Horizontal Scaling for a Zero-Knowledge Perpetuals Exchange

Sharding the State Tree, and Proving It Back Together

Dustin Norwood

Lambdachain

dustin@lambdachain.xyz

September 1, 2026

Abstract

General state sharding has failed repeatedly for one structural reason: with arbitrary smart contracts anything can touch anything, so shard boundaries are imposed rather than discovered and the traffic across them is unbounded. An exchange does not have that problem. Its state is already partitioned along lines the workload respects — markets never read one another, and accounts partition by index — so the seams are found, not cut.

This paper describes how Lambdachain, a zero-knowledge perpetuals rollup settling on STRATO, exploits that structure to scale horizontally. Each subtree of the global state tree is owned by one sequencer that advances it alone; the only thing that ever crosses a boundary is value, carried by an asynchronous message the sending shard debits locally and the receiving shard credits on arrival. Because a perpetuals account's health depends on every market it trades, margin is made to shard *with* the market, so health, mark-to-market and liquidation become purely local computations. Correctness is then reconstructed by proof rather than assumed: shard-block circuits, a cyclic range recursion, and a cross-shard join that proves value was conserved over histories that never synchronise. That join states conservation as a balanced multiset identity over a settlement epoch and discharges it as a grand product at Fiat–Shamir challenges bound to the epoch's own committed ranges, in the lineage of the permutation and lookup arguments of PLONK [9, 10]. A single settlement proof welds the committed global state root to that conservation at the same epoch, and to the epoch before it, so the settlement contract verifies one proof and one root equality.

We report measurements from the implementation. Execution throughput on independent shard pairs is near-linear until the machine's cores saturate (93% of ideal at four pairs, 59% at eight on a ten-core machine running sixteen shard processes). A single shared accounts shard is a hard wall — eight markets buy only 1.77× the throughput of one — and sharding the accounts tree by index range cuts the same workload's wall-clock by 2.94×. A cross-shard escrow costs 81 μs against a local account write's 24 μs, of which only $\approx 9\,\mu$s is the messaging itself. On the proving side each aggregation node costs a constant 1.35 s regardless of tree width, so total work is linear in the channel count while the critical path is the tree depth — a property realised on a fleet rather than on one machine, which is why the design leans on permissionless proving.

Every measurement here was taken on one ten-core machine at a named commit; Section C reproduces each, and Section 6.1 marks those that predate the order book.

Contents

1 The ceiling, and why an exchange can go through it

Lambdachain is a perpetual-futures exchange that runs as a zero-knowledge rollup, in the lineage of validity-proven off-chain execution that Ethereum's scaling roadmap made the default [2, 3]. Orders are accepted, matched and risk-checked by a sequencer off-chain; every block of transactions is published, proven correct by an independent prover, and settled against a contract on STRATO that verifies the proof. Nothing about correctness is trusted: a fill that was not at best-of-book, a cancelled order that was nonetheless filled, a balance moved without authorisation — none of these produce a valid proof, so none of them settle.

That architecture has one sequencer. It advances one order book at a time and moves one global state root forward, block by block. Its throughput is whatever a single machine can do, and no amount of additional hardware changes that, because the bottleneck is not hardware but serialisation: the state root is a single value and only one party may move it.

This paper is about removing that ceiling. The claim it defends is narrow and concrete:

The thesis. The exchange's state can be partitioned into subtrees that advance *independently*, on separate machines, with no synchronisation between them, and the result can be proven to be exactly the state a single machine would have produced — including that no value was created or destroyed at any boundary — at a proving cost that grows linearly in the number of boundaries and a proving *latency* that grows logarithmically.

1.1 Why general state sharding fails, and why this is not that

Sharding a blockchain's state has been attempted often and has mostly not worked; Ethereum's own roadmap eventually retreated from execution sharding to data sharding [22]. The reason is structural rather than a matter of engineering effort. In a world of arbitrary smart contracts, any contract may read or write any other contract's state, so a partition of the state is something imposed on the workload from outside. The boundaries fall in arbitrary places, the traffic across them is unbounded, and the messaging overhead consumes the parallelism that motivated the cut. Halve the state and you spend the gain passing messages.

An exchange is not in that position, because its state is *already* partitioned along lines the workload respects:

- **Markets are mutually independent.** The BTC-PERP book and the ETH-PERP book never read or write one another's state. An order in one has no bearing on matching in the other. The cut between them is not something the design imposes — it is a property of the problem.
- **Accounts partition by index.** The accounts tree is a fixed-depth Merkle tree [5] keyed by account index. Splitting it into contiguous ranges is a clean boundary, and — as Section 5 shows — one that a Merkle commitment makes provably conservative, because the two halves of a balanced tree *are* its two child subtrees.

So the seams are found rather than cut, and the traffic across them is not arbitrary contract calls but a single thing: value moving between an account and a market. That is what makes the problem tractable, and it is the reason this paper can make a claim about a sharded exchange that nobody has been able to make about a sharded general-purpose chain.

1.2 Where this sits

It is worth naming the design space before entering it, because the choice this paper makes is a point on a spectrum rather than an isolated idea.

Centralized exchanges run one in-memory matching engine per market and are, by a wide margin, the fastest systems in this comparison. They are fast for a structural reason that is not available to anything on-chain: they prove nothing and custody everything. Their throughput ceiling is a single

machine's, too — but they scale breadth-wise by simply running more machines, because no global state commitment ties the markets together. That is precisely the property this paper is trying to recover without giving up custody or proof.

On-chain order books put matching into consensus. Hyperliquid [24] runs a central limit order book on its own L1, so every order is a consensus transaction and correctness comes from replication — every validator re-executes. dYdX [26] moved in the same direction, to an application-specific chain with an off-chain book replicated across validators and on-chain settlement. The ceiling here is consensus throughput, and the cost of correctness is paid by every validator repeating the work.

Validity rollups for exchanges pay for correctness once, in a proof. StarkEx [23] pioneered this for trading systems and ran dYdX's earlier versions; Lighter [25] applies the same idea to a perpetuals venue, proving that matching followed the rules rather than asking anyone to re-execute it. Lambdachain is in this family, and everything in Sections 2 and 4 takes that starting point for granted.

What separates the work described here from all three is the axis of growth. A validity rollup already decouples correctness from re-execution; what it does not do by itself is decouple *throughput* from a single sequencer. Batches get larger, proofs get cheaper per transaction, and the state root still advances one machine at a time. This paper is about the remaining step: making the state root itself a composition of independently advancing subtrees, so that adding hardware adds throughput — the property a centralized exchange has for free and that proving normally takes away.

1.3 Two couplings that have to be severed first

Finding the seams is not the same as being able to cut along them. The exchange as it runs today has two couplings that cross the boundaries, and both must go before any subtree can advance alone.

The first is transactional. A single transaction can write an account leaf and a market leaf at once — an order debits collateral and touches the book in the same atomic step. Under sharding that becomes a local half plus an asynchronous message: the sending shard deducts and emits, the receiving shard credits on arrival (Section 2.2).

The second is harder, and it is where a perpetuals exchange refuses to behave like a spot exchange. Today an account holds *one* collateral pool backing every position across every market — cross-margin. An account's health is therefore a function of that pool together with the mark price and funding of every market it trades. Put collateral on an accounts shard and positions on market shards and the liquidation check becomes a synchronous read across every shard the account touches: precisely the fan-in the design exists to avoid. The resolution is to make margin shard *with* the market (Section 2.3), which changes the risk model and is the single most consequential design decision in this paper.

1.4 Validity is free; only liveness needs a stake

The lever that makes all of this work is the rollup's own position. Because every transition is proven, a shard *cannot* forge state. There is no honest-majority assumption to distribute, no fraud window to keep open, no committee to sample. Validity is discharged by the proof, and it costs the design nothing to spread work across machines that do not trust each other.

This is the same trade every validity rollup makes, and the reason exchange- specific rollups such as StarkEx [23] can commit to correctness without a challenge period; what is new here is spending that budget on *breadth* rather than on batch size. What a shard can still do is stall — refuse to process an inbound credit and strand a user's funds. That is a liveness failure, and liveness is what stake and

the settlement chain's forced-inclusion queue are for. The distinction matters architecturally: it is why the expensive, deferred, recursive aggregation described in Section 4 can lag arbitrarily far behind live block production without being unsafe, and it is why the cluster can eventually be permissionless rather than a closed set of trusted machines.

1.5 What is built, and what this paper reports

Everything described here exists as running code in the Lambdachain rollup repository. Concretely:

- A Go package (`scaling/`) that is the executable specification: two shard types with their own state trees, channels, blocks, and durable journals, advancing independently over an asynchronous transport.
- A shard runtime (`scaling/node`, `cmd/shardnode`) that runs one shard as a process with a mempool, a block loop, an HTTP outbox peers pull from, and journal-based recovery.
- A commitment daemon (`cmd/aggd`) and a prover daemon (`cmd/scaleproverd`) that reconstruct proving jobs from the journals.
- A Rust circuit crate (`zk/circuits`, plonky2 over Goldilocks) with the shard-block, range, product, epoch, join, conservation-tree, state-root and settlement circuits, plus the BN254 wrapping tail.
- A settlement contract (`contracts/SettlementAnchor.sol`) deployed on a STRATO testnet, with three epochs chained on chain and the third anchored end-to-end without human intervention.

Section 2 gives the architecture. Section 3 gives every data model, field by field, in the exact encoding the circuits verify — this is the longest section and it is deliberately exhaustive, because the soundness argument later rests on those encodings being injective and position-distinct. Section 4 gives the proof architecture and the conservation argument. Section 5 gives the elastic split. Section 6 reports measurements. Section 7 states, without softening, what is not built and where the argument does not yet reach.

On the honesty of the measurements. An early version of the throughput harness measured its own single-threaded workload submitter rather than the shards, and reported the system as scaling 24% of ideal at eight shards when it in fact scales 59% — which is core saturation on a ten-core machine running sixteen shard processes, not a design wall. The bug and its fix are described in Section 6.1. It is recorded here rather than quietly corrected because the failure mode — a benchmark that serialises the thing it is measuring — is one every distributed-systems claim should be checked against, including the ones in this paper.

---

2 Architecture

2.1 One sequencer per subtree

Each sequencer owns a subtree of the global state and runs only that slice of the exchange. A **market shard** runs one order book and its risk engine. An **accounts shard** — called an FCS, a *free-collateral shard*, in the implementation — maintains balances for one contiguous range of account indices. Each produces its own blocks, publishes its own data, and hands off to its own prover: exactly what

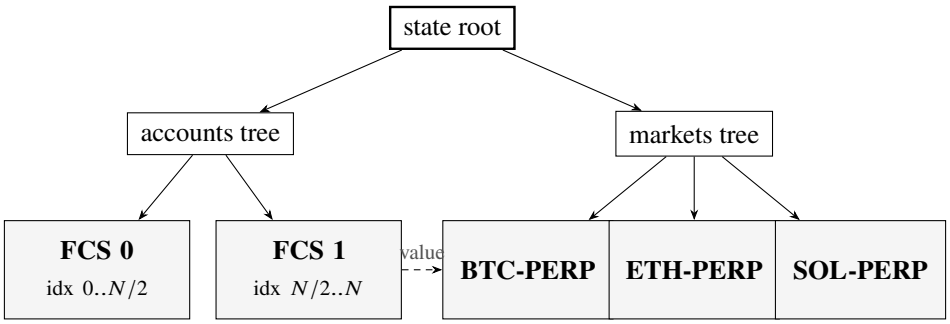


Figure 1. The state tree is already a map of independent subtrees. Sharding hands each subtree to its own sequencer and prover. Solid edges are tree commitments; the dashed edge is the only thing that ever crosses a boundary — an asynchronous message carrying value. Sibling markets never message each other, which is why conservation has to be enforced only where the accounts subtree and the markets subtree meet.

the single sequencer does today, but for its slice alone (Figure 1).

Two shards therefore never share a lock, a clock, or a block cadence. They do not agree on an ordering of events, and nothing in the design ever asks them to.

2.2 Cross-shard messages: deduct here, credit there

The only thing that ever crosses a boundary is value — USDλ moving between an account and a market. This is the receipt pattern asynchronous distributed systems converge on; what makes it available here at all is that the message set is a single type rather than arbitrary contract calls. There is no global lock and no distributed transaction. A shard that sends value **deducts locally and emits a message**; the receiving shard **credits on arrival** and records it in its own ledger. If the receiving shard declines the operation, it sends the value straight back as an ordinary reciprocal transfer. There is no reclaim path and nothing to unwind: a rejected transfer is simply returned the way it arrived (Figure 2).

That the return leg is a *transfer* and not an *unwind* carries an assumption worth stating outright, because it is load-bearing. The returned value is credited to whichever account the message names, and nothing in this design ever removes an account, so the return is **unconditional by construction**: there is always an account to credit, and a rejection can never fail for want of a home. That rests entirely on account permanence. Were accounts ever evicted to reclaim state — the pressure that eventually reaches every rollup — a transfer in flight to a since-removed account would strand, and shards would have to track in-flight value per account rather than per channel. Naming the dependency now keeps a future eviction from quietly reopening it.

Delivery is in-order; exactly-once is the consumer's job. The transport between shards promises only that a channel's messages arrive *in order* and *eventually*. It does not promise that each arrives exactly once. This is deliberately the weaker guarantee, because it is the one a permissionless network can actually keep: a message can always be re-submitted, so a duplicate is normal rather than a fault.

Exactly-once is recovered where it belongs — at the consumer. A shard consumes strictly at its

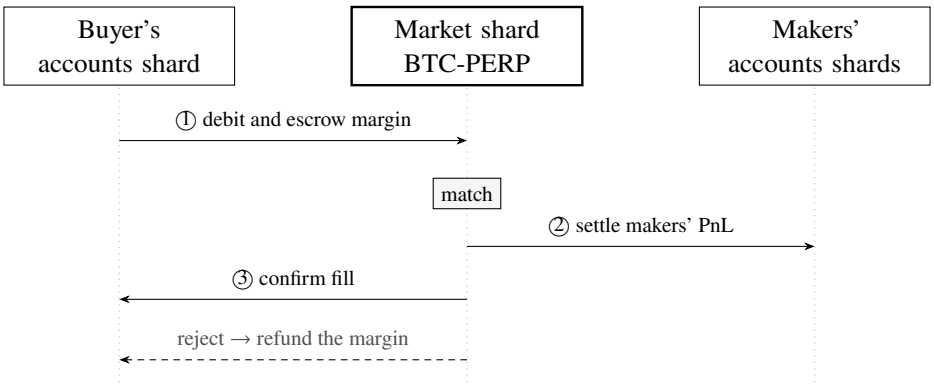


Figure 2. A cross-shard buy. No shard ever waits on another mid-transaction: each updates its own state and links the change to the next with a message. The dashed leg is the alternative outcome. A rejected escrow is returned as an ordinary *reciprocal transfer*, not an unwind — the value leaves and comes back as two independent transfers, so there is no reclaim path, nothing to roll back, and no distributed transaction anywhere in the design.

committed `wantSeq`, so a duplicate falls behind the counter and is a no-op, and a gap waits rather than skipping ahead. The alternative, a nullifier set accepting messages in any order, was rejected: it trades a bounded, checkpointable counter for unbounded per-channel state and a murkier notion of what a channel's committed position even means.

The cost of the in-order choice is head-of-line blocking — a missing message stalls its channel until it lands — and that is tolerable for exactly one reason: the settlement chain's forced-inclusion queue turns "eventually" into a guarantee, so a stalled channel always has a way to make progress that does not require trusting a sequencer.

Admission is a proof, not a choice. When an escrow arrives, the market shard either opens the position or sends the value back, and that decision is not discretionary. Opening and rejecting assert *exact complements* of one admissibility predicate over (committed state × message), so for any escrow exactly one of them is provable. Two failures are ruled out at once. A well-margined order cannot be selectively rejected — that would be censorship no stake penalty could distinguish from a genuine reject. And, more subtly, no message can fall into a gap that is *neither* acceptable nor rejectable, which would leave it provable by no block at all and, because a channel is consumed strictly in order, wedge that channel permanently.

This buys a constraint that must be honoured going forward: the admissibility predicate has to stay expressible in a circuit and *total* over the state it is admitted against. A risk control that cannot be written that way would either reopen discretion or strand messages — a real limit on the risk model, and one better known now than discovered later.

