KONTOR SCALABILITY

January 10, 2026

ADAM KRELLENSTEIN
adam@kontor.network

Contents

1 Introduction

Kontor is a next-generation metaprotocol that extends the functionality of the Bitcoin blockchain with rich smart contracts, scalable file storage, low-latency confirmations and a trustless Bitcoin bridge. [1] As a metaprotocol, Kontor fully inherits the security of the Bitcoin mining network, at the cost of also inheriting the binding constraints of Bitcoin's Nakamoto consensus: a block limit of 4,000,000 weight units (WU) and a block time of ten minutes.

Despite these constraints, it is not the case that Kontor transaction throughput is limited to that of Bitcoin itself. Whereas Bitcoin was never designed to support high-throughput applications, the Kontor protocol is able to employ a number of optimizations that qualitatively increase protocol efficiency, both in terms of maximum transactions-per-second and transaction fees. These optimizations, when combined together, allow Kontor to transmit, store and process over 1000 transactions per second without ever requiring users to leave the Bitcoin blockchain:

1. BLS signature aggregation
2. Compact registry IDs
3. Binary encoding
4. Zstd compression

Optimizations 2–4 are completely invisible to users. Direct publication uses a standard Taproot spend secured by Bitcoin Schnorr signatures, so users can publish operations with existing wallets and keys. Batching is opt-in: users sign Kontor operations with BLS

(BLS12-381, as used by the Ethereum Beacon Chain since 2020 [2]), bundlers aggregate these signatures and publish them in a single Bitcoin transaction, and Kontor indexers (not Bitcoin Script) verify the aggregate. Compression is a standard optimization technique used across many blockchains. Ethereum Layer-2 rollups use similar approaches: Arbitrum employs Brotli compression for calldata, [3] while ZK rollups and optimistic rollups often apply compression to reduce on-chain data costs. [4]

2 Baseline

2.1 Transaction Format

Kontor contracts are WebAssembly components with typed interfaces (WITs). A contract call specifies a target contract (identified by name, deployment block height, and transaction index), a function to call (by name), and arguments (serialized as WAVE text). The indexer deserializes the call, loads the contract, executes it in a sandboxed Wasm runtime, and commits state changes.

Each Kontor operation is embedded in a Bitcoin transaction via Taproot witness script:

```
[pubkey] [OP_CHECKSIG] [OP_FALSE] [OP_IF] "kor" [OP_0] [data_pushes...]
[OP_ENDIF]
```

The instruction format:

```
pub enum Inst {
    Publish { gas_limit: u64, name: String, bytes: Vec<u8> },
    Call { gas_limit: u64, contract: ContractAddress, expr: String },
    Issuance,
}

pub struct ContractAddress {
    pub name: String,
    pub height: u64,
    pub tx_index: u64,
}
```

2.2 WAVE Text Format

Operations use WAVE (WebAssembly Value Encoding) text format. The `Integer` and `Decimal` types from `built-in.wit` are 256-bit numbers:

```
record integer {
    r0: u64, r1: u64, r2: u64, r3: u64,
    sign: sign
}

record decimal {
    r0: u64, r1: u64, r2: u64, r3: u64,
    sign: sign
}
```

Example WAVE expressions:

```
// Native token transfer (Decimal type, 18 decimal places)
transfer("a1b2c3d4e5f67890...64chars...", { r0: 1000000000000000000000, r1:
54, r2: 0, r3: 0, sign: plus })

// Pool swap (Integer type)
swap({ name: "token", height: 1, tx-index: 0 }, { r0: 100, r1: 0, r2: 0, r3:
0, sign: plus }, { r0: 95, r1: 0, r2: 0, r3: 0, sign: plus })
```

2.3 Transaction Weight

Bitcoin weight is calculated as: non-witness bytes × 4, witness bytes × 1.

