KONTOR: A NEW BITCOIN METAPROTOCOL FOR SMART CONTRACTS AND FILE PERSISTENCE

November 11, 2025

ADAM KRELLENSTEIN
adam@unspendablelabs.com

WILFRED DENTON
wilfred@unspendablelabs.com

OUZIEL SLAMA
ouziel@unspendablelabs.com

Contents

1. Introduction . . . 2
  1.1. Blockchains and Decentralized Finance . . . 2
  1.2. Decentralized Finance on Bitcoin . . . 3
  1.3. Bitcoin's Tokenomics . . . 4
  1.4. The Problem with Layer-2s and Rollups . . . 5
  1.5. Metaprotocols . . . 6
2. The Kontor Protocol . . . 6
  2.1. Kontor Transactions . . . 7
    2.1.1. Bitcoin Integration . . . 7
    2.1.2. Transaction Encoding . . . 8
  2.2. Sigil Smart Contract Framework . . . 10
    2.2.1. Runtime . . . 11
    2.2.2. Surface Language . . . 13
    2.2.3. Modularity . . . 14
    2.2.4. Upgradeability . . . 16
    2.2.5. Deterministic Execution . . . 16
  2.3. File Persistence Protocol . . . 17
    2.3.1. Background . . . 17
    2.3.2. Design Principles . . . 18
    2.3.3. Protocol Description . . . 19
    2.3.4. Storage Node Economics . . . 23
    2.3.5. Security Analysis . . . 24
  2.4. Macroeconomics . . . 25
    2.4.1. Emission Tapering and Network Maturity . . . 25
    2.4.2. Dynamic Health-Based Adjustment . . . 26
    2.4.3. Burn Mechanisms and Long-Term Stability . . . 26
    2.4.4. Economic Equilibrium . . . 27
3. Indexer Reference Implementation . . . 27
    3.0.1. Database Tables . . . 28
    3.0.2. API and SDK . . . 31
4. Conclusion . . . 32
5. Bibliography . . . 33

1. Introduction

1.1. **Blockchains and Decentralized Finance.**

The first application of blockchain technology was Bitcoin, a decentralized, peer-to-peer currency and method of payment created in 2009, but by 2013, it was well understood that blockchain technology could be used for more than the creation of digital currencies—that blockchains were about more than just Bitcoin. At this time, "Bitcoin 2.0" projects worked to migrate more and more financial systems to blockchain-based infrastructure; to disintermediate the aged and encumbered infrastructure of traditional financial markets by moving assets onto a blockchain and allowing them to assume a digitally native form. By early 2014, with the publication of the Ethereum Whitepaper, an even grander vision was established: a vision to build a blockchain-based "World Computer" that would replace centralized technological infrastructure with permissionless and peer-to-peer alternatives —alternatives that would leverage the spare computing capacity of the world to create a techno-utopian reality of digital self-sovereignty governed by rational economic incentives and code-as-law. [1] Nevertheless, in 2025, the vast majority of the value created by the blockchain industry is still to be found in digital assets, especially financial instruments (including the cryptocurrencies themselves) rather than in the value of the services that the industry has nominally been working to decentralize. Ten years after the creation of Ethereum, we are no closer to that utopian future of the World Computer to replace Amazon Web Services, even as the value of the blockchains themselves and the assets created on top of them is near an all-time high. That decentralized finance (DeFi) and tokenization are still the most prominent use cases for blockchain technology is a powerful demonstration of this fact.

The root of the disconnect between the hype and the reality is based on a misunderstanding of the fundamental nature of blockchains. A blockchain is a decentralized system built using state-machine replication and a Byzantine fault–tolerant (BFT) consensus protocol, i.e. a logically consistent system without any central points of control or failure.[1] Decentralized systems are most useful *a priori* when *trust* is an important consideration, and trust is particularly critical when there is *value* at stake. In the operation of any blockchain, there is the simple necessity of replicating a transaction to the computers of anonymous network participants whose systems then need to verify the integrity of that transaction in a consistent order. This operation has the potential to provide very high assurances as to transaction availability and correctness; it is, however, relatively slow and inefficient. In practice, this does not pose as great a challenge as might be assumed: if the value of a transaction is low, then I can trust a third party to manage it for me and therefore do not need a blockchain; if it is high, then the speed with which the transaction is executed is of lesser importance relative to the guarantees provided by the decentralized and trustless manner in which settlement occurs.

The chief problem with the World Computer narrative is that computation is a commodity, and there is no particular great motivation for it to be decen-

[1]In Bitcoin and other permissionless blockchains, notably, this BFT protocol must also be Sybil-resistant.

tralized for its own sake.[2] Computation itself isn't particularly valuable, and, as a consequence of their relative inefficiency, blockchains aren't very good at it; decentralized computation is not inherently better than centralized computation. Financial transactions, on the other hand, are fundamentally bi- or multi-lateral and decentralized: a certain party wants to create an asset which, in turn, one or more parties want to purchase, etc. Wherever traditional market infrastructure does not exhibit these properties, it is an explicit flaw in the financial system for historical reasons (such as the "Paperwork Crisis" of the late 1960s and early '70s that led to the creation of the DTC in the United States). [2]

The core use case for blockchain technology then was immanent in the very first application of the technology: blockchains are useful for creating, managing and transferring value. This is the reason that Bitcoin is still the most valuable blockchain even today. Value creation supports value creation, and the powerful incentives associated with a native digital currency create virtuous cycles in the production of greater value. Performing computation on a blockchain must be a means to an end, and that end must be connected directly with the creation of economic value.

1.2. **Decentralized Finance on Bitcoin.**

Insofar as blockchains are naturally self-contained ecosystems and markets, building each new blockchain network is fundamentally dilutive to Bitcoin's value. The ethos behind Bitcoin Maximalism is rooted in the recognition of this fact: the diffusion of resources away from Bitcoin—especially towards speculative technological development with ill-defined goals—is a loss for the industry as a whole. But Bitcoin alone cannot do everything:

1. Bitcoin Script is extremely limited in comparison with modern smart contract languages, which are stateful and pseudo–Turing Complete.
2. Bitcoin's tokenomics do not support a gas model for smart contracts; the BTC token is ultimately deflationary and fees are based on transaction weight only.
3. Bitcoin is ill-suited to persisting significant quantities of data that may be used for NFT issuance and DeFi protocols.

Immense economic pressure to unlock the value of the Bitcoin network has produced numerous projects and protocols that in one way or another attempt to grow the capabilities of Bitcoin, with mixed success and uncertain prospects. Indeed, even as Bitcoin's value proposition has expanded beyond simply serving as sound, censorship-resistant money, the Bitcoin ecosystem has been held back by a fierce tribalism that has fragmented its infrastructure and economy.

Since 2014, the central debate has revolved around what it is appropriate to use the Bitcoin network for. Conflicting narratives regarding file storage and smart contract capabilities have proliferated, creating tension between Bitcoin and other

[2]N.B. There may be cases where it is actually more efficient to perform certain kinds of computation, or data storage and transfer, in a decentralized fashion—for instance with BitTorrent or Folding@home—however it is no coincidence that this 'computational' work is essentially charitable in nature.

protocols built on top of it. This in turn led to the multiple `OP_RETURN` wars [3] and the Blocksize War, [4] with the consequent altcoins including Bitcoin Cash [5] and Ethereum [6]. And none of the efforts at preventing the use of the Bitcoin blockchain for non-native use cases has had the least effect. By late 2023 and into 2024, Ordinals and Inscriptions repeatedly captured a majority share of Bitcoin transactions over extended periods. [7]

In 2025, the question is no longer *whether* Bitcoin will be used for more than just payments in BTC. DeFi is coming to Bitcoin. The question is now, What will it look like? Will it be helpful or harmful to the ecosystem as a whole? How will DeFi on Bitcoin be different from DeFi on Ethereum and Solana? If this matter is not handled correctly, it could be disastrous for the Bitcoin network and the value of BTC. And while there are many efforts both extant and planned to bring a more powerful smart contract system to the Bitcoin ecosystem, these projects have largely been unsuccessful because they have failed to provide the proper tokenomics nor have they implemented the necessary file persistence solution.

1.3. **Bitcoin's 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 tokens for smart contract execution in one form or another. But 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.

The analogies to physical assets are elucidative: Bitcoin serves as digital gold and other cryptocurrencies as "digital gasoline"—if gold were (widely) used as fuel directly, then it either would not be a good fuel, because it would be too expensive; or it would not be a good money, because it would be consumed aggressively. We can see this directly: holders of gold all benefit when the price is high; users of gasoline benefit when the price is low. The natural relationship between Bitcoin and a gas token is the same as that which exists between money and fuel: people buy their gas with their money in order to use the gas. For a healthy smart contracts system, you need both.

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. The increased transaction fees associated with the greater density of protocols residing on the Bitcoin blockchain are only problematic under the assumption that Bitcoin is supposed to compete directly with inherently much more efficient centralized systems for which small constant-factor cost-reductions are in fact relevant. 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.