2.3 Margin has to shard with the market

Here is where a perpetuals exchange refuses to behave like the spot example above, and where the design earns its keep.

Today the account leaf holds one collateral pool backing every position across every market. An account's health is a function of that pool *and* the mark price and funding of each market it trades.

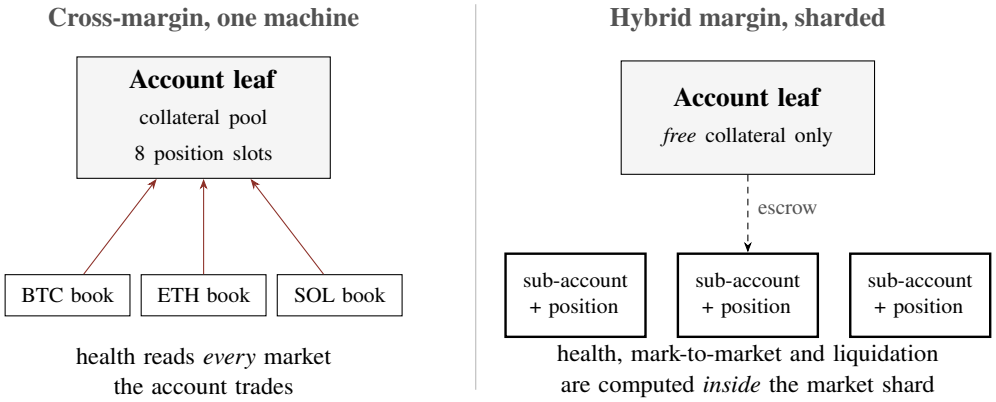


Figure 3. The decision the design turns on. Left: one collateral pool backs every position, so valuing an account requires reading every market it trades — the synchronous fan-in sharding must avoid. Right: when a position opens, its margin is escrowed out of the free pool into a sub-account that lives *in the market shard*, beside the position it backs. Health, mark-to-market and liquidation become purely local. Cross-margin does not disappear; it becomes an explicit rebalance — a slow cross-shard round trip, which is acceptable precisely because it is rare.

Put collateral on an accounts shard and positions on market shards, and deciding whether an account can be liquidated becomes a read across every shard it touches.

The resolution is to make margin shard with the market. When a position opens, its margin is escrowed out of the account's free pool and into a sub-account that lives *inside the market shard*, alongside the position it backs (Figure 3). Health, mark-to-market and liquidation for that position are then computed entirely within the market shard, with no cross-shard read on the hot path, ever. Cross-margin does not disappear — it becomes an **explicit rebalance**, moving free collateral between shards, which is allowed to be a slow cross-shard round trip because it is rare.

This is a revision of the exchange's published state model, not an extension of it, and it is the change with the largest product consequence in this paper. Its payoff is measured in Section 6: a liquidation costs a market shard nothing it would not already pay, because every input it needs is in the leaf it is already reading.

2.4 Committing is not proving

Two roles are deliberately kept apart, and conflating them is the most common way to misread this architecture.

A single **root aggregator** takes every sequencer's block header and moves one leaf of the global tree — that shard's subtree root. It maintains the whole tree from single-leaf updates and holds a current global root at all times: the **pending root** the speculative account view samples, posted to the settlement chain at fixed intervals. It never needs more than the latest root per shard, because a block header already commits its predecessor's root and therefore that shard's whole history transitively.

Nothing about the aggregator is trusted. A wrong root earns no valid proof, so it never settles and nothing gates on it. Committing is a liveness role, exactly as open to being made permissionless as proving. And because one machine holds the whole tree, it can reject a block whose parent is not that shard's committed root the instant it arrives — so the pending view never runs on an obviously

broken chain even though what lands on chain is the aggregate.

Provers begin where the aggregator ends. Once a root is committed, the tree beneath it is fixed, and the recursion of Section 4 folds it into the **settled root** that a withdrawal waits on.

Withdrawals gate on a reconciled root. Shards advance their own state freely and cheaply. Value leaves the exchange only against a root the cross-shard join has proven *conserved*. That single rule is what lets reconciliation lag arbitrarily far behind live activity without being unsafe — and it is why the expensive recursive aggregation is a throughput-and-continuity layer rather than a safety prerequisite.

2.5 Exactly-once, and where safety actually lives

Proving conservation for one batch is not enough. Value is safe across time only if a transfer cannot be **replayed**, **fabricated**, or **rewound**. Each is defended, and — usefully — each is defended at a *different* layer (Table 1), which keeps the design honest about where its safety comes from.

Attack	Caught by	Layer
A shard consumes a message at the wrong position	the message's sequence number must equal the channel's committed `wantSeq`	inside the shard-block proof
A shard fabricates a message no one emitted	the consumer's inbound chain must equal the emitter's outbound chain	at reconciliation — the join
A shard rewinds its counter to re-consume	each block must open its predecessor's committed root; a rewind is a fork	per-shard chain continuity
The composed global state forks	the composed root chains block to block	global continuity — the aggregate

Table 1. Only *one* of the four defences — fabrication — is discharged by the cross-shard join. The rest live where a rollup already defends its state.

The third row of Table 1 deserves emphasis, because it is *not* new machinery. Rewinding a counter means offering an earlier root as a block's parent, and refusing that is precisely what a rollup's parent-root check already does — the same mechanism that stops any chain rewinding its own state. So exactly-once does not depend on the cross-shard aggregation for its safety. The aggregation gives *global continuity*: that the composed root is itself a chain. Real and required, but a continuity property, not the thing that stops a double-credit.

2.6 And why transfers need no signatures — yet

A receiving shard cannot credit itself value nobody sent, because the aggregate would fail to reconcile. The *proof*, not a signature, is what stops fabrication. Signatures on the handshake between two shards would prove they agreed to a transfer; they would not prove value was conserved.

Signatures return only in a future where a shard must accept a message from an *untrusted* sender before that sender's proof exists — and then the cheap form is a signature on the sender's committed chain tip, checked off-circuit, so an observer can attribute a fault without re-running a proof. While

the conservation proof is the enforcement, they are weight the design does not need to carry.

3 Data models

This section is exhaustive on purpose. Every commitment in the system is a Poseidon2 hash over a fixed vector of field elements, and the soundness argument in Section 4 depends on those vectors being injective and position-distinct. A reader who wants to check that argument needs the exact element lists, so they are given here in full, each with the file that defines it.

3.1 The field, the hash, and the encodings underneath everything

Field. All arithmetic is over the Goldilocks prime

$$p \;=\; 2^{64} - 2^{32} + 1 \;=\; \texttt{0xFFFFFFFF00000001},$$

chosen because it is the native field of the plonky2 prover [16] (`gl/gl.go:34`). A **digest** is four canonical field elements, $\mathsf{Digest} = \mathbb{F}_p^4$ (`gl/gl.go:37`).

Hash. $\mathsf{H}(x_1, \ldots, x_n)$ is plonky2's `hash_n_to_hash_no_pad` [16] instantiated with **Poseidon2** [19] over Goldilocks, rate 8, no padding (`gl/gl.go:124`). Poseidon2 is the arithmetisation-friendly successor to Poseidon [18], chosen because the dominant cost in both the execution layer and the circuits is hashing state-tree paths, and an algebraic hash is far cheaper to prove than a bitwise one. It is the same permutation the circuits use, from the same library, so a Go-computed commitment and its in-circuit recomputation are equal by construction rather than by agreement. The Merkle node function is

$$\mathsf{T}(l, r) \;=\; \mathsf{H}(l_0, l_1, l_2, l_3,\; r_0, r_1, r_2, r_3).$$

The empty leaf is the literal zero digest $\mathbf{0}$, not the hash of zeros (`gl/gl.go:41`) — a plonky2 convention that makes an empty sparse subtree cheap to recognise.

Integer encodings, and why they differ. Two encodings recur in every leaf, and their asymmetry is not arbitrary.

A USDλ amount ($< 2^{128}$) is *four 32-bit* limbs. `U128Limbs` (`gl/gl.go:264`) splits an amount into four little-endian 32-bit pieces. The width is chosen so that the *product* of two limbs stays below p: an in-circuit multiplication of two 32-bit values cannot wrap the field, so amount arithmetic needs no reduction gadget. Every `Amount`, `Margin`, `Free` and `pool` contributes four elements.

An owner address ($< 2^{160}$) is *limbs that are canonical by construction* — but not always the same ones. An owner is never multiplied, so the wrap argument above does not apply, and the obvious three 64-bit limbs look cheapest. They are unsafe. A 64-bit limb can exceed p, and H rejects a non-canonical element rather than reducing it. An address is not ours to choose — a withdrawal destination is named by the user — so a destination whose low 64 bits are $\geq p$ would abort the sequencer at apply time. That makes a crafted address a denial of service rather than a bad withdrawal. Any limb narrower than the field's 63.99 bits is canonical by construction and the check cannot fail.

Which narrower width, though, turns out to matter more than it looks, and the system now uses two.

Three 56-bit limbs (`U160Packed`, `gl/gl.go:438`), the seven-byte packing, wherever a commitment sits near a sponge-rate boundary and nothing outside the protocol reads it. The account leaf is the case that matters: at three limbs it is an eight-element hash, and eight is the last arity that fits one Poseidon2 permutation (Section 3.11).

Five 32-bit limbs (`U160Field`, `gl/gl.go:419`), wherever a contract reads the same commitment. The exit leaf's destination is decoded on chain by the withdrawal vault, which shifts it by $0, 32, \ldots, 128$ and feeds eleven inputs to its Poseidon2 — so that encoding is an interface, not a choice, and repacking it would gain nothing anyway: at nine elements the exit leaf still spans two permutations.

Two encodings for one type is a footgun, and it is worth being plain about why it is worth carrying. The rule is mechanical — packed where the arity is load-bearing and the commitment is internal, field where something off chain must reproduce it — and the circuit does not witness both forms independently, which would let them disagree. It derives the field form from the packed one by splitting each 56-bit limb into bits it must range-check anyway, so the two cannot diverge without the range check failing first.

The cost of getting this wrong is not hypothetical. An intermediate version used five limbs for the account leaf as well, which took it from eight elements to ten: across the rate, one permutation to two, $\approx 2\,\%$ of execution throughput measured end to end (Section 6.5). The security argument was right and the encoding was still the expensive one.

An amount, then, is four 32-bit limbs because the *product* of two limbs must not wrap the field; an address is limbs of some width < 64 bits because each limb must be a legal field element in the first place. The second constraint is the stronger one — it binds even where no arithmetic happens at all — and it is easy to miss, because a hash of a wider limb is correct for every value anyone would test with and wrong only for values an adversary supplies.[0] Mixing them up is the single easiest way to write a circuit that disagrees with the specification, which is why sixteen conformance vectors — a block, a consume, a multi-peer emit, a withdrawal block, the product, the aggregator root, the challenge derivation, the split and merge roots, and the rest — are emitted by the Go tests and pinned into the Rust ones, locking both sides to the same values (Section C). The mechanism earned itself when the owner encoding changed: rewidening an owner moves every root that commits one, and the vectors are what establish that the Go model and the circuits moved together rather than separately.

It also showed its limit. One pinned vector — a multi-peer emit's data-availability hash — had been wrong for three months without anyone noticing. The earlier widening had grown the record encoding's owner lane, which moves the DA hash of *every* record, including record types that carry no owner at all and are merely zero-filled in that lane. The constant was never re-derived. It stayed invisible because an earlier assertion in the same test was also stale and failed first, so execution never reached the line that would have caught it. A conformance vector only conforms if the test gets to it, and a stale assertion upstream is an effective mute on everything below it.

The sparse tree. A shard's state lives in a fixed-depth sparse Merkle tree (`gl.SparseTree`). Depth is 24 in the running configuration — 2^{24} leaves — and every leaf sits at the same level, which matters for Section 3.11. Writing a leaf costs one hash per level. Measured, that is $20.99\,\mu$s on the reference machine (Section 6.2), and it is the dominant cost in the entire execution path.

[0] The legacy three-limb 64-bit `U160Limbs` remains in the frozen v3 engine, which cannot change its commitments; it never meets the scaling model, which is reached by regenesis rather than by migration.

3.2 The accounts shard

The account leaf. Under hybrid margin the account leaf carries only *free* collateral; positions and the margin backing them have moved into the market shard (`scaling/accounts.go:14`).

```
type acctLeaf struct {
    Owner *big.Int // u160 STRATO address
    Free  *big.Int // u128 free collateral
    Nonce uint64
}
```

Its commitment is an eight-element hash (`accounts.go:21`) — one permutation, deliberately (Section 3.11):

$$\mathsf{H}\big(\underbrace{o_0, o_1, o_2}_{\text{owner}},\ \underbrace{f_0, f_1, f_2, f_3}_{\text{free}},\ nonce\big), \qquad \text{empty slot} \mapsto \mathbf{0}.$$

Contrast the exchange's current (unsharded) account leaf, which also holds the collateral pool and eight position slots. The sharded leaf is deliberately a *new* shape rather than a mutation of the pinned one: changing the deployed leaf would be a regenesis-class event and would invalidate every existing fixture, so the scaling package commits its own leaves and reuses only the primitives.

Ownership, and the base offset. An accounts shard owns a contiguous range $[base,\ base + 2^{depth})$ of *global* account identifiers. An account keeps its global id, but sits at local position $idx - base$ in the shard's own (possibly shallower) tree (`accounts.go:104`). Only an elastic split ever moves `base` off zero (Section 5).

A requirement the benchmark forced into the open. Account identifiers must be globally unique across accounts shards, not per-shard indices. Two shards' "account 5" must not collide as the same key in a market's sub-account tree, or one silently displaces the other and value strands. In practice that means addresses derived by a hash across the whole space, with each shard owning a contiguous range. Note what this does and does not buy: uniform addresses balance *storage* across shards; they do not balance *load*, because one hyperactive account still lands wholly on one shard.

The shard root. An accounts shard's committed root folds its state tree together with its channel root and its exit tree (`accounts.go:145`):

$$R_{\text{acct}} \;=\; \mathsf{H}\big(t_0, t_1, t_2, t_3,\ c_0, c_1, c_2, c_3,\ e_0, e_1, e_2, e_3\big),$$

where t is the account-tree root, c the channel root of Section 3.5, and e the root of the **exit tree** — an append-only tree of $\mathsf{H}(account, amount[0..3], dest[0..4], seq)$ leaves, one per withdrawal leaving for L1, indexed by the shard's withdrawal sequence. Folding it here is what carries a withdrawal into the settled global root: the release contract proves a payout by opening its leaf against an exit root that a settlement has anchored, recomposing R_{acct} from the accounts and channel roots as opaque siblings. It is the same move as proving a receipt against an anchored receipts root, and it is the reason a withdrawal needs no trusted relayer — but the withdrawal path proper is outside this paper's scope; what matters here is that it is a third sibling under the same root, advanced by the same block

proofs. This is the structural decision that moves exactly-once out of unproven memory: because the channel counters and chains live *inside* the root, every shard-block proof attests to them, and a shard cannot advance its sequence cursors without that advance appearing in the root its next block opens.

3.3 The market shard

Position and sub-account. A position (`market.go:16`) is a side, a size in steps, and the entry mark: $\mathsf{H}(side, size, entry)$, three elements, **0** when absent. The sub-account is the storage the unsharded market leaf does not have, and growing it is exactly what lets health be computed locally (`market.go:34`):

```
type subAccount struct {
    Margin  *big.Int // u128, escrowed out of the account's free pool
    Pos     *pos     // the position it backs
    openSeq uint64   // the Seq of the EscrowOpen that opened it
    owner   ShardID  // which accounts shard to settle back to
    account uint32
}
```

Its commitment is eight elements (`market.go:42`):

$$\mathsf{H}\big(m_0, m_1, m_2, m_3,\ \pi_0, \pi_1, \pi_2, \pi_3\big), \qquad \pi = \mathsf{H}(side, size, entry).$$

What is not committed, and what that leaves unproven. `openSeq`, `owner` and `account` are *not* hashed. They are routing metadata: they record which escrow opened the sub-account and which accounts shard a settlement must be addressed back to. Leaving them out keeps every sub-account leaf and every circuit that opens one smaller, and the values are reconstructible by replaying the shard's published data, since the `EscrowOpen` that created the sub-account is itself a record carrying all three.