Vanilla P2TR payment (1 input, 2 outputs, key-path spend):

Component	**Bytes**	**Weight**
Non-witness (version, inputs, outputs, locktime)	137	548 WU
Witness (marker, flag, stack count, signature)	67	67 WU
Total	204	**615 WU**

Table 1: Vanilla P2TR payment weight breakdown

TPS: TPS $= 4,000, \frac{000}{615 \cdot 600} \approx 10.8$ (baseline)

Kontor transaction (unoptimized) (1 input, 2 outputs, script-path spend with payload):

Component	**Bytes**	**Weight**
Non-witness (same as vanilla)	137	548 WU
Witness base (marker, flag, control block)	36	36 WU
Kontor envelope (pubkey, opcodes)	42	42 WU
Payload (typical WAVE call)	90–144	90–144 WU
Schnorr signature	64	64 WU
Total	359–413	**770–824 WU**

Table 2: Unoptimized Kontor transaction weight breakdown

TPS: TPS $= 4,000, \frac{000}{800 \cdot 600} \approx 8.3$ (0.7× vanilla)

An unoptimized Kontor transaction is slightly heavier than a vanilla Bitcoin payment. The payload itself (90–144 WU) is a minority of the total cost; the majority is fixed overhead from the Bitcoin transaction structure.

3 Optimizations

The optimizations form an interdependent stack: each technique enables or amplifies the others.

3.1 BLS Signature Aggregation

Bitcoin transaction overhead dominates per-operation cost. Bundling many operations into one Bitcoin transaction amortizes this overhead across many operations. A batch of N operations pays the fixed cost once, reducing per-operation overhead from ~680 WU to ~$\frac{680}{N}$

WU. The bundle itself is published as a normal Taproot spend (Schnorr-signed by the bundle publisher), but each operation still requires an authorization signature from its signer.

Without signature aggregation, you must include N per-operation signatures (for example, Schnorr signatures) at 64 bytes each, which can dominate the payload. BLS signatures have the property that N signatures on N distinct messages can be combined into a single 48-byte aggregate, verifiable by Kontor indexers. BLS is unforgeable under the co-CDH assumption. Users may derive a BLS keypair from their seed phrase (via a Kontor-specific derivation path), and sign operations using dedicated Kontor wallet tooling.

Crucially, BLS aggregation is trustless: any party with access to the signed operations can produce a valid aggregate. This means any infrastructure provider can offer bundling services without requiring trust from users—users sign their individual operations, and bundlers compete to include them efficiently.

Users have two publication paths:

1. **Direct**: the user constructs a Bitcoin transaction containing a single Kontor operation and signs it with a standard Schnorr/Taproot wallet.
2. **Bundled**: the user signs a Kontor operation message with their BLS key and broadcasts it to one or more bundlers. A bundler aggregates many users' operations, signs the resulting Bitcoin transaction with its **publisher key** (the Taproot key that authorizes the spend), and publishes it to Bitcoin.

3.1.1 Censorship resistance

Bundlers can censor only by omission (dropping signed operations). Users mitigate this by broadcasting to multiple bundlers, self-bundling, or falling back to direct publication. Since operations are signed, bundlers cannot forge, edit, or redirect operations—only include or exclude them.

3.1.2 Ordering and MEV-like attacks

A bundler can choose ordering within a bundle and may insert its own operations. MEV-style attacks are no worse than for any other metaprotocol; general mitigations for such attacks are discussed elsewhere.

3.2 Registry

The registry maps long identifiers (64-character hex public keys, contract addresses) to compact numeric IDs.

3.2.1 Capacity

ID Size	Max Entries	Calculation
2 bytes	65,536	2^{16}
3 bytes	16,777,216	2^{24}
4 bytes	4,294,967,296	2^{32}

Table 3: Registry capacity by ID size

Four-byte IDs provide ample capacity for the foreseeable future while reducing identifier overhead from 64+ bytes to 4 bytes—a 16× reduction.

3.2.2 Automatic Registration

When a new public key first signs a Kontor operation, the indexer automatically assigns it the next sequential registry ID. This is deterministic: all indexers process blocks identically, so they assign the same IDs.

