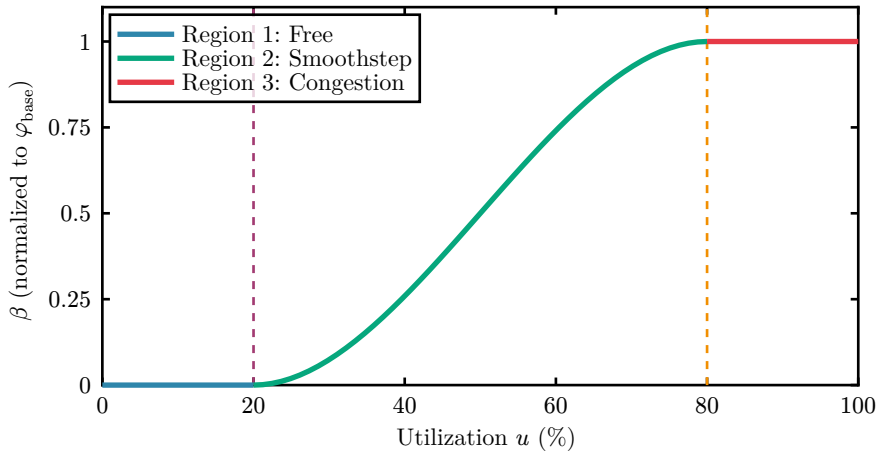

Figure 3: Three-region congestion pricing (instantaneous, state-free view). Region 1 ($u < 20\%$): $\beta = 0$. Region 2 ($20\% \le u < 80\%$): β rises smoothly from 0 to 1. Region 3 ($u \ge 80\%$): β is floored at 1; multiplicative growth is a **time** effect from the recurrence when congestion persists (Figure 4).

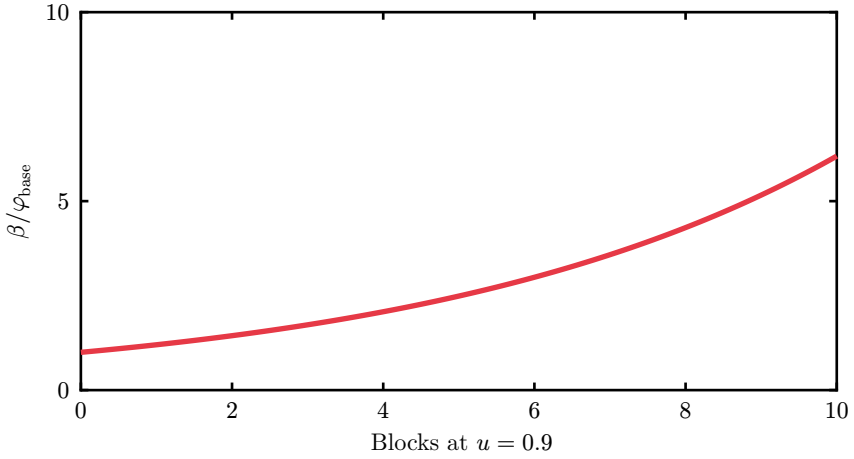

Figure 4: Price growth under sustained congestion ($u = 0.9$, $\kappa_{\text{cong}} = 2.0$). After 10 blocks, fees reach $\approx 6\times$ the base price.

- **Region 1 ($u < u_{\text{low}}$)**: Low utilization. Price decays toward zero at rate λ_{decay}, enabling true zero-cost transactions during bootstrap.
- **Region 2 ($u_{\text{low}} \leq u < u_{\text{high}}$)**: Moderate utilization. Price follows smoothstep, providing C^1-continuous transition.
- **Region 3 ($u \geq u_{\text{high}}$)**: High utilization. Price grows multiplicatively each block (Figure 4), creating strong backpressure.

Congestion pricing parameters:

- u_{low}, u_{high}: Utilization thresholds
- $\varphi_{\text{base,gas}}$, $\varphi_{\text{base,data}}$: Base prices at smoothstep top
- κ_{cong}: Price growth rate above u_{high}
- λ_{decay}: Price decay rate per block when utilization drops
- $G_{\max}$, $D_{\max}$: Maximum gas and data per block

Fee metering. During transaction processing, the protocol meters both resources independently:

- **Gas metering**: Each smart contract operation consumes gas units according to its computational cost. The total gas g consumed by a transaction is tracked during execution.
- **Data metering**: Transaction data bytes d are measured, including smart contract call data, file storage proofs, and other on-chain data.