1.4. **The Problem with Layer-2s and Rollups.**

Up until the present day, efforts to extend the Bitcoin blockchain have predominantly followed the *sidechain* architectural pattern. A sidechain is, in simple terms, a blockchain which has an independent consensus system but also includes a mechanism for transferring state to and from a "mainchain". In the case of Bitcoin, this mechanism takes the form of a two-way peg with BTC, such that BTC may be locked on the Bitcoin ledger and its value temporarily "transferred" to the sidechain, which may support additional functionality, faster block times, etc. Many of the developers of Bitcoin Core have been engaged in the development of sidechain technology going back to the founding of Blockstream in 2014. [8], [9] What makes the sidechain approach particularly attractive is that it is also touted as a scaling solution. Indeed, improving the scalability of Bitcoin has always been the highest priority for all extensibility efforts; but while sidechains have long been an interesting avenue of technological development, from a game-theoretic perspective, the architectural pattern is fundamentally problematic: sidechains are not in fact a means of improving the scalability of the Bitcoin network such as it is—they are better understood as *alternatives* to Bitcoin, with independent consensus mechanisms, security guarantees, and economics.

Sidechains of all sorts, by design, are inexorably parasitic *vis-à-vis* the underlying Bitcoin blockchain: they siphon transaction volume and transaction fees away from the Layer-1 network, weakening the L1′s security and threatening its economic model. The more successful a sidechain is, the more harm it does—and the negative effects consequent to sidechain adoption are clearly visible within the Ethereum ecosystem, where migration of activity away from Ethereum to Polygon, Arbitrum, Optimism, etc. has placed downward pressure on the price of ETH. Sidechains, moreover, must implement their own consensus system and then rely on a set of (often centralized) entities to validate all logic that is outside of the Bitcoin protocol. Such architectures are often described euphemistically as "trust-minimized" because their security model is qualitatively weaker than that of Bitcoin (cf. Stacks [10], Citrea [9]) Much of the development effort undertaken in the past decade to improve Bitcoin has been dedicated to mitigating the weakened security associated with the sidechain architecture. [11]

The most promising development along these lines has been the use of zero-knowledge proofs to mitigate the trust assumptions that Nakamoto consensus would otherwise provide (and thereby to improve the sidechain–mainchain state transfer mechanism). Indeed, zero-knowledge proofs are here strictly an *alternative* to the blockchain architecture rather than an improvement thereupon. A rollup is simply a sidechain with a different mechanism for anchoring to the Layer-1. Fortunately, the limitations of Bitcoin Script make the deployment of zero-knowledge rollups on Bitcoin quite challenging. This is the origin of the political movement to enable additional opcodes in Bitcoin Script, in particular `OP_CAT`, which would allow for the anchoring of zkRollups onto Bitcoin without relying on the fraud proofs of BitVM. ColliderVM offers a mechanism for bypassing this restriction, but it is still computationally and economically impractical. [12] Citrea works today, but it requires a "toxic-waste" ceremony for initializing the parameters of its Groth16-based proof scheme, [13] and involves the introduction of five new protocol participants (Operators, Watchtowers, Challengers, Signers, and Users) in order to emulate Bitcoin covenants with "committees" of at least one honest party. [9] These technological solutions either break Bitcoin's fundamental trust assumptions, or, if they don't, have greater potential to dilute Bitcoin's value than even the most promising altchains; for, unlike the latter, which offer merely a 'wrapped' BTC predicated on centralization, zkRollups propose to take BTC itself off the Bitcoin network.

1.5. Metaprotocols.

The best alternative to sidechains and rollups are **metaprotocols**, which address the limitations of vanilla Bitcoin not by building alternative blockchain networks, nor by fundamentally changing Bitcoin's value proposition, but rather by building directly on top of Bitcoin. A metaprotocol is an extension to an underlying blockchain that parses data from the blockchain and 'interprets' them in a novel way (using what may be termed "embedded consensus"). Data in the blockchain ledger that are explicitly ignored by Bitcoin nodes are parsed by the metaprotocol implementation and the additional state is derived deterministically, such that metaprotocols are capable of adding arbitrary functionality to the blockchain on which they reside while relying on the blockchain to provide transaction finality, ordering and availability. This effectively takes the architectural pattern of a blockchain to its logical conclusion, extending the underlying protocol by adding parsing logic and derived state and treating the Bitcoin blockchain as a log-structured store for additional protocol messages. Metaprotocols are generally characterized as "Layer-1" solutions because all of the protocol data reside on the underlying blockchain. [14]

A Bitcoin metaprotocol has the same security model as Bitcoin: every metaprotocol transaction maps to a Bitcoin transaction, and the complete history of transactions is thus secured with the full hashpower of the Bitcoin mining network. The only difference is that users of the metaprotocol must rely on two codebases rather than on Bitcoin Core alone. Bitcoin miners, being 'unaware' of the metaprotocol, validate a smaller part of the *complete* protocol than the Bitcoin protocol; but this distinction is quantitative rather than qualitative. Accordingly, metaprotocols are fundamentally synergistic with Bitcoin, and Bitcoin miners are incentivized to relay metaprotocol transactions, to include them in their blocks, and to support protocol changes that encourage their adoption. By contrast, miners are not incentivized to implement protocol changes to Bitcoin which would threaten its economics by making it easier to implement zero-knowledge rollups which have the potential to further balkanize the Bitcoin ecosystem and reduce the fees that miners collect.

2. The Kontor Protocol

Kontor is a new Bitcoin-based metaprotocol with a powerful smart contract system and a built-in file-persistence layer offering strong perpetual availability guarantees. Kontor is designed from the ground up for true interoperability with

Bitcoin and other Bitcoin metaprotocols and for providing a powerful, safe and easy-to-use smart contracts system that enables the creation of rich and sophisticated applications on the Bitcoin network. Kontor thus has the potential to support the development of the native Bitcoin ecosystem and to serve as the nexus for decentralized finance on the Bitcoin blockchain, the blockchain network with both the deepest history and the greatest value.

Because Bitcoin does not have the tokenomics to act as a fuel for smart contract execution, the Kontor network's native currency, called KOR, serves as the gas token for this purpose, and it is complementary to, rather than competitive with, BTC. KOR may be purchased on demand with Bitcoin, on-chain, using trustless atomic swaps, in the very same commit-reveal transactions that execute the desired smart contract. BTC serves as the unit of account for the Kontor network and may be found in the default currency trading pair of Kontor-based decentralized exchanges. Every Kontor transaction also requires that a BTC transaction fee be paid for inclusion in the Bitcoin network, and in proportion to the cost imposed on Bitcoin users. KOR, unlike BTC, is fundamentally inflationary. Emissions incentivize the perpetual storage of files; KOR fees (for use of the file persistence network and smart contract execution) are burned. The protocol implements a sophisticated emission tapering mechanism where new files receive rewards that decrease as the network grows, ensuring that total emissions remain relatively stable even as the network scales to billions of files.

2.1. **Kontor Transactions.**

2.1.1. *Bitcoin Integration.*

Whereas many blockchain projects over the years have claimed to offer cross-chain interoperability and cross-chain asset transfers, almost all such schemas require trusted escrow agents (cf. Stacks [10]) or provide nothing more than mere standardization at the API level (cf. Cosmos [15]). Because a blockchain is a consensus system, true protocol interoperability requires that both protocols necessarily reside *on the same network*; only then do nodes of each network have access to the same transaction ledger, which is a necessary condition for the most basic consistency guarantees. That is, only by building directly on top of another blockchain can one achieve true protocol interoperability.

Traditional sidechains provide only very weak interoperability with their parent chain, requiring many hours even to move assets from the parent chain to the side chain or *vice versa*, and often with relatively weak security guaranties. Zero-knowledge rollups provide greater interoperability, but at the economic cost of parasitizing the underlying blockchain (Section 1.4). Other Bitcoin metaprotocols offer interoperability within the Bitcoin ecosystem but at the expense of general functionality—they are limited by their compatibility with Bitcoin Script.

Kontor offers deep, native integration with Bitcoin and other Bitcoin metaprotocols, while also providing the power and flexibility of sidechains and altcoins generally—all completely on-chain. Kontor achieves this by defining an interface for moving transferable assets between an account-based smart contract system and a UTXO-based balance system. Users may *attach* assets to a Bitcoin UTXO such that the asset ownership is then tracked as it is spent according to Bitcoin's own rules; users may then *detach* the asset from other UTXOs so that the balance may again be modified by a Kontor smart contract.[3]

The most important feature of this development is that which allows for trustless, single-confirmation, atomic swaps between Kontor assets and both Bitcoin and Bitcoin UTXO–based metaprotocols, including Ordinals, Runes and Counterparty. Notably, `attach` and `detach` transactions may be chained with Kontor smart contract calls so that Bitcoin users are able to swap their BTC for KOR and use that KOR for smart contract execution (including to pay fees for the file persistence protocol) *in a single confirmation*.

2.1.2. *Transaction Encoding.*

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

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, [16] 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. [17] The Kontor SDK (Section 3.0.2.2) 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

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

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 is such a way that the data are completely ignored during evaluation by the Bitcoin protocol.