3.2.3 Consistency Requirements

Registry state must be identical across all indexers:

- IDs assigned sequentially on first use
- Deterministic: all indexers derive identical mappings from the same blocks
- Reorgs require registry state rollback

3.3 Binary Encoding

With compact identifiers available from the registry, the next optimization is the payload format itself. Naive WAVE encoding includes syntactic overhead (parentheses, commas, field names) that binary encoding eliminates.

3.3.1 Naive WAVE Encoding

```
Inst::Call {
    contract: ContractAddress { name: "token", height: 1, tx_index: 0 },
    expr: r#"transfer("a1b2c3...64chars...", { r0: ..., r1: 54, r2: 0, r3: 0,
sign: plus })"#
}
```

3.3.2 Binary Format

```
struct BinaryCall {
    signer_id: u32,       // Registry ID of caller
    contract_id: u32,     // Registry ID of target contract
    function_index: u16,  // Index into contract's WIT exports
    args: Vec<u8>,        // Postcard-serialized arguments
}
```

Each operation explicitly includes its `signer_id`, which maps the operation to its signer for BLS verification. A batch consists of N `BinaryCall` structs followed by a single 48-byte aggregate signature. The indexer verifies the batch by:

1. Extracting all unique `signer_id` values from the operations
2. Looking up corresponding BLS public keys in the registry
3. Verifying the aggregate signature against all (public key, message) pairs
4. Executing each operation with its designated signer as caller

3.4 Compression

Compression is the final step in the pipeline. After operations are encoded in binary format with registry IDs, Zstd compression exploits redundancy across batched operations. Repeated contract IDs, common function indices, and similar argument patterns all compress well.

4 Benchmarks

4.1 Methodology

The benchmark generates realistic transaction workloads and measures the weight (in WU) required to encode them under different format configurations. For each batch of N operations:

1. **Serialization**: Operations are serialized using Postcard (a compact binary format based on variable-length integer encoding).
2. **Compression**: The serialized batch is compressed using Zstd at level 15 (high compression).
3. **Signature overhead**: BLS aggregate signatures add 48 bytes plus 4 bytes per unique signer in the batch (for signer ID lookup).
4. **Transaction overhead**: Fixed costs include non-witness data (548 WU), Taproot witness base (36 WU), Kontor envelope (42 WU), and push opcodes (1–3 bytes depending on payload size).
5. **TPS calculation**: $\text{TPS} = \frac{\frac{\text{Available block weight}}{\text{WU per operation}}}{600}$, where available block weight is 3.9 MW (reserving 100 KW for non-Kontor transactions).

Results use a fixed random seed for reproducibility. Ranges in the tables reflect variation across six scenarios.

4.2 Scenarios

The benchmark suite runs six workload scenarios:

- **Network sizes**: 1K, 100K, and 1M registered users
- **Operation mixes**: Realistic (52% transfers), transfer-heavy (80%), DeFi-heavy (35% AMM swaps), uniform
- **User activity**: Top 10% of users generate ~38–48% of operations
- **Temporal bursts**: ~71% of operations occur in bursts (consecutive operations from the same user)
- **Signer diversity**: Varies by network size (higher diversity in smaller networks)

Each scenario generates 50,000 operations across 100,000 registry users and 500 contracts.

4.3 Results

The following table shows how each optimization contributes to throughput at batch size $N = 100$. Improvement factors are relative to vanilla Bitcoin (11 TPS):

Configuration	**WU/op**	**TPS**	**Improvement**
Vanilla Bitcoin P2TR	615	11	baseline
Naive WAVE ($N = 1$)	770–824	8	0.7×
+ BLS batching ($N = 100$)	33–38	172–199	16–19×
+ Registry IDs	21–25	264–305	25–29×
+ Binary encoding + Zstd	15–18	366–443	35–42×