Indexer fee settlement. The indexer fee $f_{\text{ix}}(t)$ is **burned** upon transaction execution. It functions as rent for protocol resources and spam protection—not as payment to any particular party. Indexers are funded by emissions, not by fees.

3.2.2 Ordering Fee (Optimistic Consensus Service)

Ordering stakers are funded primarily by emissions, which cover the base cost of providing optimistic consensus. The ordering fee $f_{\text{ord}}(t)$ is an optional payment that users can include to increase priority or express willingness to pay for service.

Ordering fees are escrowed when a transaction is included in a signed batch. On BatchConfirmed, a fraction β_{fee} is burned and the remainder is paid to batch signers proportionally to bonded stake. On expiry or rollback, the fee is burned.

3.2.3 Bundling Fee (Aggregation Service)

Bundlers publish aggregated transactions to Bitcoin, paying BTC fees for block space. The bundling fee $f_{\text{bun}}(t)$ compensates bundlers for this cost.

Bundling fees are escrowed when a bundler commits to include the transaction. On Bitcoin confirmation, a fraction β_{fee} is burned and the remainder is paid to the bundle publisher. Bundle publishers are identified explicitly in the bundle structure (see Kontor Scalability [6]).

3.3 Staking

Kontor uses a unified staking model: operators maintain a single KOR stake balance that simultaneously backs storage commitments and ordering participation . Stake is not needed to "secure the network"; instead, stake serves as collateral enabling specific protocol functions, with slashing for misbehavior.

3.3.1 Delegation

Delegation allows KOR holders to earn rewards without running infrastructure. Delegated KOR is non-custodial: the operator cannot withdraw it. Operators set a commission rate

$\gamma \in [0, 1]$; delegator rewards are $(1 - \gamma)$ of the operator's rewards, distributed proportionally to delegation amounts. Commission changes take effect after $T_{\text{commission}}$ blocks, allowing delegators to exit before unfavorable changes.

3.3.2 Stake Requirements

An operator's effective stake $k_{\text{eff}}(n, t) = k_{n(t)} + \sum_{d \in D_{n(t)}} a_{d(t)}$ (own stake plus delegations) determines capacity across all activities. The total required stake at time t is:

$$k_{\text{req}}(n, t) = k_{\text{req,storage}}(n, t) + k_{\text{req,ordering}}(n, t) \tag{6}$$

Operators must maintain $k_{\text{eff}}(n, t) \geq k_{\text{req}}(n, t)$; violation triggers automatic removal from activities until stake sufficiency is restored. Operators must also maintain minimum self-stake $k_{n(t)} \geq \alpha_{\min} \cdot k_{\text{eff}}(n, t)$ as a fraction of effective stake.

Optimistic consensus uses per-batch bonds as stake-at-risk. Hard collateralization is enforced by requiring that the operator's outstanding bond exposure on unresolved batches is included in $k_{\text{req,ordering}}(n, t)$.

3.3.3 Slashing

Operators bear first-loss: their own stake is slashed before delegated stake. If the penalty exceeds the operator's self-stake, the remainder is distributed proportionally among delegators. This ensures operators have skin in the game and delegators are protected up to the operator's self-stake.

3.3.4 Unbonding

Stake withdrawal requires an unbonding period of T_{unbond} blocks. During this period, stake does not back protocol activities, earns no rewards, but remains slashable. The unbonding period must exceed detection windows for all slashable offenses. Formally, the protocol chooses T_{unbond} such that $T_{\text{unbond}} \geq \max(W_{\text{proof}}, W)$, ensuring that evidence for any slashable event can be included before stake becomes liquid. Redelegation transfers instantly between operators but remains slashable by both operators for T_{grace} blocks.

3.3.5 Security

Attack	**Description**	**Defense**
Commission manipulation	Raise commission after attracting delegations	Notice period $T_{\text{commission}}$
Slashing escape	Redelegate to escape pending slash	Grace period T_{grace}
Unstaking race	Misbehave before unbonding completes	$T_{\text{unbond}} \geq T_{\text{withdrawal}}$

Failure Mode	**Recovery**
Operator slashed	Delegators lose proportionally; redelegate
Operator exits	Delegations unbond normally
Mass unstaking	Higher rewards attract new stake

4 Conclusion

Bitcoin's brilliance is not found in the elegance of its code, which is, in fact, extremely idiosyncratic and complex, but in the holistic design of its ecosystem. The economic interplay

between miners and full nodes, the carefully managed soft forks, the strict adherence to backwards compatibility, and even the seemingly wasteful proof-of-work mechanism—these all contribute to a system of unparalleled robustness and value. The strength of a blockchain ecosystem lies not in its breadth, but in its depth. Healthy markets are always recognizable by the layers of products and abstractions that they support. So, while building alternative blockchains—whether they be sidechains or ZK-rollups—is in fact dilutive to Bitcoin, building on *top* of Bitcoin is *additive* to the value of the larger ecosystem.