That is a real trade rather than a free one, and the consequence is worth separating carefully into two claims, because only one of them is discharged.

Value is conserved. A `SettleClose` is emitted once and consumed once, so it appears exactly once on each side of the balanced identity of Section 4.6. No value is created or destroyed by a settlement, whoever receives it.

The destination is not enforced. Conservation balances regardless of *which* accounts shard receives, and `owner` is uncommitted, and — decisively — there is no settle-side circuit anywhere in the scaling stack: the inventory of Section 4.4 proves the *open* side, and nothing binds a `SettleClose`'s destination to the `EscrowOpen` that caused it. So a market shard that addressed a settlement to the wrong accounts shard would produce a perfectly conserving epoch. The correct destination is *recoverable* from published data — the causing escrow is there, naming its source — but recoverable is not enforced.

The distinction is between value being *created* and value being *misdelivered*. The first is proven impossible; the second is not yet proven anything. Binding a settlement's destination to its originating escrow belongs with the settle-and-liquidation proving work Section 7 sets out, and it is listed there rather than left implicit here.

Risk parameters and the pool. `MarketParams` (`market.go:52`) carries the market-local risk config: `QuoteMultiplier`, `ImrBps` (initial margin in basis points of notional), `MmrBps` (maintenance margin), and `MaxSizeSteps`. The **pool** is all collateral the shard custodies — every open position's escrowed

margin plus any seeded insurance. Escrow-in adds to it; settle-out subtracts. That is what makes

$$\sum_{\text{accounts}} free \;+\; pool \;+\; inFlight \;=\; \text{constant}$$

an invariant *by construction* at every instant, including between a debit and the counterpart credit — not merely at a drain point.

The market root. Thirteen elements (`market.go:219`):

$$R_{\text{mkt}} \;=\; \mathsf{H}\big(\underbrace{s_0..s_3}_{\text{sub-accounts}},\; \underbrace{mark}_{\text{oracle}},\; \underbrace{p_0..p_3}_{\text{pool}},\; \underbrace{c_0..c_3}_{\text{channels}}\big).$$

An oracle update, a sub-account write and a settlement all advance the same root, so none of them can happen off the record.

A shard that runs an order book folds its book root as a fifth sibling, making seventeen; a bookless shard folds nothing extra and its root is byte-identical to what it was before books existed. Resting orders are therefore inside the root every block proof attests to, which is what stops an order appearing, moving or vanishing between blocks without a proof saying so. The book's own structure and its matching rules are outside this paper's scope.

Health, locally. With margin resident, the three risk quantities are functions of one leaf and one scalar:

$$\begin{aligned} \text{pnl}(\pi) &= (mark - entry) \cdot size \cdot qm \cdot (side = \text{long}? + 1 : -1), \\ \text{equity}(a) &= margin_a + \text{pnl}(\pi_a), \\ \text{maint}(a) &= size_a \cdot mark \cdot qm \cdot mmrBps/10^4, \end{aligned}$$

and a is liquidatable exactly when $\text{equity}(a) < \text{maint}(a)$ (`market.go:219`). No cross-shard read appears anywhere in that computation. That is the entire payoff of Section 2.3, and it is why a liquidation is not a distributed operation in this design.

A close — voluntary or forced — pays $\max(0, margin + \text{pnl})$ out of the pool and emits a `SettleClose` back to the owning accounts shard. A bankrupt position returns zero and its shortfall is absorbed by the pool; a payout that would overdraw the pool is refused outright rather than silently netted, because that condition is an insurance shortfall requiring auto-deleveraging, which is flagged and not implemented (Section 7).

3.4 Cross-shard messages

The message. Value never moves except as a `Message` (`messages.go:56`), and every message is folded into *both* shards' commitments — the emitter's and the consumer's — which is what turns conservation into a comparison of hashes.

```
type Message struct {
    Src, Dst ShardID  // uint32
    Seq      uint64   // per-channel monotone emitter nonce
    Kind     MsgKind  // EscrowOpen=1, SettleClose=2, Refund=3
    Account  uint32
```

```
    Market   uint32
    Amount   *big.Int // u128, the quote asset
    CauseRef uint64   // Seq of the EscrowOpen this settles/refunds; 0 for an open
    Side     uint8    // EscrowOpen only: 1 = long, 0 = short
    Size     uint64   // EscrowOpen only: size in steps
    Price    uint64   // EscrowOpen only: limit price; 0 = open at mark
    Expiry   uint64   // EscrowOpen limit only: 0 = good-till-cancel
}
```

`Seq` is the emitter-assigned per-channel nonce: simultaneously the ordering key and the idempotency key. `CauseRef` ties a settlement or refund back to the specific escrow it resolves, so a credit is against a named escrow and not a bare pool.

The message field vector. Thirteen field elements, in this exact order (`multiset.go:64`):

$$\vec{v}(m) = \big(Src,\ Dst,\ Seq,\ Kind,\ Account,\ Market,\ a_0, a_1, a_2, a_3,\ CauseRef,\ Side,\ Size\big).$$

This one vector serves two mechanisms that must agree, and the fact that they share it is load-bearing:

- the **chain fold** (`messages.go:84`), $\mathrm{fold}(h, m) = \mathsf{H}(h_0, h_1, h_2, h_3 \,\|\, \vec{v}(m))$, a 17-element hash that advances a channel's running commitment; and
- the **fingerprint** (`multiset.go:102`), $\varphi_\beta(m) = \sum_j v_j(m)\,\beta^j$, the single field element that stands for m in the grand product.

Because the chain and the product are taken over the *same* vector, a proof that the product was computed over some multiset and a proof that the chain landed on a committed value together force the product to be over exactly the messages the chain carried. That is the first of the three bindings in Section 4.7.

The two encodings are no longer *identical*, however, and the place they diverge is worth stating here rather than burying it in the proof chapter. A limit escrow carries a resting price and an expiry, and the chain fold appends them — as a pair, so their order cannot shift, and only when the escrow is priced, so an at-mark escrow folds byte-identically to what the system committed before limit orders existed. The fingerprint appends nothing: $\vec{v}(m)$ is thirteen elements for every message. Two escrows differing only in resting price therefore have the *same* fingerprint and *different* chains. Section 4.8 shows why that is the right split rather than an oversight, and what has to hold for it to be safe.

3.5 Channels

A **channel** is a directed pair of shards. A shard's `channelSet` (`messages.go:117`) holds, per counterparty: the running fold of what it has emitted and of what it has consumed, the outbound and expected-inbound cursors, and the cumulative value in each direction.

```
type channelSet struct {
    self  ShardID
    peers []ShardID                    // fixed, sorted counterparty set
    emitted, consumed map[ShardID]gl.Digest
    nextSeq, wantSeq  map[ShardID]uint64
    emittedValue, consumedValue map[ShardID]*big.Int
}
```

The peer set is *fixed at genesis* and kept sorted, so the channel root always commits the same leaves in the same order — a fixed structure the shard-block circuit can prove without a variable-length peer list. (The one exception is `addPeer`, which grows the set across an elastic split; a fresh peer starts at zero counters, so its leaf is the empty leaf and the root simply folds one more.)

The channel leaf. Eighteen elements (`messages.go:171`):

$$L_p \;=\; \mathsf{H}\big(\mathit{nextSeq}_p,\ \mathit{wantSeq}_p,\ \mathrm{out}_p[0..3],\ \mathrm{in}_p[0..3],\ \mathrm{ev}_p[0..3],\ \mathrm{cv}_p[0..3]\big).$$

An idle channel's leaf is H of eighteen zeros — a specific, computable value the split circuit recomputes rather than accepts as a witness (Section 5).

The channel root. The leaves fold in ascending peer order from $\mathbf{0}$ (`messages.go:187`):

$$c \;\leftarrow\; \mathsf{H}\big(c_0, c_1, c_2, c_3,\ p,\ L_{p,0}, L_{p,1}, L_{p,2}, L_{p,3}\big) \quad \text{for each } p \in \text{peers in order.}$$

Two consequences matter. First, an *idle* channel's in-flight position is pinned exactly like an active one, because its counters are committed whether or not this block touched it. Second, a shard talking on many channels is still one proof, read per channel: a block advances the peers it touched and carries the rest unchanged.

Emit and accept. `emit` (`messages.go:198`) stamps $\mathit{Src} \leftarrow \mathit{self}$ and $\mathit{Seq} \leftarrow \mathit{nextSeq}[\mathit{dst}]{+}{+}$, folds the message into $\mathrm{out}[\mathit{dst}]$, and adds its amount to $\mathrm{ev}[\mathit{dst}]$. `accept` (`messages.go:210`) requires $\mathit{Dst} = \mathit{self}$ and $\mathit{Seq} = \mathit{wantSeq}[\mathit{src}]$, then folds into $\mathrm{in}[\mathit{src}]$ and advances the cursor. Contiguity from zero is exactly-once and in-order; a duplicate falls behind the cursor and is a no-op, a gap waits.

The two value counters are what make in-flight a first-class quantity rather than an error state: for a channel $a \to b$, the value in flight is $\mathrm{ev}_a[b] - \mathrm{cv}_b[a] \geq 0$, and it is non-negative by the delivery discipline alone.

3.6 Blocks and data availability

Records. A block is a batch of transitions in a form a replayer can apply deterministically. Seven record kinds (`block.go:21`): `RecCredit`, `RecOpen`, `RecConsume`, `RecSetMark`, `RecFundInsurance`, `RecClose`, `RecLiquidate`.

An *inbound* message is embedded in the consuming shard's record; an *outbound* message is not stored at all — it falls out of replaying the operation that emits it. That asymmetry is what keeps the two shards' proofs independent: neither has to carry the other's data to be replayable.

`Record.encode` (`block.go:51`) produces a vector that is injective across kinds because every field is present, zero where inapplicable: a sixteen-element prefix

$$\big(\mathit{Kind},\ \mathit{Account},\ \mathit{Side},\ \mathit{Size},\ \mathit{Price},\ o_0..o_2,\ a_0..a_3,\ g_0..g_3\big)$$

(owner, amount, margin), then a presence flag: 1 followed by the thirteen message elements of Section 3.4 when a message is embedded, or a bare 0 when not.

The data-availability hash. Records fold in order (`block.go:77`), from $\mathbf{0}$:

$$h \;\leftarrow\; \mathsf{H}\big(h_0, h_1, h_2, h_3 \,\|\, \mathrm{encode}(r)\big).$$

The block header. The header is the shard-block circuit's public interface — what a proof exposes and what the layer above reads (`block.go:91`):

```
type BlockHeader struct {
    Shard        ShardID
    Number       uint64
    PrevRoot     gl.Digest   // the shard root this block opens
    NewRoot      gl.Digest   // the shard root it produces
    Timestamp    uint64
    OutboxCommit gl.Digest   // cumulative emitted, per channel
    InboxCommit  gl.Digest   // cumulative consumed, per channel
    DAHash       gl.Digest
}
```

`Header.Hash` is 23 elements in struct order (`block.go:103`). `OutboxCommit` and `InboxCommit` fold the per-channel chains keyed by counterparty (`block.go:155`), so the layer above can open a specific channel's position without opening the shard root.

3.7 Ranges, frontiers, and the epoch

Range summary. A **range** is a contiguous run of one shard's blocks reduced to a single summary (`range.go:23`):

```
type RangeSummary struct {
    Shard                   ShardID
    FirstNumber, LastNumber uint64
    RootBefore, RootAfter   gl.Digest
    DAChain                 gl.Digest  // fold over the range's per-block DA hashes
    Outbox, Inbox           gl.Digest  // cumulative commitments at RootAfter
}
```

Folding two adjacent ranges (`range.go:55`) checks the same shard, consecutive numbers, and $l.\mathit{RootAfter} = r.\mathit{RootBefore}$; the DA chain composes as $\mathsf{T}(l, r)$ and the endpoint commitments are the right range's, since they are cumulative. A single block is a range of length one whose DA chain is the block's own hash, unwrapped — which costs no extra hash and keeps a one-block range distinguishable from a folded one.

The frontier. At any point the proof has reached, some messages are sent but not yet received. That is not an error; it is the resting state of an asynchronous system, and the design carries it explicitly.

The load-bearing observation is that in-flight on a channel is exactly the emitter's messages with $\mathit{Seq} \in [\mathit{wantSeq}, \mathit{nextSeq})$ — a *contiguous suffix*. Both counters are already committed inside the shard roots, and the messages themselves are in the emitter's published data. So **the frontier needs no commitment of its own**: it is pinned by state the range proofs already prove, and a settlement recomputes it rather than trusting a carried blob (`frontier.go`).

The epoch. An **epoch** accumulates one settlement period's sends and receives against the frontier and, at close, yields the four multisets the conservation identity balances (`frontier.go:137`):

$$(\textbf{sent},\ \textbf{received},\ \textbf{inFlightIn},\ \textbf{inFlightOut}).$$

By construction **inFlightOut** = **inFlightIn** ⊎ **sent** \ **received**, so an honest epoch always balances; the identity is what a dishonest one cannot satisfy. And because the frontier persists across epochs, this epoch's **inFlightOut** *is* the next epoch's **inFlightIn**, with nothing to re-assert.

3.8 The epoch commitment and its challenges

The grand-product argument is sound only if its challenges are unpredictable when the epoch's messages are chosen. Otherwise a prover grinds two unequal multisets that collide at a known γ. So settlement is **two-phase**: first the epoch's range summaries fold into one commitment K, then the challenges are derived from it, and only then are the products computed (`scaling/challenge.go`).

$$K(\text{epoch}) = \text{fold of the per-shard range summaries, sorted by shard,}$$
$$(\beta, \gamma) = \mathrm{FS}\big(\text{priorSettledRoot},\ K\big).$$

Concretely, K seeds at $\mathsf{H}(\textsc{epochk})$ and folds, for each range in shard order,

$$\mathsf{H}\big(K_{0..3},\ \mathit{Shard},\ \mathit{First},\ \mathit{Last},\ \mathit{RootBefore}_{0..3},\ \mathit{RootAfter}_{0..3},\ \mathit{DAChain}_{0..3}\big),$$

and the challenges are read off the four limbs of one transcript hash $t = \mathsf{H}(\textsc{chalbg},\ \mathrm{prior}_{0..3},\ K_{0..3})$ as

$$\beta = (t_0, t_1), \qquad \gamma = (t_2, t_3),$$

each an element of the *quadratic extension* $\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2 - 7)$ (`challenge.go:59`). The non-residue matches plonky2's degree-two extension for Goldilocks, so the Go reference and the circuits agree by construction rather than by convention; Section 4.9 explains why the grand product runs in the extension rather than the base field. EPOCHK and CHALBG are the ASCII strings packed as field elements, `0x45504f43484b` and `0x4348414c4247`.

Two details are worth drawing out. The **prior settled root** fixes the derivation to a single history, so a challenge cannot be replayed from another chain. And *Outbox* and *Inbox* are deliberately *not* folded into K: they live inside *RootAfter*, so binding the root binds them. Change any message and its shard's DA chain moves, so K moves, so the challenges move — which is exactly the property the argument needs.

3.9 The global commitment

The root aggregator (`scaling/aggregator.go:27`) maintains a sparse Merkle tree over the shards' subtree roots, with **leaf index = shard id**. Three operations:

`Ingest(header)` moves one shard's leaf to the block's `NewRoot`, refusing outright if `PrevRoot` is not that shard's currently committed root (`aggregator.go:52`). This is the fail-fast the aggregate commit would otherwise have given up: an obviously non-chaining block is rejected at commit time, not merely caught in-proof.

`Split(parent, right, leftRoot, rightRoot)` reshapes one leaf into two, atomically, at an epoch boundary (`aggregator.go:79`).

`Merge(parent, retired, mergedRoot)` is its inverse (`aggregator.go:98`).

3.10 Durability: the journal

Each shard writes an append-only JSON-lines journal with an `fsync` per append, genesis entry first, and a hard refusal to open if the tail is corrupt (`scaling/store/journal.go`). Recovery replays it: `replayPrologue` checks the shard id and `PrevRoot` chaining, and `verifyHeader` re-asserts `NewRoot`, `OutboxCommit`, `InboxCommit` and `DAHash` after re-applying the records (`scaling/recover.go`).