Table 4: Cumulative effect of optimizations at $N = 100$

Batch Size	**WU/op**	**TPS**	**Improvement**
1	696–698	9	1×
10	80–82	79–81	8×
100	15–18	366–443	35–42×
1000	7–10	683–986	65–93×
10000	5–8	771–1295	73–123×

Table 5: Binary + BLS format: ranges across all workloads and network sizes

5 Bundler Economics

The bundling fee $f_{\text{bun}}(t)$ compensates bundlers for Bitcoin block space costs. Bundlers accept transactions whose fees exceed their marginal cost (share of BTC block space fee + operational overhead). Users benefit when the bundling fee is less than the BTC fee they would pay for individual publication.

Each bundle commits to a **publisher key** that identifies who receives the fees. Bundling fees are **inclusion-conditioned**: paid when the bundle confirms on Bitcoin. A fraction β_{fee} of each fee is burned; the remainder is paid to the bundle publisher.

Bundling is trustless: any party with access to signed operations can produce a valid aggregate. Bundlers compete on fee efficiency, latency, and reliability.

5.1 Market Structure

The bundling market is permissionless and competitive. Bundlers compete on price, latency, and reliability; users can broadcast to multiple bundlers and select the best offer. Monopolization is prevented by protocol design: users sign with their own keys (any bundler can include any signed operation), fees are transparent on-chain, entry barriers are low (no staking required), and users can self-bundle. These dynamics drive fees toward marginal cost.

5.2 Fee Structure

The bundler's revenue per operation is:

$$r_{\text{bundler}} = (1 - \beta_{\text{fee}}) \cdot f_{\text{bun}}(t) \tag{1}$$

where β_{fee} is the protocol burn rate. The bundler's cost per operation is:

$$c_{\text{bundler}} = \frac{f_{\text{BTC}}}{N} + c_{\text{ops}} \tag{2}$$

where f_{BTC} is the Bitcoin transaction fee for the bundle, N is the number of operations in the bundle, and c_{ops} is operational overhead (bandwidth, computation, etc.).

Bundlers are profitable when $r_{\text{bundler}} > c_{\text{bundler}}$, which simplifies to:

$$f_{\text{bun}}(t) > \frac{\frac{f_{\text{BTC}}}{N} + c_{\text{ops}}}{1 - \beta_{\text{fee}}} \tag{3}$$

At large batch sizes ($N \approx 1000$), the Bitcoin fee is amortized to near-zero per operation, and bundling fees approach pure operational costs. This creates economies of scale that favor larger bundlers, balanced by the competitive dynamics above.

6 Conclusion

The metaprotocol architecture's commitment to settling all transaction data on Bitcoin imposes a hard constraint: 4,000,000 weight units per block. Within this constraint, a naive one-transaction-per-operation approach yields comparable throughput to that of vanilla Bitcoin payments.

The optimization stack presented here—BLS signature aggregation, compact registry IDs, binary encoding, and Zstd compression—achieves 35–42× improvement at batch sizes of 100 operations, enabling 366–443 TPS. At batch sizes of 10,000, throughput reaches 771–1295 TPS (73–123×). This positions Kontor to support sophisticated DeFi applications—order books, AMMs, lending protocols—at throughput levels competitive with dedicated Layer-1 chains, while maintaining the security and finality guarantees of Bitcoin settlement.

7. Bibliography

[1] Adam Krellenstein, Wilfred Denton, and Ouziel Slama, "Kontor: A New Bitcoin Metaprotocol for Smart Contracts and File Persistence," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/whitepaper

[2] Ethereum Foundation, "Phase 0 – The Beacon Chain: BLS Signatures." [Online]. Available: https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#bls-signatures

[3] Offchain Labs, "Inside Arbitrum." [Online]. Available: https://docs.arbitrum.io/how-arbitrum-works/inside-arbitrum-nitro

[4] Vitalik Buterin, "An Incomplete Guide to Rollups." [Online]. Available: https://vitalik.eth.limo/general/2021/01/05/rollup.html