Kontor neither competes with Bitcoin, nor does it attempt to change what Bitcoin is: Kontor's approach is in fact the true "Bitcoin Maxi" platform for DeFi, in contrast to that of protocols which strive, in one way or another, to treat BTC as gas rather than digital gold. Kontor is built for Bitcoin because that is precisely where the most digital value resides, and the Kontor project is a conscious effort to grow the Bitcoin economy and ecosystem beyond its current scope and limits while being supportive of Bitcoin and true to its original mission.

The architecture presented in this paper—Sigil smart contracts, perpetual file storage, optimistic consensus, and scalable transaction encoding—forms a coherent system where each component reinforces the others. Smart contracts govern storage agreements; the file persistence layer provides data availability for contract state and NFT content; optimistic consensus enables sub-block confirmation for time-sensitive applications; and transaction batching makes all of this economically viable at scale. Unified KOR staking across all protocol activities creates capital efficiency and aligns incentives across the network.

This path calls for a pragmatic approach rooted in incremental developments within decentralized finance rather than in speculative futurism and the associated hype cycles. By thoughtfully iterating upon Bitcoin's stable and battle-tested platform, it is possible to leverage its powerful network effects to create new and exciting layers of financial innovation. A modern Bitcoin metaprotocol has the potential to redirect the locus of innovation and value creation back to Bitcoin, the natural center of the entire cryptocurrency industry. It also has the potential to serve as a nexus for interoperability and synergy between and across all protocols that maintain compatibility with Bitcoin and Bitcoin Script. In essence, Kontor represents a return to the original promise of blockchain technology, as a platform for trustless and peer-to-peer commerce and value creation.

5 Appendix

5.1 Protocol Parameters

5.1.1 Global Parameters

Symbol	Description	Suggested
B	Bitcoin blocks per year	52,560
$\text{KOR}_{\text{initial}}$	Total KOR supply at genesis	10^9
μ_0	Annual emission rate	0.10

5.1.2 Staking Parameters

Symbol	Description	Suggested
T_{unbond}	Unbonding period (blocks)	2016 (~2 weeks)
$T_{\text{commission}}$	Commission change notice period (blocks)	2016 (~2 weeks)
T_{grace}	Redelegation slashing grace period (blocks)	2016 (~2 weeks)
$\alpha_{\min}$	Minimum self-stake fraction	0.10
χ	Ordering emission fraction	0.10

5.1.3 Storage Parameters

Symbol	Description	Suggested
$n_{\min}$	Minimum storage nodes per file	3
$s_{\min}$	Minimum file size	10 KB
$s_{\max}$	Maximum file size	100 MB
c_{stake}	Base stake normalization factor	2×10^6 KOR
F_{scale}	Network size normalization factor	50,000
χ_{fee}	Ratio of user fee to per-node base stake	0.003
λ_{slash}	Slash penalty multiplier	30
β_{slash}	Burn fraction of slash penalty	0.50
C_{target}	Target challenges per file per year	12
s_{chal}	Symbols sampled per challenge	100
W_{proof}	Proof submission window (blocks)	2016 (~2 weeks)
W_{offer}	Sponsorship offer expiration (blocks)	144 (~1 day)
β_{bond}	Sponsorship bond amount	~3 KOR

5.1.4 Ordering Parameters

Symbol	Description	Suggested
$\sigma_{\min}$	Minimum stake to become a staker	10M KOR
q	Quorum threshold (stake fraction)	$\frac{2}{3}$
W	Batch expiry window (blocks)	12

Symbol	Description	Suggested
k	Bitcoin finality depth (confirmations)	6
λ_{equiv}	Equivocation penalty (fraction of stake)	1.00
λ_{invalid}	Invalid batch penalty (fraction of stake)	0.10
$\lambda_{\text{conflict}}$	Conflict penalty (fraction of stake)	0.10
σ_{expiry}	Expiry base penalty per transaction	10K KOR
λ_{cap}	Max bonded stake lost per batch	0.01
β_{slash}	Slash burn rate (rest to reporter)	0.50
β_{fee}	Service fee burn rate	0.50