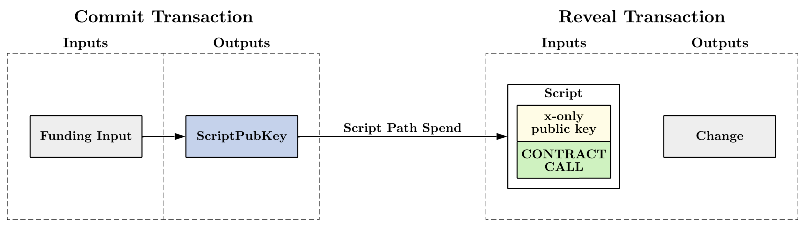
```
/// Tapscript

// 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 [18], [19] 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 CBOR (RFC 8949). [20]

P2TR uses a single address type, eliminating the distinction between output types and enhancing privacy, as observers cannot determine whether a P2TR output uses a Key Path Spend (akin to P2WPKH) or a Script Path Spend (akin to P2WSH). [21] A user's Kontor wallet displays a Bitcoin P2TR ScriptPubKey as their Taproot address, designed for a Key Path Spend, enabling spending via a Schnorr signature with the user's private key. This address is available within the smart contract execution environment for authentication of contract operations. Bitcoin wallets such as Xverse expose both the Key Path Spend address and the x-only public key, as they are essential for constructing partially signed bitcoin transactions (PSBTs) [22] and supporting Script Path Spends. Notably, a Kontor P2TR output is cryptographically unlinkable to the user's Key Path Spend address, ensuring that the Reveal transaction cannot be directly linked by any adversaries to the user's address used for vanilla Bitcoin transactions.

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.

Asset balance swaps with Bitcoin or other Bitcoin metaprotocols may be performed by first attaching a balance to a UTXO and then detaching it through a combination of partially signed Bitcoin transactions (PSBTs). [22] A swap of an asset balance begins when a seller performs an `attach` operation. The seller then creates a PSBT containing the `detach` Commit input and a price output, signing it with `SIGHASH_SINGLE | SIGHASH_ANYONECANPAY`. [23] After publishing this PSBT to an (off-chain or on-chain) swap marketplace, a buyer can purchase the swap: the buyer creates a `detach` Reveal transaction by incorporating the signed PSBT from the seller, adding a funding input and including an `OP_RETURN` output containing their x-only public key, directing Kontor to send the balance to them.

2.2. Sigil Smart Contract Framework.

Kontor is best distinguished from other Bitcoin metaprotocols by its support for rich smart contracts, and in particular its inclusion of a proper gas system for contract execution. Other metaprotocols, such as colored coins, Ordinals, Runes and BRC-20, support only the most minimal smart contract logic—that which is supported by Bitcoin Script. This effectively limits these protocols to enabling the creation of new tokens and the transfer of these tokens. Counterparty, released in 2014, supports a great deal more functionality, including a decentralized order-matching engine, prediction markets, asset dividends, oracles, fairminting and more. Counterparty, however, is limited by the fact that all of its smart contract logic is hard-coded by the protocol developers. [24] The virtual machine model, pioneered by the Ethereum project, offers a much more extensible solution by allowing individual users to publish bytecode directly to the network, which code may then be executed deterministically in the EVM by any user that pays the necessary transaction fees. [1]

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. Contract module publication involves the broadcast of WebAssembly bytecode to the Bitcoin blockchain, where it is parsed by the Kontor indexer. The functionality of Kontor 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.

2.2.1. *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. [25] 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, [26] 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. [27], [28] 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 WAVE (WebAssembly Value Encoding) for serializing all values that cross component boundaries, 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; [29] Polkadot uses the SCALE codec; [30] CosmWasm uses JSON messages; [31] EOSIO uses a custom binary format; Ethereum uses Contract ABI encoding with a 4-byte function selector for contract calls). [32] 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 WAVE and WIT 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.

```rust
use stdlib::*;

contract!(name = "token");

#[derive(Storage)]
struct TokenStorage {
    pub ledger: Map<Address, Decimal>,
}

impl Guest for Token {
    fn init(ctx: &ProcContext) {
        TokenStorage { ledger: Map::default() }.init(ctx);
    }

    fn mint(ctx: &ProcContext, n: Decimal) -> Result<(), Error> {
        if n <= Decimal::zero() {
            return Err(Error::new("amount must be positive"));
        }

        let ledger = storage(ctx).ledger();

        if !ledger.is_empty(ctx) {
            return Err(Error::new("already minted"));
        }

        let to = ctx.signer();
        let balance = ledger.get(ctx, to).unwrap_or(Decimal::zero());
        ledger.set(ctx, to, balance + n); // checked arithmetic
        Ok(())
    }

    fn transfer(ctx: &ProcContext, to: Address, n: Decimal) -> Result<(),
Error> {
        if n <= Decimal::zero() {
            return Err(Error::new("amount must be positive"));
        }

        let from = ctx.signer();
        if from == to {
            return Ok(());
        }

        let ledger = storage(ctx).ledger();

        let from_balance = ledger.get(ctx, from).unwrap_or(Decimal::zero());
        let to_balance = ledger.get(ctx, to).unwrap_or(Decimal::zero());

        if from_balance < n {
            return Err(Error::new("insufficient funds"));
        }

        ledger.set(ctx, from, from_balance - n);
        ledger.set(ctx, to, to_balance + n);
        Ok(())
    }

    fn balance(ctx: &ViewContext, acc: Address) -> Option<Decimal> {
        storage(ctx).ledger().get(ctx, acc)
    }
}
```

```
package kontor:contract;

world contract {
    include kontor:built-in/built-in;
    use kontor:built-in/context.{view-context, proc-context, signer};
    use kontor:built-in/error.{error};

    export init: func(ctx: borrow<proc-context>);

    export mint: func(ctx: borrow<proc-context>, n: decimal) -> result<_,
error>;
    export transfer: func(ctx: borrow<proc-context>, to: Address, n: decimal)
-> result<_, error>;
    export balance: func(ctx: borrow<view-context>, acc: address) ->
option<decimal>;
}
```

2.2.2. *Surface Language.*

The Sigil framework allows contracts to be written in any language that supports the WebAssembly target—contracts written in different languages can run and interoperate transparently as long as they share the same WIT interfaces. However, the Sigil SDK offers first-class support for development in a Rust eDSL. Sigil's `contract!` procedural macro automates the generation of bindings and integrates built-in interfaces. This differs from other Rust-based smart contract frameworks: for example, `ink!` for Polkadot and NEAR's Rust SDK also use macros (like `#[ink::contract]` and `#[near_bindgen]`) to reduce boilerplate, but these generate code that is tied to their specific runtime ABIs, not to a universal WIT interface. Kontor's macros implement traits like `ReadContext`/`WriteContext` for storage access and wire up the component model interfaces at compile time, which emphasizes compile-time type safety. Sigil encodes execution semantics directly in the types of exported functions: `proc` functions may read and write state and are eligible for transaction execution; `view` functions are read-only and exposed via node APIs; `fall` is reserved for the fallback mechanism. These distinctions are visible in the WIT interface and enforced at compile time and at runtime, preventing accidental mutation along read-only paths and clarifying the call graph for tools and auditors. [26], [27]

Sigil's storage system utilizes a hierarchical key space with `DotPathBuf` and traits like `Store`/`Retrieve` that provide compile-time type checking for stored data, with fine-grained access to nested fields that allows contracts to read or update a subfield without materializing an entire aggregate. As a result, the resource model closely tracks the operations performed: updating an allowance within a nested map touches only the relevant path and charges gas accordingly. While similar in concept to storage systems like ink!'s, Sigil's implementation focuses on providing ergonomic storage handling through nested maps and `derive` macros (`Root`, `Wrapper`) for typed accessors. This approach helps prevent key collisions or data type errors. In comparison, CosmWasm developers must manually select unique string prefixes for each `Map` and ensure they do not collide; [33] NEAR serializes its entire state into a single Borsh blob, making migrations fragile; [29] Alkanes accesses state via string-based `StoragePointer`s and generic `get_value`/`set_value` calls. [34] Sigil's `DotPathBuf` and `Store`/`Retrieve` traits provide nested maps and typed accessors without manual key management. In systems that serialize full objects to and from a key–value store (e.g., JSON or Borsh values under string-keyed prefixes), developers must carefully manage namespaces, avoid collisions, and pay the cost of encode/decode round-trips even for localized changes; reasoning about gas then depends as much on serialization strategy as on business logic. By contrast, Sigil's typed accessors and path-scoped operations reduce these failure modes and make costs easier to estimate ex ante. [29], [31]

All runtime primitives in Sigil are abstracted behind Rust traits with multiple backends, so the same contract logic runs unmodified against an in-memory implementation for unit tests and against the on-chain runtime for deployment. This arrangement encourages pure, reusable libraries and enables fast, deterministic testing of multi-contract scenarios using exactly the interfaces exercised in production. Other ecosystems provide bespoke harnesses or mocks whose APIs diverge from on-chain execution, which can allow integration defects to surface late. With Sigil, a team validating an auction's settlement rule can simulate bidders, calls, and storage transitions entirely in memory, then deploy the identical code with the runtime backend, reducing the gap between test behavior and mainnet behavior.

2.2.3. *Modularity.*

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 `decimal`—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. [28], [31], [35], [36] Indeed, Sigil's use of Wasmtime locally makes it trivial to develop contracts in the exact same runtime as on-chain.

To perform a cross-contract call, Sigil provides a Rust macro `import!` which parses a contract's WIT and generates a convenient interface for use in the calling contract. For example, a contract can import an arithmetic contract with:

```
import!(name = "arith", path = "arith/wit");

let result = arith::eval(ctx, value1, arith::Op::Sum(arith::Operand { y:
value2 })).value;
```

This tells Kontor to import the interface from `arith.wit` (the IDL for the arithmetic contract). The macro generates a Rust module (named `arith`) with functions and types matching the contract's interface. Once imported, calling the foreign contract is as simple as calling a Rust function. This is a significant improvement over other systems where cross-contract calls are more manual:

- CosmWasm requires constructing a `WasmMsg::Execute` message with a target address and a JSON-encoded payload, executed atomically within the same transaction with optional `SubMsg` replies. [31], [37] There's no enforced standard interface, so developers rely on shared libraries or copied message definitions.
- In Substrate, cross-contract calls can be facilitated by contract reference types, which generate a Rust stub. However, this is not a chain-wide standard and requires both contracts to use the same ink! codebase for the interface. [36]
- NEAR supports cross-contract calls via asynchronous promises, but the correct formatting of the call is up to the developer and isn't checked until runtime. [35]
- Alkanes embeds a block/tx identifier and numeric opcode in a cellpack and shifts bytes off an input stack. [25]
- Ethereum/EVM (though not Wasm-based) uses off-chain ABI definitions to encode function selectors and parameters into a byte blob, and a wrong encoding will only fail at runtime. [32]

Kontor's cross-contract model provides a higher level of interoperability and safety through a standardized IDL and stub generation, whereas others rely on manual or ad-hoc methods. In general, Kontor's approach with WIT and WAVE emphasizes compile-time correctness and explicit IDLs, which can prevent many integration bugs. [26] By contrast, other frameworks often rely on implicit or off-chain agreement of interfaces (CosmWasm's message types, NEAR's method names, Solana's IDL) or simply trust the developer to get it right (Ethereum's interface usage).

2.2.4. *Upgradeability.*

Sigil supports upgradeability through explicit component composition and a `fallback` hook that routes unmatched calls to successor components while preserving strict storage isolation. This approach avoids executing foreign code within a contract's storage, thereby eliminating the `delegatecall`-style hazards common in EVM proxy patterns—layout coupling, clashing storage slots, and upgrade keys with broad authority. The proxy intercepts all requests and forwards them to the implementation contract, where implementations own their state, allowing each version to have a different state schema and reference predecessors for lazy state migrations. In practice, when a token interface evolves (e.g., from fixed balances to interest-bearing balances), deployers publish a new component and direct unresolved calls via `fallback`; the new component reads prior state through typed cross-contract calls and performs staged migrations under whatever governance the application specifies. By contrast, EVM proxies rely on `delegatecall` into the caller's context, where even innocuous refactors can corrupt state; CosmWasm/NEAR migrations run as bespoke entrypoints with manual schema coordination and runtime checks rather than through a chain-wide typed interface, and NEAR's documentation warns that modifying stored structures can raise a "Cannot deserialize the contract state" error. [38] Sigil's fallback pattern avoids such issues by letting each version manage its own state schema. Because versioning, upgrades, and authorization can be implemented in userspace, Sigil supports as much complexity as necessary to support safe contract upgrades—for example, DAO-based voting procedures—unlike current-generation Wasm blockchain languages, which have a key-based primitive upgrade mechanism, giving users no guaranteed protection from malicious upgrades. Sigil's model makes upgrade flows auditable, explicitly routed, and type-checked end-to-end. [29], [31], [32]

2.2.5. *Deterministic Execution.*

Kontor runs contracts in Wasmtime, a modern Wasm engine, with non-deterministic features turned off. This aligns with best practices for blockchain execution, where determinism is crucial for all indexers to achieve the same results. The target architecture is `wasm32-unknown-unknown`, which disables a large number of features that can lead to non-deterministic behavior, including for instance file-system access. In addition, threading and SIMD are disabled and floating-point NaN values canonicalized.