The property that buys is stronger than crash-safety. A journal written by one version of the code and replayed by another fails at the exact block where the two diverge, rather than silently producing a different root — so the journal is both the durability mechanism and a continuous conformance test of the transition function against its own history. It is also the handoff between sequencing and proving: the prover daemon reads the same journals and reconstructs the proving job from them, with no channel from the sequencer.

3.11 A note on domain separation

One property of the encodings above should be stated plainly rather than left for a reader to notice. With two exceptions — the epoch commitment and the challenge derivation, which carry the EPOCHK and CHALBG tags — *none* of these hashes is domain-separated by a tag. The sub-account leaf and the Merkle node function are both eight-element hashes over the same permutation, and the channel root step and the challenge transcript are both nine.

They are nonetheless unambiguous, for a structural reason: every tree in the system has fixed depth, so a leaf is never interpreted at an internal node's position, and no verifier is free to reinterpret a digest as a structure other than the one its position dictates. The classical variable-depth Merkle second-preimage confusion does not arise.

Three commitments are not fixed-shape.. The stronger claim — that every circuit fixes the *arity* of every hash it computes — no longer holds, and it is worth being exact about why the weaker one survives. Three encodings now extend conditionally on their own contents. A message fold carries $(price, expiry)$ only for a priced escrow, so it is 17 or 19 elements; a market shard root folds a book root only on a book-enabled shard, so it is 13 or 17; a record encoding is 19, 32 or 34. In each case the extension is guarded so that the unextended form is byte-identical to what the system committed before the feature existed, which is what lets a book arrive without moving a single already-settled header.

That guard buys compatibility, not unambiguity. Unambiguity has to come from the sponge, and it does — conditionally. H is `hash_n_to_hash_no_pad` at rate $r = 8$ with *no padding*: input is absorbed in blocks of eight by overwriting the first lanes of the state, and a final partial block overwrites only the lanes it occupies. Above the rate this distinguishes length on its own. The state entering the last absorb is a permutation output rather than zero, so appending elements overwrites lanes that were not zero to begin with, and the digest moves.

Below the rate it does not. A single-block input is absorbed into a zero-initialised state, so trailing zeros are indistinguishable from absence, and two inputs that agree on their first n elements and are zero thereafter hash identically. Exhaustively over $n \le 40$ and extensions of up to eight zeros, the collisions are exactly the pairs with $n + k \le 8$: $n = 4$ with $k \le 4$, $n = 5$ with $k \le 3$, $n = 6$ with $k \le 2$, $n = 7$ with $k = 1$, and no others. Length-extension by zeros is invisible strictly inside the first block and detectable everywhere after it.

The three variable-arity commitments are therefore safe, because the shortest of them is thirteen

elements and every alternative form of each is above the rate. This is a real margin, but it is a margin rather than a design: nothing in the type system prevents a future commitment of arity four from acquiring a conditional two-element tail, and that one would be genuinely ambiguous. The invariant to carry forward is narrow enough to state in one line — *a commitment may extend conditionally only if every form it can take exceeds the sponge rate* — and it is not currently checked anywhere.

The same boundary, priced.. Rate 8 governs more than ambiguity. Because absorption is by whole blocks, the cost of a hash is a step function of its arity rather than a slope, and the steps fall in the same place the ambiguity argument does. Measured over `HashSlice`:

Arity	Cost	Permutations
4–8	$\approx 0.80\,\mu$s	1
9–16	$\approx 1.57\,\mu$s	2
17	$\approx 2.36\,\mu$s	3

An extra element is free until it is not, and then it costs 1.97×. So the same number that decides whether a conditional extension is unambiguous also decides whether a field added to a leaf is free or doubles it, and a commitment sitting at arity exactly eight is sitting on a cliff edge. The account leaf was at eight and went over it; Section 6.5 gives what that cost.

Two consequences, one boundary: below the rate, ambiguity; across it, a permutation.

That said, the separation as a whole is maintained by *discipline* rather than guaranteed by *construction*. Adding explicit domain tags to the leaf and node hashes, and absorbing an element count wherever arity is data-dependent, would make the property robust at the cost of one field element per hash. That is the recommendation this paper makes for a production deployment, and the conditional extensions above have moved it from a precaution to a live one.

4 The proof architecture

4.1 Two folds, in two directions

Because each shard produces its own blocks, each has its own prover, and the state tree hands the aggregation its shape for free. But what aggregates is not one thing; it is two, and separating them is the key to reading Figure 4.

Up each shard — the *vertical* fold. A run of consecutive blocks folds into one, each block opening its predecessor's root, so a segment reduces to a segment of segments and finally to a single proof of that shard's history: one start root, one end root, one data chain, and one cumulative cross-shard position. This is the ordinary rollup range proof and needs nothing new.

Across shards — the *horizontal* fold, or the *join*. Those per-shard histories must reconcile with one another: every message one shard consumed was emitted by another, and nothing was created or lost. This is the hard one, because the histories never line up.

Note what the tree's shape implies. Sibling markets never message each other, so *within* the markets subtree there is nothing to reconcile. The messages that carry value run between the accounts subtree and the markets subtree, so conservation is enforced exactly where those two meet.

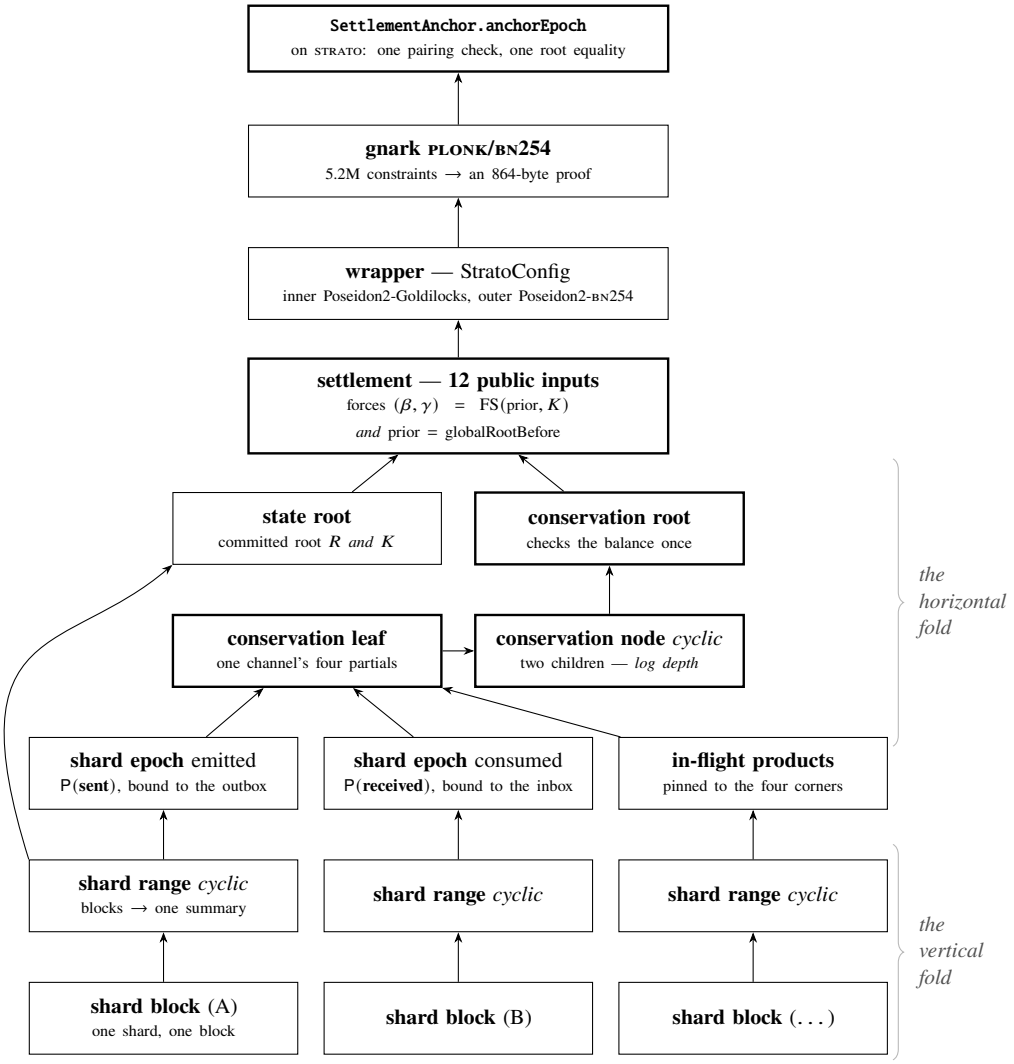


Figure 4. The proof stack, bottom to top. Everything below the settlement is plonky2 over Goldilocks with a Poseidon2 configuration; the two layers above re-prove into BN254 so a pairing check can run on chain. Circuits marked *cyclic* are recursive fixpoints — the same circuit verifies its own output — which is what lets a fold of unbounded depth keep a bounded verifier. The two folds run in different directions and meet only at the conservation leaf: *up* each shard is its own block history, and *across* shards is the conservation that machines which never synchronise must nonetheless satisfy. The curved edge is the state-root circuit reading the same range proofs the conservation side consumes — which is what makes the committed root and the conservation provably one epoch.

4.2 Why nothing lines up, and what holds anyway

A message one shard sends and another consumes land in different blocks. The two shards keep their own cadence and their own clock, and the provers building toward a settled root work over *segments* — ranges of many blocks — not synchronised pairs. So conservation cannot be stated as "this block's send equals that block's receive." Nothing lines up.

What holds is a statement about *sets*, not blocks. Over each shard's whole committed history, the messages it sent equal the messages received plus the messages still in flight. Every send eventually lands in exactly one receive, or is on its way to one. That is an accumulation over history — order-independent, indifferent to which block or which shard or how far apart the two ends are — which is precisely why it survives shards that never synchronise.

Making that provable requires each shard's commitment to carry a running fingerprint of everything it has sent and received, in a form where a send here and its receive there cancel wherever both are finally included. The rest of this section is that fingerprint and the recursion over it.

4.3 Prover configuration and the recursion pattern

All inner proofs are plonky2 [16] — a PLONK-style arithmetisation [9] over a FRI polynomial commitment [15, 14] rather than a pairing-based one, which is what makes recursion cheap enough to do at every layer [17] — over Goldilocks with extension degree $D = 2$ and a `Poseidon2GoldilocksConfig` — a fork with a native Poseidon2 gate rather than stock Poseidon, which is what makes the hash-heavy leaf circuits affordable (`zk/spike/src/lib.rs:24`).

Recursion uses one pattern throughout. A parent verifies its child against `constant_verifier_data`; for a *cyclic* child — one whose circuit verifies its own output — the parent additionally pins the verifier data embedded in the child's public inputs to that constant, connecting both the circuit digest and the Merkle caps. Cyclic templates pad to a fixed degree so that the common circuit data matches the recursion fixpoint (`aggregate.rs:87`). Without both halves of that binding, a prover could substitute a different circuit's proof at a recursion site; with them, the only proof that verifies is one from the circuit the fixpoint names.

4.4 Layer 0: the shard-block circuits

A shard-block proof attests that one shard advanced its own subtree by one block, and exposes the block header directly as 22 public inputs (`shardblock.rs:44`):

$$\big[\mathit{rootBefore}[4],\ \mathit{rootAfter}[4],\ \mathit{number},\ \mathit{ts},\ \mathit{da}[4],\ \mathit{outbox}[4],\ \mathit{inbox}[4]\big].$$

The inventory covers the full escrow lifecycle. `build_shard_block` proves n credit-or-open records on an accounts shard, emitting the escrow; `build_escrow_in` proves a market consuming one `EscrowOpen` at $\mathit{Seq} = \mathit{wantSeq}$, opening the sub-account and moving the pool from zero to the margin, and asserting the escrow admissible; `build_reject` proves its exact complement; `build_credit_consume` proves an accounts shard crediting a returned transfer. Two multi-peer variants matter for the grid topology: `build_multichannel_emit_block`, in which one accounts shard opens on every market peer and exposes a *per-peer* outbox at $\mathit{OUTBOX} + 4j$, and `build_multichannel_consume_block`, its dual.

Each of these opens a Merkle path against `rootBefore`, writes the updated leaf, recomputes `rootAfter` along the same path, and folds the record's canonical encoding into `da` — exactly the Go encodings of Section 3, recomputed in-circuit rather than assumed.

4.5 Layer 1: the vertical fold

The shard-range circuit (`shardrange.rs`) is a cyclic left fold over shard-block proofs, producing the range summary of Section 3.7 as 22 public inputs:

$$\big[\mathit{rootBefore}[4],\ \mathit{rootAfter}[4],\ \mathit{first},\ \mathit{last},\ \mathit{daChain}[4],\ \mathit{outbox}[4],\ \mathit{inbox}[4]\big].$$

A node takes a left child that is a block *or* a range and a right child that is a block, checking continuity: $\mathit{left}.\mathit{rootAfter} = \mathit{right}.\mathit{rootBefore}$ and $\mathit{right}.\mathit{number} = \mathit{left}.\mathit{last} + 1$. Because the summary shape is the fold's fixpoint, a range of any depth reduces to one proof the settlement chain can check.

The two fields beyond the ordinary rollup range shape — the cumulative outbox and inbox at the range's end root — are what let the cross-shard layer above read a range's message position *without opening its root*. That is a small interface decision with a large consequence: the join circuits never need a Merkle path into a shard's state.

4.6 Conservation as a grand product

The statement. Per settlement epoch, the property proven is a **balanced multiset identity**:

$$\mathbf{sent}(e) \ \uplus\ \mathbf{inFlightIn}(e) \ \equiv\ \mathbf{received}(e) \ \uplus\ \mathbf{inFlightOut}(e).$$

The messages live-or-newly-emitted this epoch equal the messages newly-consumed or still in flight. A dropped send sits on the left with no match; a fabricated receive sits on the right with no match. Either breaks the identity.

Note what this buys over the naive formulation. Requiring **sent** = **received** is only true at a *drain point*, when nothing is in flight — and in an asynchronous system that moment may never arrive. Carrying in-flight explicitly makes conservation a property that holds *continuously*, at every epoch boundary, however far the consumer lags.

The mechanism. The construction is the permutation argument of PLONK [9], in the form *plookup* [10] generalised it: a multiset is compressed to a single field element by a randomised product, so that multiset equality reduces to one field equality. What is specific here is not the technique but what it is applied to — message histories on shards that never synchronise — and how its inputs are bound, which is Section 4.7. A message compresses to one field element, and a *set* of messages to one product:

$$\varphi_\beta(m) = \sum_{j=0}^{12} v_j(m)\,\beta^j, \qquad \mathsf{P}_{\beta,\gamma}(\mathbf{S}) = \prod_{m\in\mathbf{S}} \big(\gamma - \varphi_\beta(m)\big),$$

over the thirteen-element vector of Section 3.4 (`multiset.go:77,91`). The product is **order-independent**, which is the whole point: it composes across blocks, segments and shards by plain multiplication, so two histories that fold their messages in different orders land on the same value. Multiset equality becomes one field equality:

$$\mathsf{P}(\mathbf{sent}) \cdot \mathsf{P}(\mathbf{inFlightIn}) \ =\ \mathsf{P}(\mathbf{received}) \cdot \mathsf{P}(\mathbf{inFlightOut}).$$

Scaling to a whole exchange. The identity is stated per directed channel, but the product is multiplicative, so the system's conservation is just the product of every channel's. A message on the $A \to B$ channel carries its source and destination *in its fingerprint*, so it can never be matched by a

receive on another channel: cross-channel fabrication is a distinct fingerprint that unbalances the product. Each channel keeps its own in-flight bound to its own four corners, so in-flight cannot be shuffled between channels to fake a balance. And one shared (β, γ), derived from a commitment over *every* shard's range, ties them together, so a prover cannot take a favourable challenge on one channel and a different one elsewhere.

4.7 The three bindings

The balanced identity means what it says only because three separate bindings hold. Each turns a would-be cheat into an unprovable statement, and each is a distinct circuit-level mechanism.

Binding I — sent and received are tied to their ranges. The message-product circuit (`shardproduct.rs:46`) does two things over the *same* witnessed field vectors: it accumulates the product, and it accumulates the `Message.fold` hash chain. Its public inputs are

$$\big[\beta[2],\ \gamma[2],\ \mathit{product}[2],\ \mathit{chainBefore}[4],\ \mathit{chainAfter}[4]\big],$$

the bracketed counts being limbs: the challenges and the product are $\mathbb{F}_{p^2}$ elements, the chains are digests.