5.2 Staking Formulas

5.2.1 Delegation and Commission

Operators set commission rate $\gamma \in [0, 1]$. When earning reward r:

$$r_{\text{operator}} = r \cdot \gamma \tag{7}$$

$$r_{\text{delegators}} = r \cdot (1 - \gamma) \tag{8}$$

5.2.2 Slashing

Operators bear first-loss:

$$\text{slash}_{\text{self}} = \min(\text{slash}_{\text{total}}, k_n) \tag{9}$$

$$\text{slash}_{\text{delegators}} = \max(0, \text{slash}_{\text{total}} - k_n) \tag{10}$$

5.2.3 Slashing Offenses

The unbonding period must exceed detection windows for all slashable offenses:

Offense	Detection	Window
Storage proof failure	Challenge expires	W_{proof}
Ordering batch expiry	Deadline passes	W
Ordering conflict	Conflicting tx confirms	$\leq W$
Ordering equivocation	Fraud proof	Immediate

5.2.4 Delegation Flow

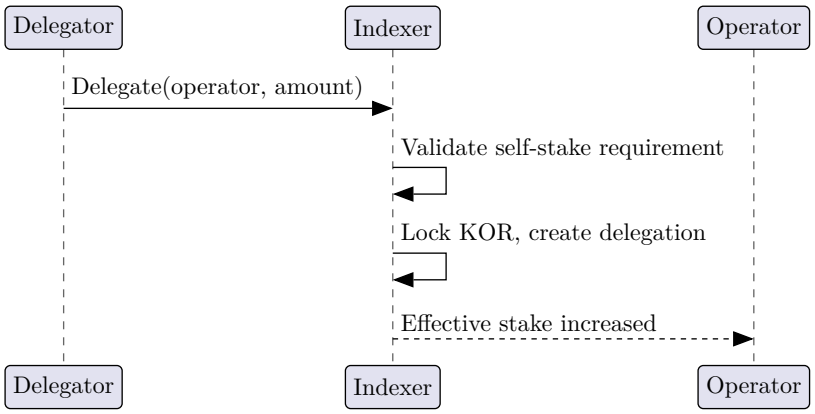

Figure 5: Delegation Flow

5.2.5 Undelegation Flow

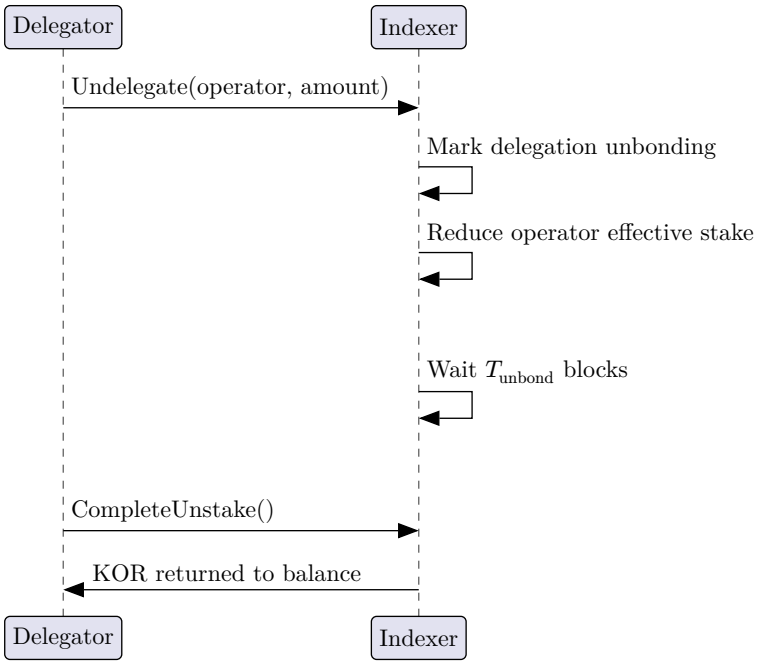

Figure 6: Undelegation Flow

6. Bibliography

[1] A. Back *et al.*, "Enabling Blockchain Innovations with Pegged Sidechains," 2014. [Online]. Available: https://blockstream.com/sidechains.pdf