```rust
let mut config = wasmtime::Config::new();
config.async_support(true);
config.wasm_component_model(true);
config.wasm_threads(false);
config.wasm_relaxed_simd(false);
config.cranelift_nan_canonicalization(true);
```

All timeouts and resource limits are deterministic by enforcing static memory allocation ahead of time. [39] Finally, all custom functions and resources provided via the runtime, including transitive dependencies, are audited for determinism directly.

2.3. **File Persistence Protocol.**

2.3.1. *Background.*

Bitcoin is designed as a state machine for decentralized agreement on state transitions (primarily payments), but it is not optimized for storing large quantities of stateless data. Indeed, while the Bitcoin protocol offers miners strong economic incentives for honest behavior, it provides no direct incentives at all for non-miners to run full nodes. Even a very large mining pool need run only a single full node, and then solely to accept Bitcoin transaction fees. There is certainly no incentive for Bitcoin full nodes actually to store or even download data that is provably prunable, [40], [41] such as witness signatures, [42] or `OP_RETURN` payloads. Most nodes won't store provably prunable data long-term, and pruned nodes are first-class citizens in Bitcoin Core. [43]

Kontor explicitly addresses this limitation of Bitcoin by establishing a protocol for trustless, high-availability data storage that elegantly complements the state-machine replication of Kontor and Bitcoin.[4] That is, Kontor's architecture implicitly draws a distinction between mutable and immutable data—i.e. between contract state and *other data* that are still valuable and need to be highly available. Kontor, with its smart contracts system on Bitcoin and its Byzantine fault–tolerance file persistence protocol, supports not just one but both, and in a highly integrated fashion.

Most importantly, data stored on the Kontor network is designed to be stored *forever*. That is, the user pays once, and thereafter the network takes care of properly incentivizing the ongoing data storage. This is a critical guarantee for many blockchain use cases, as is evidenced by the widespread adoption of various techniques for storing data on-chain. Still, because of the high cost of on-chain data storage, most NFT data are still stored on centralized systems and regularly disappear when the hosts in question one day decide to take down their servers. [44]

The cost of storing data with Kontor is much lower than that of storing data directly in the Bitcoin blockchain—and the availability guarantees are at least as good. Costs with Kontor are expected to be generally higher than with Filecoin, Sia or Chia because the availability guarantees associated with these are much lower. The uploading user is not necessarily the only user who wants to rely on that data being available. Such systems effectively provide no long-term availability guarantees whatsoever. Their goals are different: they are attempting to use blockchain technology to compete with centralized services (such as Amazon S3) [45] rather than to provide a blockchain-native solution for inherently decentralized applications. In particular, a system is needed which is designed for more moderate amounts of *high-value* data.

[4]Zero-knowledge rollups provide no data availability guarantees at all: they provide cryptographic proofs that the correct state transitions occurred, but they cannot guarantee that the data hosted on the rollups will not simply disappear one day.

A few other projects have had similar goals to those of Kontor, most notably Arweave. [46] Unfortunately, the Arweave protocol is simply incomplete in a number of critical ways. In particular, Arweave's tokenomics do not incentivize the perpetual storage of data uploaded to the network: economic incentives for ongoing data storage are taken from an "endowment pool" that one can only hope will be sufficient. The Arweave Yellow Paper does not even purport to offer a solution.

> *"While the Arweave's mechanism design is generally engineered to promote adaptivity to new circumstances, the core Arweave team does not expect that the network as it is currently formulated will continue to produce blocks in true perpetuity. This does not, however, mean that we expect that the information stored inside the weave will be lost after the final block is mined. It is our expectation that when eventually a permanent information storage system more suited to the challenges of the time emerges, the Arweave's data will be 'subsumed' into this network. After the mining of the final block, the financial incentive mechanisms for data preservation will subside and give way to social incentives for data preservation. This effect will likely be compounded by the exceptionally low cost of storing the data from the network, due to its decreasing relative cost over time."* — Arweave Yellow Paper [46, p. 21]

Even more problematically, Arweave nodes are not punished in any way for failing to store the data that they have committed to. Finally, because Arweave—like Filecoin, Sia and others—constitutes its own separate blockchain network, Bitcoin users must purchase additional cryptocurrency off-chain, with entirely incompatible tooling, in order to use the network.

Kontor uses its native token (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 is based on known schemes for compact proofs of retrievability. [47]

2.3.2. *Design Principles.*

2.3.2.1. *Fair Resource Allocation Through Logarithmic Scaling.*

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.

A linear scaling model would create a structural bias against small files. The rewards for storing a multi-gigabyte file would vastly exceed those for storing a few-kilobyte file containing a valuable private key, making the latter economically irrational for nodes to support, even when its security is just as critical.

Logarithmic scaling corrects this distortion. Its key properties are:

- **Fair Node Incentives:** It ensures that the return on capital for a storage node is roughly constant across a vast range of file sizes. This makes both small and large files similarly attractive to store, ensuring a uniform level of security for all data on the network.

- **Equitable User Costs:** The one-time storage fee paid by the user is also scaled logarithmically. This results in a steeply regressive cost-per-byte, removing economic barriers to securing small but critical files while still ensuring that users who consume more physical storage resources pay a proportionally larger (albeit sublinear) total fee.

This design is a deliberate choice to engineer an economy that prioritizes data security irrespective of data size.

2.3.3. *Protocol Description.*

2.3.3.1. *Summary.*

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.

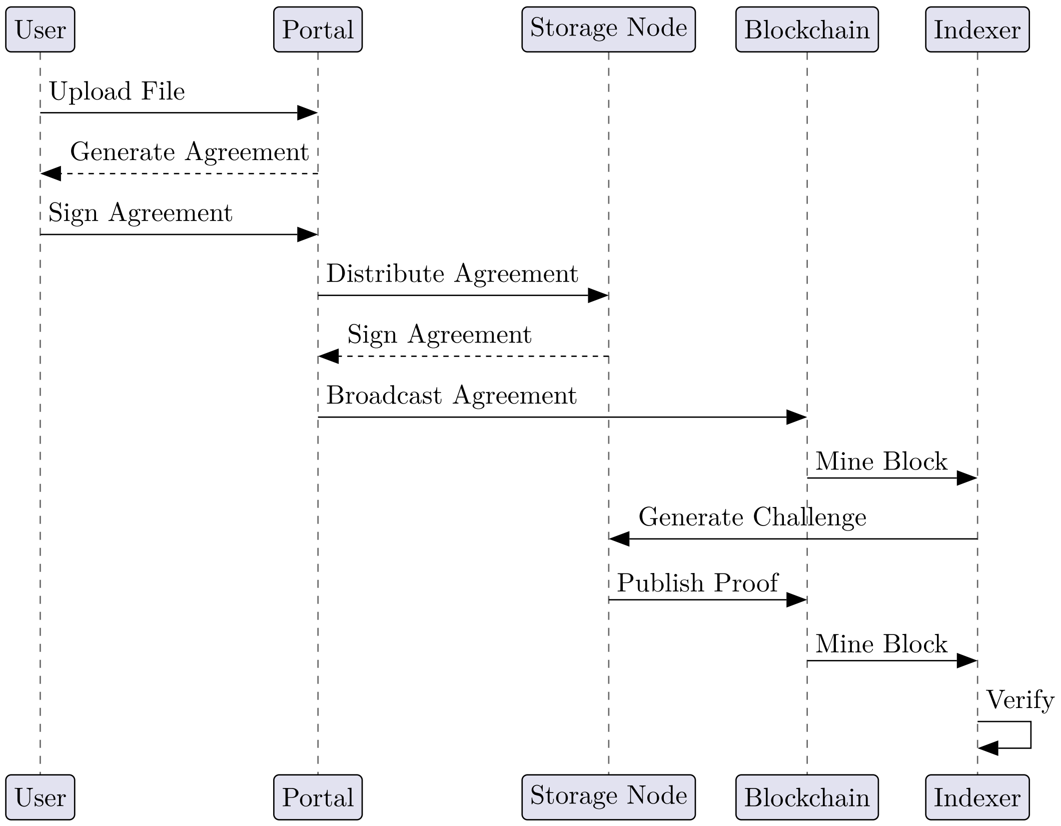
The Kontor protocol uses SNARK-based proof-of-retrievability system to ensure data is stored correctly. This system is built using Nova recursive SNARKs, which allow for the creation of extremely efficient and compact proofs, and unlike many other SNARK schemes, Nova's security is based on the Inner-Product Argument (IPA) and therefore allows for a fully transparent setup. [48] A storage node demonstrates that it possesses the data it has committed to by answering deterministic challenges against a Merkle tree commitment.

A **verifier** (any Kontor indexer) can audit arbitrarily large files by examining only:

- The public Merkle root of the file (recorded in the storage agreement),
- The random challenge (derived from Bitcoin block hashes), and

- A small, constant-size compressed SNARK (the proof, produced by the storage node).

Crucially, the verification is stateless and efficient. The final compressed proof is only about 10 kB, and verification takes approximately 50 milliseconds, regardless of the file's size or the number of challenges answered. This makes the construction highly suitable for on-chain verification with minimal overhead. The system uses the Poseidon hash function for building Merkle trees, ensuring consistency between on-chain and off-chain computations. [49] The reference implementation is built using the `arecibo` library for its folding scheme. [50]



The Kontor file persistence protocol allows any Kontor user account to act as a "storage node"; there is no separate data storage network. For the purpose of explanation, however, we distinguish between users and storage nodes, where the former are users that simply upload data to the network and the latter are those that commit to storing user data.

2.3.3.2. *File Preparation.*

On their local machine, the user prepares the file for upload. This preparation phase turns ordinary data into a fault-tolerant, self-authenticating artifact that can be efficiently audited. The process involves two main steps:

1. **Partitioning**: The file is partitioned into fixed 31-byte symbols. This symbol size is determined by the Pallas scalar field constraint (255 bits), ensuring each symbol can be directly encoded as a field element.
2. **Erasure Coding**: The symbols are encoded using Reed-Solomon over GF(2^8) in a multi-codeword structure. Each codeword encodes 231 data symbols and generates 24 parity symbols (10% overhead), totaling 255 symbols per codeword. Large files are partitioned into multiple independent codewords. This provides fault tolerance: the file can be reconstructed if each codeword retains ≥90% of its symbols.
3. **Merkle Tree Commitment**: Each 31-byte symbol is directly encoded as a field element (via little-endian byte representation, with no hashing) and becomes a leaf in a Merkle tree constructed using Poseidon. The root serves as a compact cryptographic commitment. This direct encoding ensures proof-of-retrievability: a prover cannot generate valid proofs without storing the actual symbol bytes.

2.3.3.3. *File Agreement Creation.*

A file storage operation is memorialized in a `FileAgreement` Kontor transaction. For this transaction to be valid, it must be signed by the user and an initial set of greater than n_{min} storage nodes.

The user pays a one-time storage fee v_f that is entirely burned. This fee is not a bid, but is calculated deterministically as a fraction of the per-node base stake requirement for the file at the time of its creation.

$$v_{f(t)} = \chi_{\text{fee}} \cdot k_{f(t)}$$

where χ_{fee} is a fixed protocol parameter and $k_{f(t)}$ is the per-node base stake (defined below). The file's perpetual emission weight ω_f, which determines its share of rewards, is based on its size and immutable creation rank: $\omega_f = \frac{\ln\left(s_f^{\text{bytes}}\right)}{\ln(1+\text{rank}_f)}$. This ensures that files created when the network is smaller receive higher rewards in perpetuity.

The protocol uses a pooled stake model. Instead of escrowing funds per agreement, each node maintains a single stake balance k_n that must be sufficient for all files it has committed to store. The required stake for a node n is the sum of the per-file base stake requirements, amplified by a dynamic factor that disincentivizes Sybil attacks.

- **Per-file base stake**: The base stake $k_{f(t)}$ is dynamic, scaling with the file's proportional weight and the network's overall size: $k_{f(t)} = \left(\frac{\omega_f}{\Omega(t)}\right) \cdot c_{\text{stake}} \cdot \ln\left(1 + \frac{|\mathcal{F}(t)|}{F_{\text{scale}}}\right)$. This ensures capital costs adjust to network conditions.
- **Total required stake**: $k_{\text{req}}(n,t) = \left(\sum_{f \in \mathcal{F}_n} k_{f(t)}\right) \cdot \left(1 + \frac{\lambda_{\text{slash}}}{\ln(2+ |\mathcal{F}_n|)}\right)$

A node can only join a new agreement if its current stake k_n is sufficient to meet the new, higher requirement.

The protocol ensures that the number of nodes storing any file does not drop below a minimum threshold n_{min}.

2.3.3.4. *Challenges.*

With the mining of each Bitcoin block, Kontor indexers deterministically derive a set of **challenges** from the block hash. The number of files challenged per block, $\theta(t)$, scales linearly with the total number of files in the network to maintain a constant target challenge rate per file per year, C_{target}.

$$\theta(t) = \frac{C_{\text{target}} \cdot |\mathcal{F}(t)|}{B}$$

where B is the expected number of Bitcoin blocks per year. This ensures that the per-file challenge probability, $p_f = \frac{C_{\text{target}}}{B}$, remains constant and predictable as the network grows, providing consistent security guarantees without introducing adaptive complexity.

For each file selected for a challenge, the protocol randomly selects one of its storing nodes, $\mathcal{N}_f$, to produce a proof. The challenge itself consists of a set of randomly selected leaf indices from the file's Merkle tree. The indices are derived deterministically from the block hash, ensuring they are unpredictable beforehand but publicly verifiable afterward.

The protocol's challenge mechanism is designed to provide a consistent level of security for all files, regardless of their size. This is achieved by sampling a fixed number of sectors (Merkle leaves) from each challenged file. The probability of detecting data loss depends only on the fraction of missing data, not the absolute file size. This ensures uniform security guarantees across the network. Detection of extremely small quantities of missing data is unnecessary, as the file's erasure coding ensures it is still fully recoverable in this case.

2.3.3.5. *Proofs and Proof Verification.*

All storage challenges must be answered within a predefined submission window (a certain number of Bitcoin blocks). For each challenge, the selected storage node generates a set of recursive SNARKs. Each step in the recursion proves:

1. The correct deterministic calculation of a challenged leaf index.
2. Knowledge of a valid Merkle path from that leaf to the public Merkle root.
3. The correct update of a cryptographic hash chain, which links the steps together and prevents replay attacks.

These individual proofs are then efficiently combined into a single, constant-size `CompressedSNARK`. This final proof is broadcast to the Bitcoin network.

When the proof is included in a block, all Kontor indexers act as verifiers. They check the compressed SNARK against the public inputs, which include the file's Merkle root and the challenge seed. If the proof is valid, the node has successfully demonstrated possession of the file. If the proof is invalid or not submitted in time, the node is immediately slashed.

2.3.3.6. *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 triggers a stake-sufficiency-restoration process. The node is gracefully removed from file agreements (with a penalty of $k_f \times \lambda_{\text{slash}}$ deducted from its stake for each involuntary exit) until its stake is sufficient again. If this is not possible without violating minimum replication on its remaining files, its entire remaining stake is burned, and it is removed from all agreements.

2.3.4. *Storage Node Economics.*

A detailed description of the protocol's economic model can be found in a separate document. This section provides a high-level overview.

Storage nodes are modeled as rational, profit-seeking agents. Their decisions to join, leave, or remain in file agreements are governed by their expected profitability.

2.3.4.1. *Joining and Leaving Agreements.*

Following the creation of a file agreement, other nodes may join to store the file and earn rewards. A node can join in one of two ways:

1. **Unsponsored Join**: If a node can acquire the file data through off-chain means (e.g., from the original user), it can join directly, provided it has sufficient stake.
2. **Sponsored Join**: If a new node (an "entrant") cannot obtain the data, it can enter into a formal, on-chain sponsorship agreement with an existing storer (a "sponsor"). The sponsor provides the file data in exchange for a temporary commission on the entrant's future rewards. This market-driven mechanism ensures that there is always a permissionless path to join any file agreement.

A node's ability to leave an agreement, however, is governed by powerful mechanisms designed to ensure data preservation. The protocol combines preventative disincentives with corrective incentives. Departure is programmatically forbidden if a file's replication level $|\mathcal{N}_f|$ is at or below the minimum, $n_{\min}$. If $|\mathcal{N}_f| > n_{\min}$, a departing node must pay a leave fee, $\varphi_{\text{leave}}(t) = k_{f(t)} \cdot \left(\frac{n_{\min}}{|}\mathcal{N}_f|\right)^2$, which is burned. This fee scales with the current base stake requirement and the inverse square of the number of participants, becoming punitively high as a file becomes more vulnerable. This acts as a preventative disincentive.

The corrective incentive arises if a file becomes under-replicated. The per-node reward, $r_{\text{n|f}}(t) = \varepsilon_f \frac{t}{|}\mathcal{N}_f|$, increases hyperbolically for the remaining nodes. This reward magnification attracts new nodes to join and "repair" the file's replication. Together, these mechanisms create robust defenses against data loss.

2.3.4.2. *Revenue and Costs.*

Revenue for storage nodes comes from two primary sources:

1. **Storage Rewards**: The primary revenue stream is the ongoing KOR emissions generated by each file. Total emissions for a block, $\varepsilon(t)$, scale with the total KOR supply and are adjusted by a dynamic emission multiplier, $\alpha(t)$, which is a function of the network's time-averaged replication factor. A file's share of these total emissions, $\varepsilon_f(t)$, is proportional to its emission weight relative to the network's total weight: $\varepsilon_f(t) = \varepsilon(t) \cdot \left(\frac{\omega_f}{\Omega(t)}\right)$. The emissions for a file are distributed equally among all nodes storing it ($r_f = \frac{\varepsilon_f}{|}\mathcal{N}_f|$), taking into account sponsorships agreements (Section 2.3.4.1).
2. **Slashed Funds**: Nodes receive a share of the KOR slashed from other nodes that fail challenges for the same files.

Costs for storage nodes include:

1. **Capital Costs**: The opportunity cost of the total KOR a node must stake (k_n), which cannot be used for other purposes.
2. **Operational Costs**: The Bitcoin transaction fees paid to submit proofs on-chain. Physical data storage costs are considered negligible in comparison to capital and operational costs.

2.3.4.3. *Equilibrium and System Stability.*

The interplay of revenue and costs creates a natural equilibrium.

- **Crowding Effect**: As more nodes join an agreement, the per-node share of rewards decreases, making it less attractive for new nodes to join. This creates a stable equilibrium for file replication.
- **Repair Incentive**: If a file becomes under-replicated (e.g., due to slashing), the per-node rewards for the remaining nodes increase hyperbolically. This creates a powerful incentive for new nodes to join and "repair" the file's replication.
- **Sybil Resistance**: The protocol's pooled stake model, specifically the dynamic stake factor, makes it more capital-intensive to run many small nodes than one large one. This incentivizes operators to consolidate their stake, which mitigates Sybil attacks and aligns capital efficiency with network health.

This economic design ensures that nodes are robustly incentivized to join and remain in file agreements, guaranteeing the long-term persistence of user data.

2.3.5. *Security Analysis.*

The protocol's economic design has been engineered to be robust against a variety of attacks from rational, profit-seeking adversaries. (A complete security analysis may be found in a separate document.)

2.3.5.1. *Attacks on Storage Provision.*

- **Disk-Sharing Attack:** A common Sybil attack is disk-sharing, where an adversary uses multiple node identities to collect rewards for storing multiple copies of a file while only storing one physical copy. The goal is to profit from saved physical storage costs. The protocol mitigates this by ensuring a node's **capital costs** (from staking and proving) dominate its

physical storage costs. An attacker's savings on physical storage are insignificant compared to the full capital costs they must bear for each Sybil identity, rendering the attack unprofitable.

2.3.5.2. *Attacks on Market Mechanics.*

- **Sybil Attack (Risk Compartmentalization):** A more sophisticated operator might use Sybil nodes not to save storage costs, but to compartmentalize risk. By splitting a large portfolio of files across many identities, an operator can shield the bulk of their stake from a catastrophic loss caused by a systemic operational failure affecting one identity. The protocol counters this with a **dynamic stake factor** $(1 + \frac{\lambda_{\text{slash}}}{\ln(2+|\mathcal{F}_n|)})$. This factor imposes a significant capital premium on nodes with fewer files. Consequently, the most capital-efficient way to operate is to consolidate all files under a single identity. This aligns operator incentives with network health by making Sybil strategies economically irrational at any significant scale.
- **Wash-Trading:** An attacker might store their own data to farm rewards. This is prevented by the one-time user fee, v_f, which is entirely burned upon file creation. The fee ratio χ_{fee} is calibrated to ensure that this non-recoverable upfront cost is greater than the net present value (NPV) of the future reward stream the attacker could hope to gain. Because the user fee scales with the same parameters as the file's reward potential, this makes wash-trading unprofitable by design.

2.4. **Macroeconomics.**

The KOR token serves dual purposes: incentivizing perpetual data storage and fueling smart contract execution. Unlike Bitcoin, KOR is fundamentally inflationary by design, with new tokens continuously minted and distributed to storage nodes as rewards for maintaining file agreements.

The protocol incorporates a sophisticated monetary policy designed to maintain the long-term health and stability of the network while controlling inflation as it scales. This policy combines three key mechanisms: emission tapering, dynamic rate adjustment, and strategic burn mechanisms that work together to balance the inflationary pressures from storage incentives with deflationary pressures from network usage.

2.4.1. *Emission Tapering and Network Maturity.*

The protocol implements a "grandfathering" mechanism where files receive a fixed emission weight ω_f at creation time, based on their size and unique creation rank: $\omega_f = \frac{\ln\left(s_f^{\text{bytes}}\right)}{\ln(1+\text{rank}_f)}$. This creates a powerful tapering effect on total emissions as the network grows. Files created when the network is young (i.e., have a low rank) receive higher reward shares in perpetuity.

This design achieves multiple objectives:

- **Controlled Emission Growth**: As the network grows, the total emission weight $\Omega(t)$ increases, but the contribution of new files diminishes. This prevents the total KOR emissions, which are proportional to total supply, from causing runaway inflation.

- **Early Adopter Incentives**: Files stored early in the network's history become increasingly valuable to maintain relative to later files, creating incentives for early participation.
- **Sustainable Growth**: New files remain profitable to store even at massive scale because their dynamic stake requirements and operational costs scale down proportionally with their rewards.

2.4.2. *Dynamic Health-Based Adjustment.*

Network health is measured by $\overline{R}(t)$, the emission-weighted average replication level across all files: $\overline{R}(t) = \frac{\sum_{f \in F} \omega_f \cdot |\mathcal{N}_f|}{\sum_{f \in F} \omega_f}$. This metric gives more weight to older, higher-value files in determining overall network health.

The global emission rate $\alpha(t)$ adjusts continuously based on the network's health buffer, the difference between the current average replication $\overline{R}(t)$ and the minimum required replication n_{min}. The rate is designed to be stable when the network is healthy but to increase rapidly as replication approaches the minimum:

- If the network is healthy ($\overline{R}(t) > n_{\text{min}}$), the emission rate is a baseline rate plus an amount inversely proportional to the health buffer. As the buffer shrinks, rewards increase.
- If the network becomes critically under-replicated ($\overline{R}(t) \leq n_{\text{min}}$), the emission rate immediately jumps to its maximum configured value, creating the strongest possible incentive for repair.

This creates a counter-cyclical feedback loop that stabilizes the network against both internal dynamics and external market shocks.

2.4.3. *Burn Mechanisms and Long-Term Stability.*

KOR's dual role—serving both storage incentivization and smart contract execution—creates complementary deflationary mechanisms that evolve with the network's lifecycle:

1. **Storage Fees**: Users pay a one-time fee v_f that is entirely burned, creating deflationary pressure proportional to network growth.
2. **Slashing**: A portion β_{slash} of all slashed stakes is burned, removing KOR from circulation when nodes fail to meet their obligations.
3. **Leave Fees**: Nodes pay a fee to exit agreements, which is burned to discourage churn.
4. **Smart Contract Execution**: Most critically, gas fees from general-purpose smart contract execution are burned. This provides a burn mechanism that scales with network usage rather than file storage.

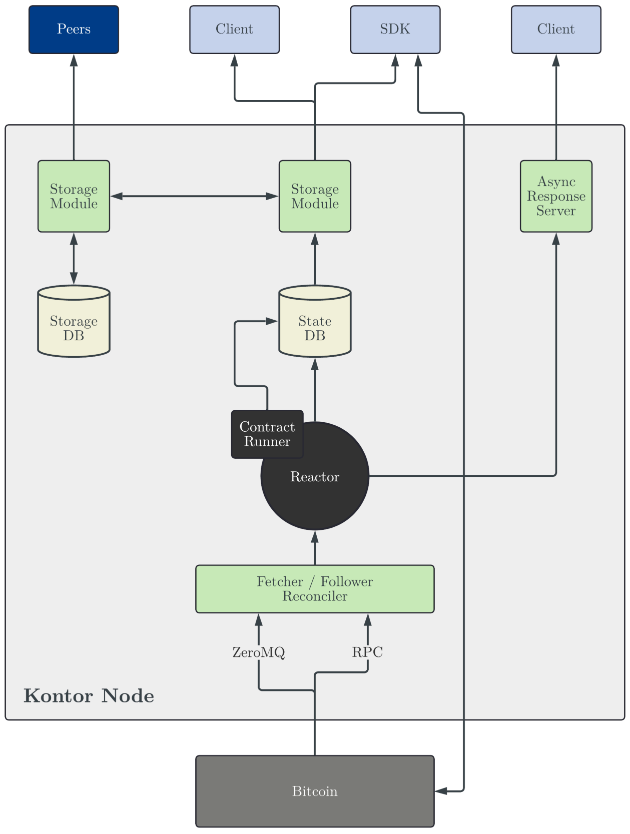
This architectural decision is fundamental to KOR's economic sustainability. As the file storage network matures and growth slows, storage-related burns naturally diminish. However, smart contract activity—the primary use case for KOR—grows independently and provides an ever-increasing source of deflationary pressure. The storage protocol serves as a critical infrastructure layer that bootstraps the network and provides perpetual data availability for smart contracts, while the smart contract ecosystem becomes the primary driver of KOR demand and burns over time. This ensures that even as storage emissions stabilize due to the tapering mechanism, the inflation rate remains controlled through burns from a vibrant smart contract ecosystem.

2.4.4. *Economic Equilibrium.*
The interplay of these mechanisms creates a robust economic equilibrium:

- Early in the network's life, high emission weights attract storage nodes and bootstrap the network.
- As the network grows, emission tapering prevents runaway inflation while smart contract burns begin to offset new emissions.
- In maturity, smart contract execution burns balance emissions even as the network continues to incentivize perpetual storage.

3. Indexer Reference Implementation

The reference implementation of the Kontor protocol is written in clean, modern asynchronous Rust. It implements a central `reactor` module that runs a single-threaded event loop and maintains exclusive control of the only database writer. All auxiliary processes, i.e. API server, fetcher/follower, etc. communicate with `reactor` via Go-style message passing using `tokio` and `mpsc`. The `reactor` receives block, rollback, and mempool events from the Bitcoin follower module, parses transactions, and executes language functions that trigger database updates.



3.0.1. *Database Tables.*

3.0.1.1. *Contract State Table.*