The shard-epoch circuit (`shardepoch.rs`) then verifies a range proof and a product proof together and connects the product's chain endpoints to the range's committed outbox (or inbox): `chainBefore` to the shard's chain at the epoch's start, `chainAfter` to the range's endpoint. Since the product and the chain share their message vectors, and the chain must land on a value the range's own proof committed, **the product is proven over exactly the messages the range's published data emitted — no others**. A product over invented messages would not land on the committed chain.

Binding II — in-flight is pinned within an epoch. The tempting mistake is to let the prover supply the in-flight set as a side input, which it could then inflate or fabricate. Within one epoch, it need not. The four message chains of an epoch form a commutative square (Figure 5): in-flight on a channel is the emitter's messages the consumer has not yet reached, so folding the in-flight set onto the consumer's inbox reproduces the emitter's outbox.

Each in-flight product carries a chain of its own, from a *chainBefore* to a *chainAfter*, and both ends are pinned to corners of the square (`conservationtree.rs:197-203`). That forces the product to be over exactly the $[\mathit{wantSeq}, \mathit{nextSeq})$ suffix which the committed counters delimit. A junk message injected to rebalance would not connect the corners; a dropped one leaves a gap the empty chain cannot span.

The four corners are not symmetric in how they are established, and the asymmetry is worth following through, because it is what connects this binding to the epoch boundary. The *end* corners are the range proofs' committed endpoints, so $\mathsf{P}(\textbf{inFlightOut})$ is pinned directly to proven values. The *start* corners are exposed as public inputs of the epoch proof (`shardepoch.rs:104,157`) — and what forces *those* to be the real prior frontier is not a constraint in this circuit but the state continuity of Section 4.10.

Binding III — the challenges cannot be chosen. A grand product is sound only if β and γ are unpredictable when the messages are picked; otherwise a prover grinds two unequal sets that collide at a known γ. So the challenges are Fiat–Shamir outputs [6] over a commitment to the epoch's own blocks — settlement is two-phase, as Section 3.8 describes — and the join re-derives

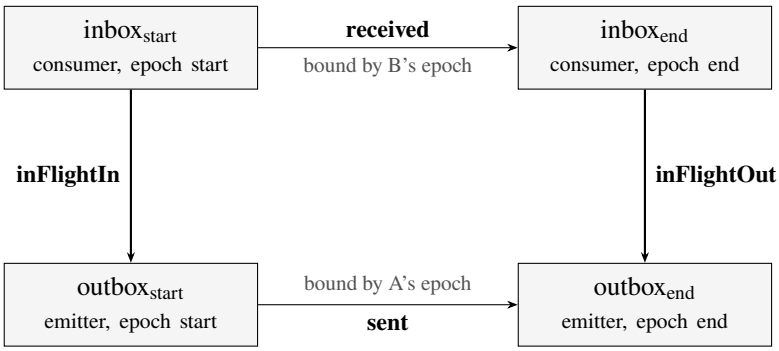


Figure 5. The commutative square that pins the frontier. In-flight on a channel is the emitter's messages the consumer has not yet reached — the suffix past its inbox chain — so folding the in-flight set onto the consumer's inbox *reproduces* the emitter's outbox. Both routes from inbox$_{start}$ to outbox$_{end}$ therefore carry the same multiset, and that equality *is* the conservation identity. The two horizontal chains are already committed by the range proofs; pinning the two vertical products' endpoints to those four corners forces each in-flight set to be exactly the un-consumed suffix. A junk message injected to rebalance would not connect the corners; a dropped one leaves a gap an empty chain cannot span. No frontier commitment of its own is required — and because epoch e's start corners are epoch $e-1$'s end corners, the frontier is carried across epochs by construction.

$(\beta, \gamma) = \mathrm{FS}(\mathrm{prior}, K)$ *in-circuit* from the committed ranges and rejects any product taken at a different one (`multisetjoin.rs`, `settlement.rs:70`).

Change any message and its shard's data chain moves, so K moves, so the challenges move. A challenge cannot be fixed in advance.

Three bindings, three attacks that no longer construct. I. Sent and received are bound to their ranges' committed chains — no product over messages a range never carried. **II.** In-flight is bound to the epoch's four corners — no inflated or dropped frontier; across epochs the start corners are fixed by state continuity (Section 4.10). **III.** The challenges are bound to the epoch commitment — no hand-picked γ. Remove any one and the argument collapses; all three are implemented and pinned to the Go specification by conformance vectors.

4.8 What the fingerprint deliberately does not carry

Binding I rests on the chain and the product being taken over the same thirteen-element vector. A limit order breaks that symmetry, and since the break is deliberate it is worth showing exactly where the safety comes from instead.

A priced escrow carries a resting price and an expiry. The chain fold appends them; the fingerprint does not (Section 3.4). The consequence is directly measurable. Take two escrows agreeing on every field including *Src*, *Dst*, *Seq*, *Amount*, *Side* and *Size*, and differing only in resting price:

$\varphi_\beta(m_{100}) = \varphi_\beta(m_{999})$	equal
Conserved$(\{m_{100}\}, \{m_{999}\})$	**accepts**
fold$(\mathbf{0}, m_{100})$ = fold$(\mathbf{0}, m_{999})$	differ
control: the same pair differing in *Size*	rejects

So the grand product, on its own, does not notice that a market shard credited an escrow at a price the accounts shard never escrowed against. The control confirms this is specific to the two unfingerprinted fields rather than a defect in the join.

What catches it is Binding II, and specifically the fact that Binding II is discharged at the conservation *leaf* rather than anywhere above it. The leaf pins each in-flight product's chain endpoints to the epoch corners (`conservationtree.rs:197-203`), and the two sides bind to different range endpoints: the consumed side to the range's inbox, the emitting side to its outbox (`shardepoch.rs:57,65`). The in-flight-out product therefore spans the consumer's inbox endpoint to the emitter's outbox endpoint. On a drained channel that product is empty, its chain cannot move, and the two endpoints are forced equal; on an undrained one they differ by exactly the tail the same product folds. Either way the consumer's inbox chain is not free. Since the chain does carry the terms, m_{999} cannot be folded where m_{100} was emitted.

The substitution is thus caught one level below where a reader looks for it. This is also why exposing only the four products upward — the aggregation node's six-element interface of Section 4.12 — costs nothing: the corners have already done their work by the time the leaf reports, and carrying them higher would grow every interior node without adding a constraint.

Why not simply fingerprint the terms. Adding two fields to $\vec{v}(m)$ would close the gap at the level a reader first looks, which is a real readability argument. It would also make the vector's length depend on the message, and $\vec{v}(m)$ is the one encoding whose fixed arity every product circuit is compiled against — one shape, shared by accounts and market shards and by priced and unpriced escrows alike. A conditional fingerprint would mean two product circuits and two conservation trees, or a padding convention with its own ambiguity to argue about. Keeping the vector at thirteen and letting the chain carry what varies is the cheaper of the two, given that (Src, Dst, Seq) already makes a message unique and the fingerprint's only job is to separate distinct messages.

The cost is that the argument no longer fits in one mechanism, and a reader who checks the product and stops will conclude something false. That is a documentation hazard rather than a soundness one, but it is a real one, and it is the reason this subsection exists.

One thing must *not* be relied on here. It is tempting to argue that a user's signature over the order binds the price upstream, making the question moot. It does not: ingestion authentication is off by default (`scaling/node/auth.go`) and is intended to sit at the gateway, so the node trusts its ingestion boundary. The leaf binding is what makes this safe, and it is the only thing that does.

4.9 The soundness bound, and the field it rests on

The bindings establish that the products are over the right messages at the right challenge. What remains is the probabilistic step: does the product identity actually imply multiset equality? It does,

with a bound worth computing rather than gesturing at.

Let N be the total number of messages across the four multisets of an epoch, and let $p = 2^{64} - 2^{32} + 1$.

Fingerprint collisions. For fixed distinct $m \neq m'$, the difference $\varphi_\beta(m) - \varphi_\beta(m')$ is a nonzero polynomial in β of degree at most 12, so it vanishes for at most 12 values. Over a uniform β and a union bound across all $\binom{N}{2}$ pairs, the probability that any two distinct messages share a fingerprint is at most $6N^2/p$.

Product collisions. Given distinct fingerprints, both sides of the identity are monic polynomials in γ of degree at most N. If the multisets differ, their difference is a nonzero polynomial of degree at most N, so by the Schwartz–Zippel lemma [7, 8] a uniform γ satisfies it with probability at most N/p.

Together the soundness error is at most $(6N^2 + N)/p$, dominated by the quadratic term.

Epoch size N	**over** $\mathbb{F}_p$ (superseded)	**over** $\mathbb{F}_{p^2}$ **deployed**	
2^4	2^{-53}	2^{-117}	what the drivers prove today
2^{12}	2^{-37}	2^{-101}	
2^{16}	2^{-29}	2^{-93}	
2^{20}	2^{-21}	2^{-85}	a busy second of settlement

Table 2. Soundness error $6N^2/p^{\,d}$ of the grand-product argument. The right column is the deployed instantiation, over the quadratic extension; the left is what an earlier version of these circuits provided over the base field, kept because it is the reason the extension is the right choice. Both columns use the same union-bound accounting.

Why the extension field, and what the base field cost. This paper's first draft described a base-field instantiation as an open weakness. It has since been closed, and the analysis is kept because it is the reason the current design is the right one rather than an arbitrary choice.

What the base field gave. β and γ were single Goldilocks elements driving base-field `mul`/`add`/`sub`, with no extension lift, no proof-of-work on the transcript and no repetition — one draw in $\mathbb{F}_p$. Two consequences followed, and the second is the one that bit.

First, no standard security level at any epoch size. A single draw in a 2^{64} field floors the error near 2^{-64} even for $N = 1$ and degrades quadratically. For any epoch a production exchange would settle, that is below an 80-bit line.

Second — and this is the rebuttal a reader should not accept — two-phase Fiat–Shamir does not close grinding. The stated rationale is that β is bound to K over the epoch's own blocks, so a prover cannot know the challenge before committing the messages. True, and it stops *pre-computation*. It does not stop *grinding*, because of an asymmetry between the two objects: K folds each range's DA chain, which is **order-sensitive** over the records within a block, while the grand product is a **multiset** and is **order-independent** by construction. A prover — who is also the sequencer — could therefore permute records within a block, or pad the epoch with empty blocks, changing K and hence β while leaving the multiset it must cheat on *identical*. Each re-derivation costs one Poseidon hash, so the search is offline and cheap: an adversary spending M hashes wins once $M \cdot 6N^2/p \gtrsim 1$, that is above

$$N \;\approx\; \sqrt{p/6M},$$

which at $M = 2^{30}$ — a laptop running for hours — is $N \approx 53{,}000$ messages per epoch, and at $M = 2^{40}$ is $N \approx 1{,}700$. Both sit inside this design's own throughput envelope: at the 10^5 transitions per second of Section 6, a one-second settlement epoch reaches the first.

The fix, and it is deployed. β and γ are now elements of the quadratic extension $\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2-7)$ — the same non-residue plonky2 uses for its own degree-two extension, so the Go reference and the circuits agree by construction. One transcript hash yields both: $\beta = (t_0, t_1)$, $\gamma = (t_2, t_3)$, so all four limbs bind the challenge (`challenge.go:59`). The fingerprint and product lane runs on extension arithmetic while the message-fold hash chain stays in the base field, which is why the layouts that carry a challenge or a product widened by exactly two slots per field and the purely hash-carrying ones did not change at all (Section B).

The bound becomes $6N^2/p^2$: 2^{-85} at $N = 2^{20}$, which no grinding budget reaches. The cost was close to nothing — the settlement circuit grew by 144 constraints out of 5.2 million, and a conservation node by about 9% of its proving time (Section 6.6). A uniform ≥ 100-bit claim across the entire envelope rather than at realistic epoch sizes would be $\mathbb{F}_{p^3}$, at proportionally more cost on the same lane.

Nothing else in the stack was affected: the range, state-root and block layers are hash-based and their soundness is the hash's.

4.10 Binding epochs to one another

The bindings of Section 4.7 establish conservation for one settlement epoch. The claim the design actually makes is stronger — that conservation holds *continuously*, epoch after epoch, with the frontier carried from each to the next — and getting from the first to the second turned out to need exactly one additional constraint.

State continuity. The settlement pins the prior settled root to the global root folded over the ranges' *starting* roots. The state-root circuit exposes that value as `globalRootBefore` (`stateroot.rs:28`), computed in the same sparse tree and with the same hashing as the committed root it exposes alongside it, and the settlement connects the two (`settlement.rs:70`):

$$priorSettledRoot \;=\; globalRootBefore(e).$$

Since the contract advances only when `priorSettled` equals its current settled root, and its current settled root is $globalRoot(e-1)$, this says $globalRootBefore(e) = globalRoot(e-1)$: epoch e *provably begins where epoch* $e-1$ *ended*. Before this constraint existed, the chain of anchored epochs established root-chaining and replay-protection but not continuation — a proof named its predecessor's root without being obliged to have started from it.

And the frontier follows, at no extra cost. The satisfying part is what does *not* need building. It is tempting to add a second mechanism that carries each channel's corners forward and checks them at the settlement. That would be a mistake, and an expensive one: a conservation node's public interface is bounded precisely because products *fold* and corners do not, so a node carrying corners would grow to $O(c)$ in the channels beneath it and destroy the property the whole tree exists for.

It is also unnecessary, because the frontier binding is already implied by two things the design has. A shard's outbox is *cumulative*: the block circuit seeds its fold from `chan.out_chain`, the channel-leaf pre-state it opens against `rootBefore` (`shardblock.rs:658-661,668`). And Binding I forces the emit product's chain to land on the range's committed outbox. So once state continuity makes `rootBefore`(e) the

prior committed state, the prior corner is forced to the real prior frontier — by the *same* collision resistance that makes Binding I sound, and on no new assumption.

The design already contained the fix. **inFlightIn**(e) runs between the consumer's inbox and the emitter's outbox at *rootBefore*(e); **inFlightOut**$(e-1)$ runs between the same two chains at *rootAfter*$(e-1)$. Continuity makes those roots equal, so the two products are over the same endpoints and therefore the same message set. **inFlightIn**(e) = **inFlightOut**$(e-1)$ falls out as a *consequence* — no carry object travels between epochs, nothing is folded up the tree, and the bounded-node property is untouched. The only thing ever missing was the constraint that ties one epoch's starting state to the last one's ending state.

A negative test drives a real two-epoch carry and pins both directions: an honest carry proves, and a fabricated **inFlightIn** does not (`zk/circuits/tests/multi_epoch_carry_pipeline.rs`). Conservation is therefore proven continuously, across epochs, and not only within one.

4.11 The joins, in three generalisations

Three join circuits exist, and they form a strict generalisation chain rather than alternatives.

The range join (`rangejoin.rs`) — exact, drain-point only. Verifies two shards' range proofs and enforces $a.outbox = b.inbox$ and $b.outbox = a.inbox$ as exact folds. Because the folds are over each shard's message *sequence* and not its blocks, the two ranges need not share block numbers, counts or boundaries — the asynchrony is already tolerated. What is not tolerated is anything still in flight: the equality holds only at a drain point.

The multiset join (`multisetjoin.rs`) — continuous, one channel. Replaces that equality with the balanced product, so a message emitted but not yet consumed rests in **inFlightOut** and the epoch conserves regardless. It verifies the emitted-side and consumed-side epoch proofs and the two in-flight products, all at one shared (β, γ) it derives in-circuit. The range join is its drain-point special case, and fold-versus-product is precisely order-dependent-versus-order-independent.

The multichannel join (`multichanneljoin.rs`) — the whole frontier. Checks the identity over every channel at once, in one circuit, with one shared challenge derived from a commitment over every shard's range.

The multichannel join carries one mechanism worth naming, because it is what makes the shared-accounts-shard grid provable at all. A multi-peer shard contributes *one* range but appears in *several* channel epochs. Folding its summary once per channel would corrupt K. So the join deduplicates (`multichanneljoin.rs:153`): each distinct shard is forced to expose an *identical* range summary across all the channels it appears in, and that summary is folded into K exactly once. Without it, the epoch commitment an $N \times K$ grid derives would not match the one the state-root circuit computes over the same ranges, and the settlement's equality check would fail — which is to say the dedup is load-bearing, not an optimisation.