[2] Hal Finney, "Bitcoin overlay protocols." [Online]. Available: https://bitcointalk.org/index.php?topic=2077.msg26888#msg26888

[3] V. Buterin, [Online]. Available: https://x.com/VitalikButerin/status/929804867568373760

[4] "Ordinals Milestone: 100M Inscriptions Reached." [Online]. Available: https://outposts.io/article/ordinal-milestone-100m-inscription-reached-1a1ad115-1d60-4833-8fc5-cf4e42f16fe3

[5] "Runes make up 68% of all Bitcoin transactions since launch." Cointelegraph, Apr. 25, 2024. [Online]. Available: https://cointelegraph.com/news/bitcoin-runes-make-up-68-of-all-transactions-since-launch

[6] Adam Krellenstein, "Kontor Scalability," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/scalability

[7] Adam Krellenstein, "Kontor Optimistic Consensus," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/optimistic-consensus

[8] Ordinals, "Inscriptions." [Online]. Available: https://docs.ordinals.com/inscriptions.html

[9] The Bitcoin Core developers, *Bitcoin Core*. GitHub. [Online]. Available: https://github.com/bitcoin/bitcoin/blob/master/src/policy/policy.h

[10] Pieter Wuille, "BIP 340: Schnorr Signatures for secp256k1." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki

[11] Bitcoin Optech, "X-only public keys." [Online]. Available: https://bitcoinops.org/en/topics/x-only-public-keys/

[12] James Munns, "Postcard." [Online]. Available: https://postcard.jamesmunns.com/

[13] Andrew Chow, "BIP 174: Partially Signed Bitcoin Transaction Format." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0174.mediawiki

[14] Alkanes documentation contributors, "Interacting with Contracts." [Online]. Available: https://alkanes-docs.vercel.app/docs/developers/contracts-interaction

[15] Bytecode Alliance, "The WebAssembly Component Model." [Online]. Available: https://component-model.bytecodealliance.org/

[16] Bytecode Alliance, "WIT Reference - The WebAssembly Component Model." [Online]. Available: https://component-model.bytecodealliance.org/design/wit.html

[17] Bytecode Alliance, *wit-bindgen: A language binding generator for WIT*. (2025). GitHub. [Online]. Available: https://github.com/bytecodealliance/wit-bindgen

[18] NEAR Protocol, "Notes on Serialization." [Online]. Available: https://docs.near.org/smart-contracts/anatomy/serialization

[19] Parity Technologies, *parity-scale-codec: Lightweight, efficient, binary serialization and deserialization codec*. (2025). GitHub. [Online]. Available: https://github.com/paritytech/parity-scale-codec

[20] CosmWasm, "WasmMsg in cosmwasm_std." [Online]. Available: https://docs.rs/cosmwasm-std/latest/cosmwasm_std/enum.WasmMsg.html

[21] Ethereum Foundation, "Contract ABI Specification." [Online]. Available: https://docs.soliditylang.org/en/latest/abi-spec.html

[22] NEAR Protocol, "Cross-Contract Calls." [Online]. Available: https://docs.near.org/smart-contracts/anatomy/crosscontract

[23] Polkadot, "Cross-Contract Calling." [Online]. Available: https://use.ink/docs/v5/basics/cross-contract-calling/

[24] Adam Krellenstein, Alexey Gribov, and Ouziel Slama, "Kontor Storage Protocol," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/storage-protocol

[25] Filecoin, [Online]. Available: https://www.filecoin.io/blog/posts/introducing-proof-of-data-possession-pdp-verifiable-hot-storage-on-filecoin/

[26] Sam Williams, Viktor Diordiiev, Lev Berman, India Raybould, and Ivan Uemlianin, "Arweave: A Protocol for Economically Sustainable Information Permanence," 2023. [Online]. Available: https://www.arweave.org/yellow-paper.pdf

[27] Hovav Shacham and Brent Waters, "Compact Proofs of Retrievability„" *Springer*.

[28] Adam Krellenstein and Alexey Gribov, "Kontor Proof-of-Retrievability," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/crypto

[29] Abhiram Kothapalli and Srinath Setty, "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes," 2021. [Online]. Available: https://eprint.iacr.org/2021/370

[30] Henning Pagnia and Felix C Gärtner, "On the Impossibility of Fair Exchange Without a Trusted Third Party„" *Darmstadt University of Technology Technical Report,*.