```
contract_id, publisher, tx_id, height, path, value, deleted - unique (id,
height, path)
```

```
struct NestedStruct {
    score: u64,
    last_updated: u64,
}

#[kontor::storage]
struct GameState {
    owner: String,
    total_players: u32,
    players: Map<String, NestedStruct>,
}
```

id	publisher	height	tx_id	path	value	deleted
1	abc123	100	1001	owner	“0x123…”	false
1	abc123	100	1001	total_players	2	false
1	abc123	100	1001	players.alice.score	100	false
1	abc123	100	1001	players.alice.last_updated	1710446820	false
1	abc123	100	1001	players.bob.score	50	false
1	abc123	100	1001	players.bob.last_updated	1710446800	false
1	abc123	101	1002	players.alice.score	150	false
1	abc123	101	1002	players.alice.last_updated	1710446900	false
1	abc123	101	1002	players.bob.score	50	true
1	abc123	101	1002	players.bob.last_updated	1710446800	true
1	abc123	101	1002	total_players	1	false

Nested structs and maps are flattened into paths. The map key becomes part of the path (e.g. `players.alice.score`). At height `101`, Alice’s score is updated to `150` and Bob’s entry is marked as `deleted`. Note that the `total_players` count is also updated to reflect Bob’s removal. The `deleted` flag in the database schema primarily serves as a convenience feature for contract authors working with Map data structures. It is essentially a special value indicating that a value is no longer valid within the contract’s context. Other state properties would typically be modeled as `Option<T>` types, where "deletion" is represented by setting the value to `None`.

The following contract code shows the state changes from the above example.

```
// Changes at height 101
game.players["alice"].score = 150;
game.players["alice"].last_updated = 1710446900;
game.players.remove("bob");  // Delete Bob's entry
game.total_players = 1;      // Update player count
```

The contract system translates these contract state changes into the following database operations:

```
game.players["alice"].score = 150;
// - Insert new row with updated score
// - contract_id=1, path="players.alice.score", value=150, height=101

game.players["alice"].last_updated = 1710446900;
// - Insert new row with updated timestamp
// - contract_id=1, path="players.alice.last_updated", value=1710446900,
height=101

game.players.remove("bob");
// - Insert new rows marking bob's data as deleted
// - contract_id=1, path="players.bob.score", value=50, height=101,
deleted=true
// - contract_id=1, path="players.bob.last_updated", value=1710446800,
height=101, deleted=true

game.total_players = 1;
// - Insert new row with updated player count
// - contract_id=1, path="total_players", value=1, height=101
```

This results in the following SQL queries, in which the node fetches the value at the path that has the highest height and is not `deleted`.

```sql
SELECT s.value
  FROM state s
  WHERE s.id = ?1
    AND s.deleted = false
    AND s.height = (SELECT MAX(s2.height)
                      FROM state s2
                      WHERE s2.id = s.id
                        AND s2.path = s.path)
```

By loading state data only when needed, the system minimizes unnecessary database reads. This approach significantly reduces gas costs, as users only pay for the specific data their transaction requires, rather than paying for loading the entire contract state. Additionally, this lazy loading strategy keeps the overall system more responsive by avoiding unnecessary database operations.

Kontor smart contracts are backed by SQLite3. SQLite3 is the most widely deployed database in the world [51] and one of the best tested. [52] The querying features of SQL, such as `joins` and `unions` allow the Kontor reference node to perform the following operations performantly:

- Query historical states by combining multiple versions of contract data
- Handle mempool state by union-ing temporary state changes with confirmed state
- Implement complex queries needed for contract interaction and cross-contract calls
- Support efficient filtering and aggregation for API endpoints
- Store data efficiently by eliminating the necessary data duplication required to efficiently read state from a key-value store
- Simplify the state interface injected into the Wasm runtime

Note: traditional SQL databases are not well-suited for storing and manipulating ordered lists. In order to insert or remove items from the middle of a list, you need to update the indices/positions of all subsequent items—similar to shifting elements in an array. The Kontor language allows users to store limited-size lists in contract state that are read from and inserted into the database in full.

3.0.1.2. *Blocks Table.*

```
height, hash - unique(height), unique(hash)
```

This table links block heights to their corresponding block hashes. This mapping enables the follower to detect blockchain reorganizations. Like the `txid` in the transactions table, `hash` values may be compressed to save space.

3.0.1.3. *Transactions Table.*

```
id, height, txid - unique(id), unique(txid)
```

While a `transactions` table is not strictly necessary for the core functionality of the system, it serves an important purpose for “explorer” applications that want to trace the history of state changes. This enables transparency by allowing anyone to examine which Bitcoin transactions resulted in specific updates to the system state. The `txid` field stores the actual Bitcoin transaction hash, while `id` is a smaller, internal integer identifier that is referenced in the contract state table. Using `id` instead of the full transaction hash in the contract state table helps reduce storage requirements and improves query performance, since integer joins are typically more efficient than string joins.

3.0.1.4. *Checkpoints Table.*

```
height, hash - unique(height), unique(hash)
```

The Kontor reference implementation maintains a running hash of the contract state data by storing and updating a hash value in the database as each new row is added. Every N blocks, the Kontor indexer creates a new checkpoint entry containing this accumulated hash. This makes it possible to efficiently verify the integrity of historical state data and provides convenient snapshots for system recovery or state verification.

As new state entries are added, the node combines their values (`contract_id`, `path`, `value`, etc.) into the running hash and continuously updates a single row in the database. When it reaches the N-block interval, that row remains unchanged and the node begins updating a new row for the next interval.