4.12 The conservation tree: where the log depth is

Verifying every channel in one circuit — what the flat multichannel join does — grows without bound in the channel count. The conservation tree (`conservationtree.rs`) replaces it with a recursion whose every node is the same bounded size.

A **leaf** carries one channel's four partial products at the shared (β, γ), and performs the frontier binding of Section 4.7. A **node** verifies two children and multiplies their four partials lane-wise, connecting the children's challenges equal so the whole tree sits at one challenge. Its public inputs are just six values:

$$\big[\beta,\ \gamma,\ \mathsf{P}(\textbf{sent}),\ \mathsf{P}(\textbf{received}),\ \mathsf{P}(\textbf{inFlightIn}),\ \mathsf{P}(\textbf{inFlightOut})\big],$$

twelve field elements in all, each of the six being a two-limb $\mathbb{F}_{p^2}$ value. The **root** checks the single balance once, at the top.

Three properties follow, and they are the reason this architecture can scale at all. Because the product is order-independent, the tree's shape carries no meaning — provers may fold in any order. Because it is multiplicative, a single non-conserving channel anywhere in the forest unbalances the root, and no root proof exists. And because every node is the same bounded circuit, total work is linear in the channel count while the *critical path* is the tree depth — logarithmic. Section 6.6 measures both, and finds the second to be a property of a *fleet* rather than of a machine.

Note that this is the very machinery a shard already uses to fold its own history: the same cyclic tree, turned from folding a chain of blocks to folding a forest of conservations.

4.13 The state root: tying the proof to the committed root

The state-root circuit (`stateroot.rs`) verifies each shard's range proof, reads its committed subtree root (the range's `rootAfter`), places it at leaf index = shard id, and folds those roots into *the same sparse tree the aggregator computes* — same empty-leaf convention, same T. From the same verified ranges it also folds the epoch commitment K. Its two public outputs are therefore exactly what settlement needs:

$$\big[\mathit{globalRoot}[4],\ \mathit{epochCommit}[4]\big].$$

So the proof's own output root *is* the root already on chain, and settlement is one equality rather than a translation between two representations.

4.14 Settlement: one proof, one epoch

The settlement circuit (`settlement.rs`) is the weld. It verifies the state-root proof and the conservation proof, and then does the one thing that makes them a single statement: it derives

$$(\beta, \gamma)\ =\ \mathsf{H}\big(\textsc{chalbg},\ \text{priorSettledRoot},\ \mathit{epochCommit}\big)$$

from the *state root's* K, and **forces** the conservation proof's (β, γ) to equal it. In the same breath it pins the prior settled root to the state root's `globalRootBefore`, which is the state-continuity constraint of Section 4.10.

The challenge constraint is what makes the pairing impossible to break apart. A valid settlement proof cannot pair one epoch's root with another epoch's conservation: change a message and its

shard's data chain moves, so K moves, so the challenge moves, and the conservation stops fitting the root. The continuity constraint does the same job across time. Its twelve public inputs are the anchor's whole interface:

$$\big[\mathit{globalRoot}[4],\ \mathit{prior}[4],\ \beta[2],\ \gamma[2]\big],$$

β and γ being $\mathbb{F}_{p^2}$ elements of two limbs each.

The contract checks one proof and one root equality. The root's validity, its conservation, their being the same epoch, and its continuing from the epoch before are all bound inside it.

4.15 The BN254 tail and on-chain verification

A Goldilocks FRI proof cannot be verified by a pairing check, so the settled proof is re-proven twice more. First a **wrapper** (`wrapper.rs:43`) re-proves it under `StratoConfig`: the inner algebraic hasher stays Poseidon2-Goldilocks, but the Merkle and leaf hasher becomes Poseidon2-BN254, and the FRI shape switches to a high-rate configuration (rate 7, cap height 4, 12 query rounds) chosen to minimise the on-chain verifier's work. `build_wrapper_plain` re-exposes *all* of an inner circuit's public inputs, which is what makes the ten-input settlement anchorable with no new circuit written for it.

That wrapped proof is then compiled to a PLONK circuit [9] over the pairing-friendly Barreto–Naehrig curve BN254 [13] using gnark [20], and proven there, yielding an 864-byte proof a pairing check verifies. The constant size is what a PLONK-style argument over a KZG polynomial commitment [12] buys, and it is why this last step trades a transparent setup for a structured reference string. `SettlementAnchor.anchorEpoch(globalRoot, priorSettled, β, γ, proof)` verifies it on chain and advances the settled root *only* when `priorSettled` equals the contract's current settled root — so settlement is root-chained, and a replayed proof against a stale prior reverts.

The contract is deployed on a STRATO testnet, with three epochs chained on chain and the third anchored end-to-end by the prover daemon with no human in the loop. Section 6.6 gives the cost of that tail, and Section 7 states what its soundness rests on.

4.16 What uniformity bought

One implementation decision deserves separate mention because it is what makes the whole tree assemble.

The shard-block and epoch circuits *witness* their shard, market and counterparty identifiers — range-checked to 32 bits — rather than baking them in as constants (`shardblock.rs:355`). The consequence is that every shard pair, whatever its identifiers, shares the same two range circuits and the same single conservation-leaf circuit.

Without that, each shard pair would be a distinct circuit with a distinct verifier. A *non-cyclic* parent could still verify several such children, one `constant_verifier_data` apiece — so the sharp statement is not that folding becomes impossible but that it stops being *cyclic*: a recursion fixpoint verifies exactly one circuit, and a pile of distinct verifiers is a circuit whose size grows with the shard count.

What uniformity really buys is therefore a *dynamic* shard count. One witnessed-id circuit means one verification key, and any number of instances fold in the same cyclic node with no per-shard verifier anywhere. That is what lets the cluster grow — as an elastic split makes it — without deploying a new circuit for the shard that appears.

The same principle later absorbed a second axis of variation, which is the better evidence that it was the right one. A market shard performs several kinds of block — resting an order, filling, cancelling, settling — and the obvious construction gives each its own circuit and a wrapper that verifies whichever one applies. That wrapper is a per-kind verifier, and it fails in exactly the way a per-shard verifier does: the parent grows with the number of kinds, and the recursion stops being cyclic. The implementation instead makes the market block one *kind-selecting* circuit that witnesses which operation it is performing and constrains accordingly, folded by the ordinary shard range — the same range circuit that folds accounts blocks, with no market-specific recursion layer between them.

So the fold is uniform across shard *kinds* as well as shard identifiers, and the tree of Section 4.12 does not know whether the range beneath it came from an accounts shard or a market shard running a book. A design that had made the shard kind structural rather than witnessed would have needed a second tree to find that out.

Selecting a value away does not select its constraints away. Uniformity of this kind carries a correctness obligation that the separate circuits it replaces do not have, and it is worth stating because the failure is silent and general rather than specific to markets.

When each operation is its own circuit, every slot in it is live by construction: a circuit that opens a position always opens one. A constraint on that slot can therefore be unconditional. Under a kind selector the arithmetic of *every* branch is laid down in one circuit and evaluated on every block, while only one branch's result is selected into the output. A slot that is selected away is still evaluated — over whatever happens to be in the leaf.

That turns a constraint which was always satisfiable into one which is sometimes unsatisfiable. The position-open path pins a floored division by witnessing the quotient and requiring $q \cdot size + r = num$ with $r < size$. On a block that opens nothing, the slot's size is zero, and $r < 0$ has no solution: a correct prover cannot satisfy a circuit describing an operation it did not perform. The constraints had to become conditional on the same selector that chooses the value (`marketblock.rs:687`) — an `assert_if` on the branch's own liveness rather than an `assert`.

The general form is that `select` removes a value from the *output*, not from the *constraint system*. Anything a branch asserts must be guarded by that branch's selector, and a range check is as much an assertion as an equality. What makes this worth a paragraph is when it appears: not while each primitive is developed and tested on its own, where every slot is live and the bug is invisible, but at the moment the circuits are unified — which is exactly when the surrounding change is being judged on whether the fold still composes, and attention is somewhere else.

---

5 Elasticity: splitting a shard without breaking conservation

A fixed topology answers the scaling question but not the operational one. If one accounts shard's load grows too large, the cluster must be able to divide it — and to rejoin the halves when load falls — without pausing the exchange and without opening a hole through which value could leak. This section gives that mechanism and the proof that attests it.

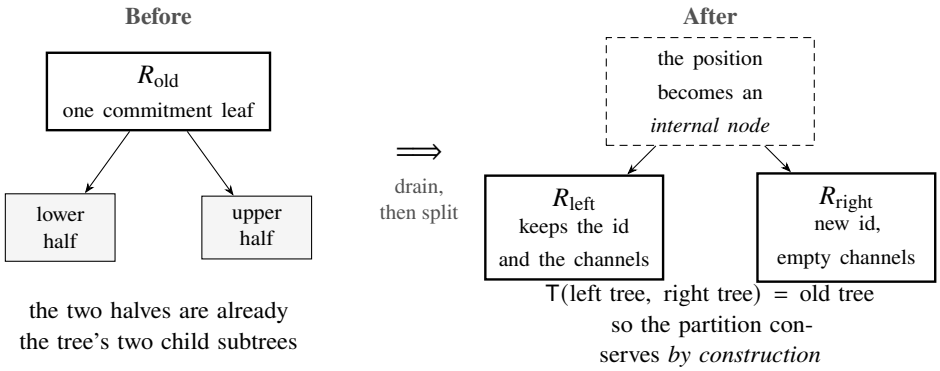


Figure 6. The elastic split. Because leaf hashes carry no position, the old tree's two child subtrees *are* the two halves — so splitting is not a migration but revealing one more level at that spot, and the parent commitment is reproduced exactly. Inbound channels are drained first, so no message crosses the boundary and the re-routing proof that a live split would need disappears entirely. The aggregator reshapes one commitment leaf into two, and that new leaf *is* the new shard's registration: markets learn of the split by reading the committed root, not from a trusted announcement.

5.1 Drain first, and the re-routing proof disappears

When an accounts shard splits, it splits at the midpoint of its owned range: the left half keeps the shard's identity, its channels and the lower range; the right half becomes a new shard over the upper range.

The tempting design lets messages keep flowing and re-routes whatever is in flight to the new shard as it goes. That re-routing is a real and separate proof obligation, because a message re-addressed to a new destination with a fresh sequence number is a *different message* — and proving that the new one stands for the old one is exactly the kind of correspondence the conservation argument is built to refuse.

The design takes the other branch: **drain the inbound channels first**. Only the inbound needs to quiesce — outbound refunds keep flowing under the rename — and a channel is drained precisely when its committed cursors say so, which is visible to anyone reading the roots. With nothing in flight toward an account that is about to move, the split carries no messages across itself and the whole re-routing proof disappears.

5.2 Why the partition conserves

A split creates and destroys nothing: the same accounts, re-homed. The statement that captures this is that the position the old shard's commitment leaf occupied becomes an *internal node* whose two children are the new shards' roots:

$$tree_{\text{old}} \;=\; \mathsf{T}\big(tree_{\text{left}},\; tree_{\text{right}}\big).$$

This holds because *leaf hashes carry no position*. An account leaf is $\mathsf{H}(owner, free, nonce)$ and nothing else (Section 3.2), so the old tree's two child subtrees literally *are* the two halves. Splitting is revealing one more level at that spot. A partition that lost or forged an account could not reproduce the committed parent.

The Go specification checks this invariant before returning (`accounts.go:155`); a violation is treated

as a bug, not a user error.

Why the midpoint, and only the midpoint. The split key must be the range midpoint, $base + 2^{depth-1}$. A balanced Merkle tree divides into two *whole* subtrees only there, and that is what keeps the partition a single T rather than an $O(depth)$ boundary-path argument.

This is less restrictive than it sounds. Any dyadic partition is reachable by splitting a half again at its own midpoint, so an explicit split key drives *multi-level* splits and the cluster can reach any power-of-two subdivision. And with account identifiers derived by a hash across the whole space (Section 3.2), a non-dyadic boundary balances no load that a midpoint split does not — so the restriction costs nothing the design wanted.

5.3 The cutover is atomic, and the structure enforces it

Because an account lives at a fixed position in the tree, the shard that owns the *subtree* is the only party that can prove a Merkle path to that account. So a message for an account in the upper range is only *consumable* by whoever owns the upper subtree, which after the split is the new shard.

A market that is slow to notice the split therefore cannot misroute an upper-range message to the old shard: the consume would be *unprovable*, its Merkle path in a subtree that no longer holds the leaf. The peer set flips exactly at the committed epoch and nothing can straddle it. What does transition gradually — the "eventually everyone talks to the new shard" — is only mempool routing of new user transactions, a liveness convenience the old shard can forward, with no bearing on conservation.

5.4 Registration by derivability

The right shard's genesis root is the upper subtree of the old shard's last committed root: a value anyone can compute from the settlement chain and the split key. Handing the new sequencer its slice of the state is therefore a *liveness* optimisation — it gives that sequencer the leaves to serve from — while the *correctness* of where it starts is anchored in the commitment, not in the transfer.

The split appears on chain as the aggregator reshaping one leaf into two (`aggregator.go:79`), and that new committed leaf *is* the new shard's registration — the thing markets read to learn the peer set changed. No side channel, no trusted announcement.

5.5 What the split circuit proves

`build_split` (`split.rs:125`) attests, against committed roots, exactly the three things the drain-point design rests on. Its four public inputs are the roots a verifier checks — the parent the aggregator replaces, the two it becomes, and the market the drain is proven against:

$$\big[OLD[4],\ LEFT[4],\ RIGHT[4],\ MARKET[4]\big].$$

1. **The state partition** (`split.rs:158`): $old = \mathsf{T}(left, right)$, with *leftRoot* recomputed as the accounts-shard root over the left tree and the *inherited* channel root, and *rightRoot* over the right tree and the *empty* channel root.
2. **The new shard starts fresh** (`split.rs:151`): the right shard's channel root is recomputed *in-circuit* as the empty channel leaf — H of eighteen zeros — over the same market peer. It is proven, not witnessed, so a splitter cannot hand the new shard a channel carrying fabricated in-flight value.

3. **The inbound was drained** (`split.rs:164`): the accounts shard's inbound chain equals the market's outbound chain on that channel — the channel reconciliation — so nothing was in flight toward an account that moved. Crucially the market's *full* committed root $\mathsf{H}(sub, mark, pool, chan)$ is folded in circuit, so the drain is tied to the market's committed state rather than to a free witness.

Its negatives hold and are tested: an undrained channel is unprovable, and a swapped partition cannot reproduce the parent (`zk/circuits/tests/split_pipeline.rs`).

5.6 Merge is the split run backwards

When load falls, an adjacent, fully-drained sibling folds back in: the tree returns to its pre-split root and the commitment collapses two leaves into one (`accounts.go:222`, `aggregator.go:98`). The retiring shard must be drained in *both* directions, not just inbound, because its identity and its channels disappear — unlike a split, where only the inbound quiesces and the new shard keeps its own fresh channels.

Because a drained merge yields the very triple a split does — a parent and its two children, with the same T relation — *the same circuit attests it*. There is no second circuit for merge, which is a small result but a satisfying one: the mechanism was symmetric all along.

5.7 Market re-registration

A market must respond to a split, or a settlement for a moved account strands at a shard that no longer holds it. `ReRegisterSplit` (`market.go:129`) opens a channel to the new shard and re-homes the sub-accounts whose account moved above the split key, so a later settle addresses the shard that *now* owns the account rather than the one that opened the escrow.

This is the market reading the committed split — the new leaf and its key — not a trusted side channel.

Scope of what is built. The split is a working, tested prover checked against the specification's own digests, with negative tests showing that freshness and drain are load-bearing. Two limits are real and should not be read past. It proves a *single* market peer — the shape of the first slice — and a multi-peer split would fold one such market leaf per market, which is not built. And it is a standalone circuit plus the Go partition: nothing in the running daemon pipeline performs a live split. The mechanism is proven; the operations that would invoke it are not written.

---

6 Empirical results

6.1 Method, and a benchmark that lied

Every number below was measured on one Apple M4 with ten cores and 24 GB of memory, at repository commit `ea2f36d`. Execution figures are the best of three runs; proving figures are single runs of an opt-in benchmark, with the reasons given in Section 6.6. Section C gives the exact commands.

What these numbers describe, and how that was checked. The measurement commit matters more than usual, because the implementation moved after it: the market execution path since gained continuous match-on-insert and a book-enabled block circuit. Every figure below that drives a market shard was therefore re-measured at the later commit `9ab2bba` and, where the two disagreed, settled by a controlled comparison rather than by argument (Section 6.5).

The conclusion is that the sharding results are unchanged. Independent-pair throughput and the fan-out wall reproduce within run-to-run noise, with identical efficiency curves. The fixed-total grid appeared $\approx 7\,\%$ lower on the reference machine, but running both commits interleaved on a dedicated sixteen-core machine returns the same speed-up from each, so that difference was the measuring environment and not the code.

One real cost did surface: a uniform $\approx 2\,\%$ drop in absolute throughput, traced to a single encoding change, priced in Section 6.5, and since recovered by re-encoding the field responsible. The figures in this section are therefore representative of the implementation as it stands, having briefly not been.

Before the numbers, the correction. The committed throughput harness originally started its clock, then submitted every shard's workload from a *single* goroutine, shard by shard. `Node.Submit` takes the node's mutex; the block loop's `Tick` holds that same mutex through its entire mempool drain and seal, and the drain empties the whole mempool in one tick. So the submitter blocked behind each shard's multi-hundred-millisecond drain before it could feed the next, and the measured elapsed time became approximately the *sum* of the shards' drains — the signature of a serial system.

K	**serial submitter**	**concurrent submitter**
1	100 %	100 %
2	69 %	99 %
4	37 %	93 %
8	24 %	59 %

Table 3. Scaling efficiency of K independent shard pairs, before and after fixing the workload submitter. Nothing about the system changed between these two columns.

The tell was processor utilisation: at $K = 8$ the run consumed 14.4 s of CPU across 8.8 s of wall-clock — 1.66 cores busy on a ten-core machine, while sixteen supposedly independent shard processes ran. The fix is to build the operations off the clock and feed every shard concurrently, one goroutine per node with a single atomic batch submission each. Nothing else changed: not what is counted, not the drain, not the seal.

This is recorded rather than quietly corrected because it is the failure mode every distributed-systems measurement is exposed to — a harness that serialises the thing it is measuring — and because the difference between 24% and 59% is the difference between a design that does not work and one that is saturating its hardware.

6.2 The primitives everything is built from

Before the system-level curves, the two costs that explain all of them (Go benchmarks, `-benchtime 2s`):

Operation	Cost	What it is
One Poseidon2 compression (T)	1.082 μs	the Merkle node function
Depth-24 sparse-tree leaf write	20.99 μs	$\approx$ 24 compressions
Root read (cached)	3.64 ns	no recomputation

Table 4. The primitives. A state write is 24 hashes, and a hash is a microsecond.

The execution layer is hash-bound, and nothing else is close. A depth-24 state write costs 24 Poseidon2 compressions and essentially nothing else. Every throughput number in this section is, to within a few percent, a count of tree writes multiplied by 21 μs. That single fact is worth holding while reading the rest: the shard runtime is not bound by matching, by serialisation, by the transport, or by contention. It is bound by hashing the state tree — which is a property of being a rollup, not of being sharded.

6.3 What a cross-shard boundary actually costs

The central design worry is that messaging overhead eats the parallelism. It is directly measurable. Each row below is a single accounts-shard/market-shard pair running one operation class to exhaustion, best of five:

Operation	Count	Per op	Work it does
Local mark update	400,000	2.70 μs	a scalar; no tree write
Local account credit	200,000	24.2 μs	one depth-24 write
Cross-shard escrow	100,000	81.2 μs	two writes here, one there, plus the message

Table 5. Marginal wall-clock per operation, one shard pair.

The escrow decomposes cleanly, and the decomposition is the result. Of its 81.2 μs, the accounts side is 48.4 μs — exactly two local credits, the debit and the position write. The remaining 32.8 μs is the market side: one depth-24 sub-account write at 24.2 μs, plus 8.6 μs for everything else.

The boundary costs about eight hashes. That 8.6 μs residue — roughly eight Poseidon2 compressions — is the *entire* cost of the cross-shard mechanism: the fingerprint fold into the emitter's chain, the outbox append, the fetch, the sequence check, and the fold into the consumer's chain. It is 11% of an escrow's cost and about a third of one state write.

Put the other way: a cross-shard escrow costs about 3.4 local account writes, and *three* of those 3.4 are state writes the operation would have to perform on a single machine anyway. Sharding does not make the work more expensive; it makes almost exactly the same work happen on two machines instead of one. This is the quantitative form of the argument in Section 1: because the only thing crossing a boundary is value, and value is one message, the cut is nearly free.

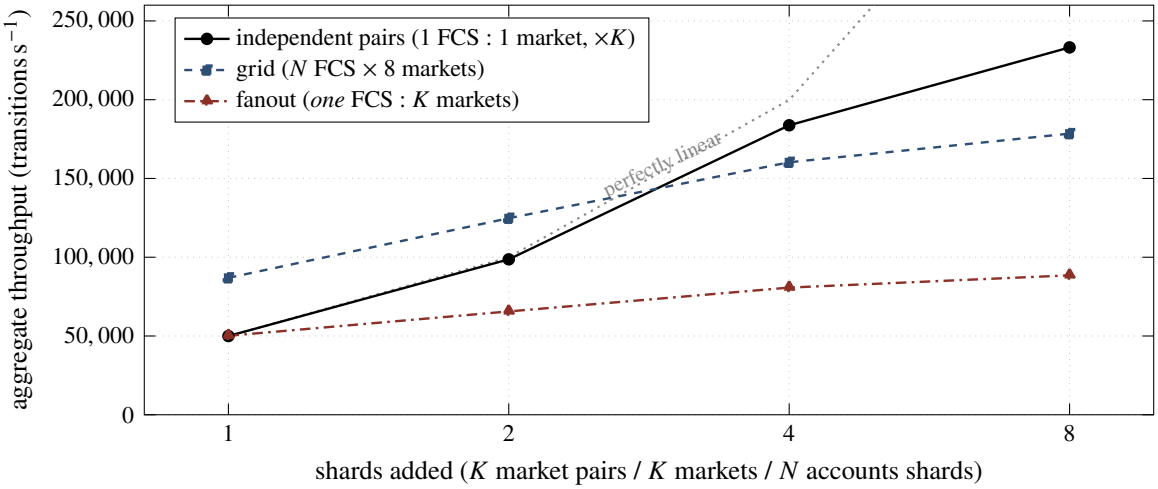


Figure 7. Execution scaling, three topologies, best of three runs on one ten-core machine. Independent pairs track the linear reference closely to $K = 4$ (93%) and fall to 59% at $K = 8$, where sixteen shard processes contend for ten cores — the ideal there is 62.5%, so this is core saturation and not a design limit. Fanout flattens hard: eight markets behind a *single* accounts shard buy only 1.77× the throughput of one, and at $K = 8$ that topology runs nine processes on ten cores, so nothing is masking it — the wall is the shared shard itself. The grid, which shards the accounts tree as well, recovers most of the loss.

And the workload sits on the cheap side. Holding the shards fixed at four pairs and varying the operation mix:

Cross-shard share	Transitions	Throughput	
33.3 %	120,000	143,667	escrow-only
25.0 %	160,000	180,702	
16.7 %	240,000	254,271	
8.3 %	480,000	456,808	
3.3 %	1,200,000	826,082	mostly local

Table 6. Four shard pairs, escrow load fixed, local operations swept.

Throughput rises 5.75× as the cross-shard fraction falls from a third to 3.3%. That is not a warning; it is a fit. A perpetuals workload is overwhelmingly local — matching, funding, mark updates and liquidation all happen inside one market shard — and only opens, closes and collateral moves cross a boundary. The design is cheapest exactly where the load actually is.

6.4 Execution scaling

Independent shards scale near-linearly until the cores run out. K independent accounts/market pairs, each with 20,000 accounts opening a position and 20,000 local mark updates:

Doubling the shard count doubles the throughput to $K = 4$. At $K = 8$ the machine runs out: sixteen shard processes on ten cores can at best achieve $10/16 = 62.5\%$ of linear, and the measurement

K	transitions	elapsed	throughput	efficiency	processes / cores
1	80,000	1.601 s	49,965	100 %	2 / 10
2	160,000	1.622 s	98,658	99 %	4 / 10
4	320,000	1.741 s	183,762	93 %	8 / 10
8	640,000	2.745 s	233,187	59 %	16 / 10

Table 7. Efficiency is $\text{tps}(K)/(K \cdot \text{tps}(1))$; 100% is perfect breadth scaling.

is 59%. The curve bends where the hardware does, not where the design does.

A shared accounts shard is a wall, and it is a real one. Now the adversarial topology: *one* accounts shard fanning out to K markets, every account opening on every market, so the single shard carries all K markets' escrow traffic.

K	transitions	elapsed	throughput	efficiency	processes / cores
1	16,000	0.319 s	50,137	100 %	2 / 10
2	28,000	0.427 s	65,598	65 %	3 / 10
4	52,000	0.644 s	80,729	40 %	5 / 10
8	100,000	1.130 s	88,507	22 %	9 / 10

Table 8. One accounts shard, K markets. Eight markets buy 1.77× the throughput of one.

Note the last column. At $K = 8$ this topology runs *nine* processes on ten cores, so unlike the pairs curve nothing is masking the result: the machine has capacity to spare and the throughput still flattens. The bottleneck is the single accounts node processing every market's opens on one thread, and no amount of concurrency elsewhere touches it.

This is the failure the design predicts, reproduced, and it is worth stating that it *survived* the harness fix that rescued the pairs curve. That is what makes it evidence rather than an artefact.

Sharding the accounts tree takes the wall down. The fix is to shard both trees. Holding the workload *constant* — 16,000 accounts, eight markets, 288,000 transitions at every point — and varying only how many accounts shards it is divided across:

N	transitions	elapsed	throughput	speed-up	
1	288,000	4.506 s	63,915	1.00×	≡ the 1:8 fanout point
2	288,000	2.666 s	108,010	1.69×	
4	288,000	1.775 s	162,273	2.54×	
8	288,000	1.530 s	188,201	2.94×	16 processes / 10 cores

Table 9. Identical work at every row. Only the number of accounts shards changes.

The same load finishes 2.94× faster across eight accounts shards. This is the measurement the

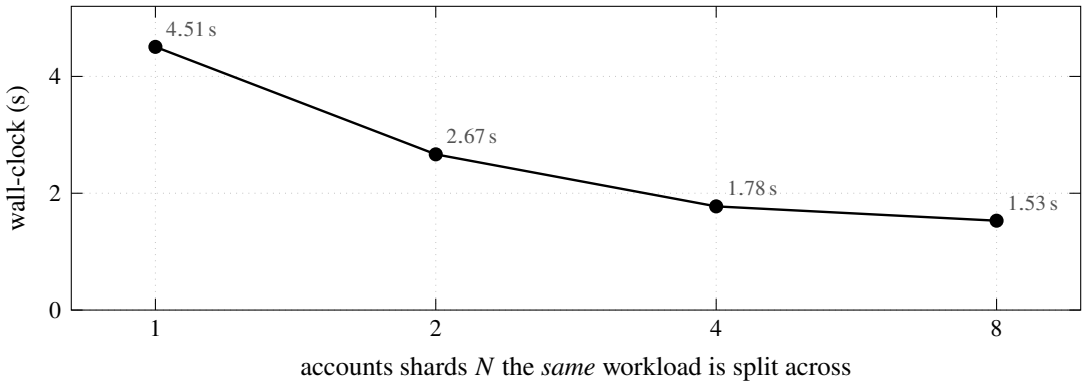


Figure 8. The measurement the design's central claim actually needs. The workload is held *constant* — 16,000 accounts each opening a position on each of eight markets, 288,000 transitions at every point — and only the number of accounts shards it is divided across changes. $N = 1$ is exactly the fanout point of Figure 7. Splitting the accounts tree eight ways cuts the wall-clock by 2.94×. The curve is sub-linear and honestly so: the eight markets remain shared and still absorb the entire aggregate order flow, and at $N = 8$ sixteen processes are again contending for ten cores.

design's central claim requires, because it is the only one that holds the workload fixed: the others measure added capacity, this one measures that a partition of the *same* work actually parallelises.

It is sub-linear, and the reasons are both known and named. Sixteen processes are again fighting for ten cores. And the eight markets are now shared across all N accounts shards, so each market consumes $N \times p$ escrows — a second-order shared bottleneck of exactly the kind the first curve exposed, one level up. Which is the honest summary of the whole execution story: **sharding by market scales cleanly; any shared shard is a wall; sharding that shard takes the wall down and reveals the next one.** A genuinely hot single market is where this sequence terminates, and that is the intra-market sharding this design does not attempt (Section 7).

6.5 What one encoding change cost

Two changes to the implementation landed between the measurements above and the paper: an order book on the market side, and a widening of the address encoding (Section 3.1). The second was subsequently reverted to a cheaper encoding *because* of the measurement below, which is the reason the measurement is reported at all. Separating their effects needs a controlled comparison, because the reference machine is a shared workstation and a busy one is indistinguishable from a regression if only one commit is ever run.

Both commits were therefore built and run *interleaved* — alternating arms, three repetitions each — on an otherwise idle sixteen-core Graviton4 machine, pinned to ten cores to match the reference topology. Interleaving means any drift in conditions falls on both arms equally, so the comparison survives a machine that is merely quiet rather than perfectly so.

Two things are visible. The *scaling* is untouched — the fixed-total grid folds 4.39× before and 4.46× after, pair efficiency is 100/100/100/96 % in both arms — so the order book, which is dead code on this path because these shards run bookless, costs nothing here. And absolute throughput falls by a uniform ≈ 2 % that shrinks to nothing as the workload becomes mostly local mark updates.

That gradient is the diagnosis, and it orders the sweeps correctly without being asked to. Fan-out

Sweep	Before	After	Δ	c_v
Fan-out, K=8	43,457	42,249	−2.8 %	0.10 %
Grid, N=1	31,384	30,487	−2.9 %	0.07 %
Pairs, K=1	23,219	22,690	−2.3 %	0.08 %
Pairs, K=8	177,808	173,867	−2.2 %	0.26 %
Mix, escrow-only	69,142	67,719	−2.1 %	0.05 %
Mix, mostly local	545,140	543,032	−0.4 %	0.51 %

Table 10. Throughput before and after, both commits interleaved on a dedicated machine. c_v is the coefficient of variation across repetitions; the effect is one to two orders of magnitude larger than the noise.

drives every market's escrows through a *single* accounts shard and so writes account leaves more densely than anything else measured; it shows the largest effect. The mostly-local mix writes them most sparsely; it shows almost none. Local marks do not write account leaves; escrows do. The account leaf is where the address widened from three limbs to five, taking the leaf from eight elements to ten — across the sponge rate, one permutation to two, a measured 1.97× on a hash that every account write performs. A cost that appears only on the account-writing path, in proportion to how much of the workload writes accounts, is that permutation and not something else.

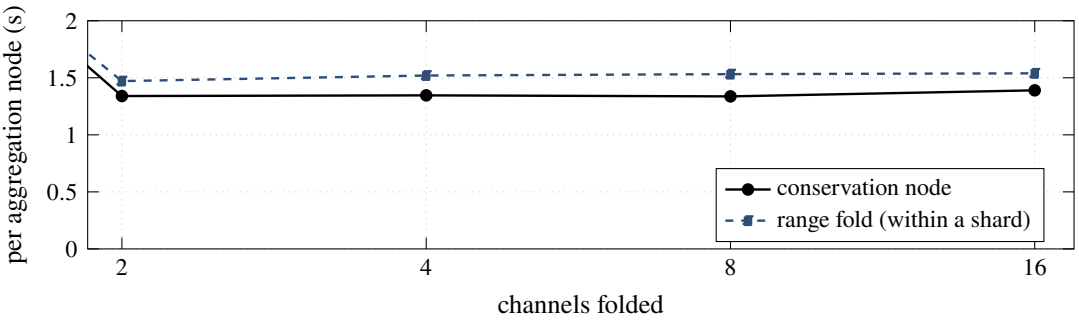
A trade with a magnitude, not a defect. The change that cost this closed a denial of service: a 64-bit limb can exceed the field modulus, H rejects a non-canonical element, and a withdrawal destination is chosen by the user — so a crafted address aborted the sequencer. Two percent is a fair price and the paper is not arguing otherwise.