```
new_hash = hash(hash(new_contract_state_row) ++ old_hash)
```

3.0.2. *API and SDK.*

The Kontor Indexer API and SDK provide a convenient interface for application developers to build decentralized apps in the Kontor ecosystem. The API includes an HTTP server for low-level access to contract state and a WebSocket server for monitoring contract events in real time. The SDK wraps this API to mirror the contract code's state object, to simplify contract function calls by abstracting transaction composition, and to offer a high-level event listening system.

3.0.2.1. *HTTP API.*

```
GET /api/contracts/<ContractAddress>/state/<path (i.e. ".value")>?mempool=true
```

The contract state endpoint provides access to path values in the same way that `state.value` does within a contract. The `mempool` query parameter lets consumers choose whether to view the current state based on mempool data or the last processed block.

3.0.2.2. *SDK.*

The SDK is primarily a code generation tool that creates three key components:

1. **State Object**: Generates a TypeScript client that wraps the API for easy state access.

```
// Access contract state
const balance = await client.state.balances.get(address);
console.log(`Current balance: ${balance}`);
```

2. **Events Interface**: Creates an events object that implements a WebSocket client for real-time updates.

```
// Listen for transfer events
client.events.onTransfer((from, to, amount) => {
  console.log(`Transfer: ${amount} tokens from ${from} to ${to}`);
});
```

3. **Contract Functions**: Provides an object containing all contract functions as callable methods.

```
// Call contract function
await client.transfer({
  to: receiverAddress,
  amount: 100
});
```

Under the hood, this routine constructs Bitcoin transactions with embedded Kontor data and broadcasts them to a Bitcoin node. The client must be configured with the user's Bitcoin address to access spendable outputs for covering BTC transaction fees.

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 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.

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, purpose-built for financial applications, 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.

Kontor was designed from the ground up to facilitate the development of rich, innovative financial instruments while maintaining deep integration with Bitcoin and other metaprotocols within the greater Bitcoin ecosystem, complementing the traditional blockchain architecture with a Byzantine fault–tolerant network for persisting high-value stateless data that support these financial functions. Kontor is fundamentally cooperative with Bitcoin. 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. Bibliography

[1] V. Buterin and G. Wood, "Ethereum White Paper," Dec. 2014. [Online]. Available: https://static.peng37.com/ethereum_whitepaper_laptop_3.pdf

[2] Market Memoir Research, "The Crisis That Closed NYSE on Wednesdays." [Online]. Available: https://www.marketmemoir.com/blogs/the-memoir/an-unusual-crisis-paperwork

[3] BitMEX Research, "The OP_Return Wars of 2014 - Dapps Vs Bitcoin Transactions." [Online]. Available: https://blog.bitmex.com/dapps-or-only-bitcoin-transactions-the-2014-debate/

[4] Jonathan Bier, *The Blocksize War: The battle over who controls Bitcoin's protocol rules.* 2021.

[5] P. Vigna, "Bitcoin Rival Launches in Volatile First Day." WSJ, Aug. 01, 2017. [Online]. Available: https://www.wsj.com/articles/bitcoin-rival-launches-in-volatile-first-day-1501790080

[6] V. Buterin, [Online]. Available: https://x.com/agkrellenstein/status/1920863331012792469

[7] Galaxy Digital Research, "Bitcoin Ordinals: A Statistical Overview," 2024. [Online]. Available: https://www.galaxy.com/insights/research/bitcoin-ordinals-statistics/

[8] Nathaniel Popper, "Some Bitcoin Backers Are Defecting to Create a Rival Currency." The New York Times, July 25, 2017. [Online]. Available: https://www.nytimes.com/2017/07/25/business/dealbook/bitcoin-cash-split.html

[9] J. Lopp, [Online]. Available: https://x.com/lopp/status/1919104031114367293

[10] Stacks, “Stacks Documentation.” [Online]. Available: https://docs.stacks.co/

[11] Blockstream, “ How do I peg in BTC to the Liquid Network? What are the different types of participants in the Liquid Network?.” [Online]. Available: https://help.blockstream.com/hc/en-us/articles/900002163803-What-are-the-different-types-of-participants-in-the-Liquid-Network

[12] Victor I. Kolobov, Avihu M. Levy, and Moni Naor, “ColliderVM: Stateful Computation on Bitcoin without Fraud Proofs,” Apr. 10, 2025. [Online]. Available: https://arxiv.org/abs/2504.05239

[13] Mashiat Mutmainnah, “How to leverage Risc Zero's ZKVM to scale Bitcoin.” [Online]. Available: https://risczero.com/blog/how-to-leverage-risc-zeros-zkvm-to-scale-bitcoin

[14] Bitcoin Magazine, “Bitcoin Magazine Editorial Policy on Bitcoin Layer 2s.” Feb. 21, 2024. [Online]. Available: https://bitcoinmagazine.com/press-releases/bitcoin-magazine-editorial-policy-on-bitcoin-layer-2s-l2s

[15] “Cosmos: Introduction to ABCI.” [Online]. Available: https://docs.cosmos.network/main/build/abci/introduction

[16] Ordinals, “Inscriptions.” [Online]. Available: https://docs.ordinals.com/inscriptions.html

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

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

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

[20] C. Bormann and P. Hoffman, “Concise Binary Object Representation (CBOR),” IETF. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8949

[21] Pieter Wuille, “BIP 341: Taproot: SegWit version 1 spending rules.” [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki

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

[23] Bitcoin.org, “Developer Guide - Signature Hash Types.” [Online]. Available: https://developer.bitcoin.org/devguide/transactions.html#signature-hash-types

[24] A. Krellenstein, "[ANN][XCP] Counterparty - Pioneering Peer-to-Peer Finance - Official Thread." [Online]. Available: https://bitcointalk.org/index.php?topic=395761.0

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

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

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

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

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

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

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

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

[33] CosmWasm documentation contributors, "Basics – cw-storage-plus." [Online]. Available: https://cosmwasm.cosmos.network/cw-storage-plus/basics

[34] Alkanes documentation contributors, "Building Alkane Contracts." [Online]. Available: https://alkanes-docs.vercel.app/docs/developers/contracts-building

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

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

[37] Archway, "Integrate with smart contracts." [Online]. Available: https://docs.archway.io/developers/smart-contracts/contract-integration

[38] NEAR documentation team, "Updating Contracts." [Online]. Available: https://docs.near.org/smart-contracts/release/upgrade

[39] Bytecode Alliance, "Config in wasmtime." [Online]. Available: https://docs.wasmtime.dev/api/wasmtime/struct.Config.html

[40] Bitcoin Core developers, *p2p, validation: Don't download witnesses for assumed-valid blocks when running in prune mode.* (Feb. 06, 2023). GitHub. [Online]. Available: https://github.com/bitcoin/bitcoin/pull/27050

[41] Bitcoin Core, "Reducing Bitcoin Core's disk usage." [Online]. Available: https://bitcoincore.org/en/doc/reduce-disk-usage/

[42] Bitcoin Core, "Segwit Benefits." [Online]. Available: https://bitcoincore.org/en/2016/01/26/segwit-benefits/

[43] Mike In Space, [Online]. Available: https://x.com/mikeinspace/status/1882864972478427272?s=46

[44] Hamza Salem and Manuel Mazzara, "Hidden Risks: The Centralization of NFT Metadata and What It Means for the Market," 2024. [Online]. Available: https://arxiv.org/abs/2408.13281

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

[46] 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

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

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

[49] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems," 2019. [Online]. Available: https://eprint.iacr.org/2019/458

[50] Microsoft, *Arecibo.* (2024). GitHub. [Online]. Available: https://github.com/microsoft/arecibo

[51] SQLite, "Most Widely Deployed and Used Database Engine." [Online]. Available: https://sqlite.org/mostdeployed.html

[52] "How SQLite Is Tested." [Online]. Available: https://sqlite.org/testing.html