What is worth recording is that the price was invisible when the change was made. The leaf sat at exactly eight elements, the last arity that fits one permutation, and nothing in the type system or the tests marks that as a cliff. The same number that Section 3.11 shows governs ambiguity also governs this, and neither is checked.

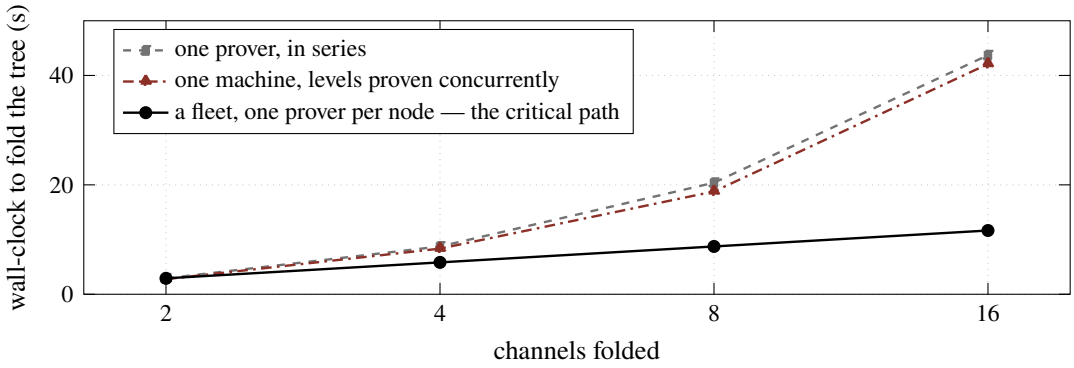
A cheaper encoding existed, and the implementation now uses it. Packing the address into seven-byte chunks yields three elements, each below 2^{56} and so canonical by construction — the same denial of service closed, in three elements rather than five, returning the account leaf to eight and the permutation with it. It was not free: it re-keys every commitment that hashes an address, which is inexpensive before a regenesis and expensive after, and it leaves the address with two encodings, since the exit leaf's five-limb form is read on chain and is an interface rather than a choice (Section 3.1). The repack was scoped to the account leaf for exactly that reason, and because the exit leaf gains nothing from it in any case — at nine elements it still spans two permutations.

So the measurement above is of a cost that has since been paid back. It is reported rather than quietly dropped because the interesting part was never the two percent. It is that a correct security fix, reviewed and tested, silently bought a permutation on the hottest hash in the system, and nothing in the toolchain said so. It took an interleaved A/B on a dedicated machine to see it at all, and the same boundary that governs whether a conditional encoding is unambiguous is the one that governs whether an added field is free.

One aside from the dedicated machine is worth recording, because it bears on how the figures in



(a) Cost per fold is flat. The k=1 points are omitted: they carry the one-time prover warm-up and are not a fold.



(b) Concurrency on one machine is *indistinguishable from* — in fact marginally worse than — proving in series. The log-depth speedup exists only across machines.

Figure 9. The two proving curves. Above: every aggregation node costs the same regardless of how wide the tree is, so total proving work is linear in the channel count — one node per fold. Below: what that buys depends entirely on where the nodes run. Sixteen channels is a depth-four tree of fifteen node proofs; in series that is $15 \times 2.787 = 41.8$ s, and on a fleet it is $4 \times 2.787 = 11.1$ s, a 3.75× reduction that a single machine cannot realise because one node proof already saturates its cores.

Section 6.4 should be read. Pinned to ten cores, the fixed-total grid folds 4.4× there against 2.9× on the reference machine, while absolute throughput is roughly half. The reference machine's ten cores are four performance and six efficiency cores; the other's ten are identical. The most plausible reading is that a heterogeneous machine flatters the single-shard baseline and penalises the split, since additional shard processes land on slower cores as the count grows — which would mean the reference figures *understate* how well the partition scales on uniform hardware. Stated as an observation rather than a result: it is one machine pair, and per-core placement was not instrumented.

6.6 Proving

Execution scaling and proving scaling are different questions with different answers, and conflating them is the easiest way to overstate this architecture.

The recursion behaves. Every aggregation node costs the same regardless of the tree's size — a bounded, constant-time fold:

channels	tree fold	per node	root check	
1	3.657 s	—	0.321 s	one-time warm-up
2	1.340 s	1.340 s	0.299 s	
4	4.037 s	1.346 s	0.361 s	
8	9.358 s	1.337 s	0.317 s	
16	20.843 s	1.390 s	0.349 s	

Table 11. Conservation-tree aggregation, leaf circuit 2^{13}, node circuit 2^{15}. One channel leaf proves in 305 ms.

Per-node cost moves by 4% across a sixteen-fold increase in tree width, so total proving work is linear in the channel count — one node proof per fold. The within-shard range recursion has the same shape: a chained credit block proves in 11.6 ms and each range fold costs a flat 1.47–1.54 s.

These are the extension-field circuits of Section 4.9. Against the base-field version they superseded, a conservation node costs about 9% more — 1.35 s against 1.24 s — and it is worth being precise about where that goes, because the shape of the overhead matters more than its size. A node's interface doubled from six field elements to twelve, so it now folds four two-limb products through extension multiplication — roughly three base multiplications each, by Karatsuba — checks the challenges limb by limb, and commits to a public-input vector twice as long.

Every one of those is a *fixed* cost per node. None of them grows with the tree, which is why the flatness in Table 11 survives the field change untouched: 4% drift across a sixteen-fold increase in width, exactly as before. The extension field shifted the constant and left the asymptotics alone, which is the only thing the log-depth argument of the next section needs.

The log depth is a fleet property, not a machine property. This is the finding that most changes how the architecture should be deployed, and it is sharpest at sixteen channels. Sixteen channels is a depth-four balanced tree of fifteen internal node proofs, at 2.913 s each warm:

channels	levels	one prover, in series	one machine, concurrent	a fleet	
2	1	2.913 s	2.761 s	2.913 s	
4	2	8.740 s	8.343 s	5.827 s	
8	3	20.394 s	18.791 s	8.740 s	
16	4	43.702 s	42.085 s	11.654 s	3.75×

Table 12. Folding the conservation tree. "One machine, concurrent" proves each level's independent nodes on concurrent threads; "a fleet" is the critical path when each node has its own prover.

Read the last row carefully. Proving a level's independent nodes concurrently on one machine takes 42.1 s against 43.7 s for proving all fifteen in series — a 4% difference, which is to say none. The parallelism is real and it buys nothing, because each individual proof already saturates the cores; threads, and separate processes too, only contend for them. (An earlier measurement on the base-field circuits had the concurrent figure marginally *worse* than serial. The honest reading of both is that they are indistinguishable.)

Give each node its own prover and a level costs one node, so the critical path collapses to the tree depth: $4 \times 2.913 = 11.7$ s, a 3.75× reduction. The arithmetic is exactly nodes/depth = 15/4, and extrapolates: a thousand channels is tens of minutes of proving in series and about ten node-times on a fleet ten levels deep.

Why this design leans on permissionless provers. The tree does not make one machine faster. It makes the work *distributable* — and the logarithmic latency exists only when the levels land on different machines. A proof serialises to bytes carrying only its verifier data, so the workers share nothing but the proofs they hand one another, which is exactly what lets the fleet be open in the first place. Permissionless proving is not a governance preference bolted onto this architecture; it is where the design's headline latency property physically lives.

End to end, and the on-chain tail. Proving one settlement for a single accounts/market pair at tree depth 8, by phase: circuit build ≈ 18 s (one-time), two range proofs ≈ 7 s, conservation tree ≈ 6 s, state root ≈ 1 s, settlement ≈ 1 s. For a two-market grid, the ranges cost ≈ 7 s and the multichannel join ≈ 2.5 s.

The tail that reaches the chain is a different order of magnitude and must not be confused with the numbers above. The settlement proof is re-proven under the BN254-hashed configuration and then compiled to a gnark PLONK circuit: **5,214,935 constraints** with twelve public inputs, an ≈ 11 min 20 s trusted-setup ceremony and 1 min 18 s to prove. The result is an 864-byte proof a pairing check verifies on chain.

Note what the extension field did to that number: it added *144* constraints out of 5.2 million, a change of 0.003%. The ceremony and proving times differ from the base-field circuit's by more than that, and those differences are measurement noise rather than an effect of the field — the honest statement is that moving the grand product to $\mathbb{F}_{p^2}$ cost the on-chain tail nothing measurable.

Do not read the small wrap as the settlement cost. A convenient 2^{12} test wrap in the benchmark suite builds in 10.3 s and proves in 21.8 s. That is *not* the settlement wrap; the two differ by roughly three orders of magnitude in constraint count. The anchoring cost is the 1 min 18 s figure, once per epoch, amortised across every transition the epoch contains.

6.7 Settling on chain

The settlement contract is deployed on a STRATO testnet and the path from a committed root to a settled one runs end to end: the prover daemon reads the contract's current settled root, proves the epoch bound to it, wraps the proof through the BN254 tail, and submits `anchorEpoch` itself, with no human in the loop.

The distinction between the two rows matters more than the epoch counts. The retired anchor demonstrated that a chain of settled roots can be maintained and that a stale proof is rejected. What it could not demonstrate — and what the current deployment's twelfth and eleventh public inputs and its continuity constraint now make provable — is that each epoch *began* where the last one ended

Current deployment — twelve public inputs, $\mathbb{F}_{p^2}$ challenges	
`SettlementAnchor`	`3dfd7c0b92b635626afaa20791677ca5b025fc8a`
`PlonkVerifier`	`78286e6f9f2c25021b17043e2f1f221e0b55d366`
`genesisRoot`	`b8040f90fbc646aaf90bce1dbf3671c69f7f184e71662043bca1f7e380d6cb6d`
epoch 0	`6e04ab72 ... 30e2`
Superseded deployment — ten public inputs, base-field challenges	
`SettlementAnchor`	`665970f0528b36193f9ea7734c512aeab1b98e75`
`PlonkVerifier`	`dd30aeae24c8818c836e99b5682556bc9e6f42bc`
epoch 0	`ba0c989ade3f80c9cf4437b9551fe3bb2d3062e6f485ec840fceb235d257266b`
epoch 1	`610b0ae4bf6f076cc2585d663e4d2732daab9ac00dd65b3ae0c1a6e0ca30823f`
epoch 2	`2b842d616dc9caf1fc199bfbd6df001e225c77b365fca3ee45b5f69ec7bc3212`

Table 13. Two deployments, and they establish different things. The superseded anchor carries three chained epochs and is the evidence for root-chaining and replay-protection — each epoch's `prior` is the previous `globalRoot`, and a replay against a stale prior reverts. It could not prove *continuation*, because the circuit that would bind an epoch's starting state to its predecessor did not yet exist. The current anchor can, and carries one epoch so far. Both remain readable on chain.

(Section 4.10). A reader comparing the two is looking at the same pipeline before and after that constraint existed.

The state is readable without authentication, so both chains are checkable rather than asserted:

```
curl -s https://node4.testnet.strato.nexus/bloc/v2.2/contracts/\
SettlementAnchor/3dfd7c0b92b635626afaa20791677ca5b025fc8a/state
```

Two caveats belong beside it. This is a *testnet* node: it can be reset or retired, so the roots are printed above rather than only linked, and the artefact stays self-contained if the endpoint does not. And the contract returns each root as a `uint256` decimal — the genesis root above, for instance, comes back as

```
83232738962522602169431160266424197506504803418358433066361250784398009617261
```

which is the same value as the 64-hex digest, in the form `gl.Digest` and the circuits use.

Section 7 states what this does and does not establish: the verification key depends on the topology, so the deployed verifier matches one fixed shard configuration, and a production deployment wants its own trusted setup rather than the borrowed reference string this used.

6.8 Liveness under an adversarial transport

Throughput is not the only claim. The two-shard slice runs as separate processes over a transport that is deliberately hostile: it drops, duplicates, reorders and delays messages, and a node can be killed mid-transfer and recovered from its journal alone (`scaling/node/chaos_test.go`, `scaling/recover.go`). Conservation and exactly-once hold throughout.

That result is what licenses the weak transport guarantee of Section 2.2. A duplicate falls behind the consumer's cursor and is a no-op; a gap waits; a killed node resumes pulling from the cursor its journal committed, so it neither re-requests nor skips. None of it depends on a cooperative in-process bus, which is precisely the thing a single-process simulation cannot demonstrate.

7 Limits

An architecture paper is only as useful as its account of what it does not do. This section is that account. It separates three things that are easy to conflate: what the design *cannot* do, what is *built and proven* but not yet wired into a running path, and what is *specified in Go* but has no circuit behind it.

7.1 What the design does not scale

Depth does not scale — a hot market is still one sequencer. Breadth scales: N markets run on $\sim N$ sequencers, and aggregate throughput grows with the markets and the hardware behind them. Depth does not. A single hot market is still one sequencer, because an order book is an inherently sequential structure — price-time priority is a total order, and a total order is what resists partition. Sharding by market does not make the BTC book itself faster.

Intra-market sharding is a separate and later design, and it is *design-blocked* rather than merely unbuilt, on one question a partitioned book forces: how price-time priority survives a match that crosses shards. The measurements in Section 6 name this as the next wall precisely because the grid curve bends where the markets become the shared resource.

Latency is hidden, not removed. A cross-shard order takes more than one block to fully settle. The sequencer confirms the fill to the trader optimistically — the moment it matches — while the cross-shard settlement finalises underneath. The experience stays fast; the finality path is genuinely longer, and the withdrawal window is the price the whole design pays for shards that never wait on one another.

Uniform account identifiers balance storage, not load. Hash-derived account identifiers with each shard owning a contiguous range distribute *storage* evenly. They do not distribute *load*: one hyperactive account still lands wholly on one shard, and no partition of a flat key space fixes that.

Data availability is a later constraint, and a real one. Far enough out, the published data outgrows what STRATO and Ethereum blobs [22] can carry. The endgame is a dedicated availability layer with sampling over erasure-coded data [21]. It is named for completeness, not as a dependency: execution sharding rides existing data availability for a long time before availability binds, and the two efforts are independent.

7.2 Proven, but not on a running path

The elastic split is a circuit, not an operation. The split circuit, the aggregator's leaf reshaping and the shard's own partition all exist, are tested, and have negative tests showing that freshness and drain are load-bearing. But nothing in the running pipeline — not the shard nodes, not the aggregator, not the prover daemon — performs a live split. The mechanism is proven; the orchestration that would decide *when* to split, and the operations that would execute it, are not written. The same applies to merge.

The split circuit also proves a *single* market peer. A multi-peer split folds one such market leaf per market, which is a mechanical generalisation but an unbuilt one.

Provable runs must be circuit-shaped. The job builders require one funding credit followed by one open per accounts block, and one escrow-in per market block, with no mark updates. An arbitrary shard-node workload — the multi-record blocks the throughput benchmarks actually produce — is not provable as it stands. So the execution numbers in Section 6.4 and the proving numbers in Section 6.6 are measurements of the same system but not of the same runs, and no claim in this paper depends on their being so.

A verifier per topology. The settlement wrapper's verification key depends on the circuit's *structure* — notably the number of ranges folded — so a different topology needs its own deployed verifier contract. The live automatic anchoring works only for the shape its deployed key matches, and the demonstration used a fixed single-pair configuration. A production cluster with a changing shard count needs either a universal circuit shape or a verifier-registry, and neither is built.

7.3 Specified in Go, with no circuit behind it

Two circuit families, and the boundary between them. This is the most important distinction in the section. The scaling shard circuits model a *simplified* market: an escrow arrives, margin is custodied into a sub-account with entry set to the current mark, and that is all. There is **no circuit for a mark update, and none for a liquidation**. Provable runs fix the mark at genesis.

The Go specification is considerably ahead of that. `MarketShard` implements mark-to-market, equity, maintenance margin, and liquidation — and liquidation is not merely judged but *executed*: a position below maintenance is force-closed through the same settle path a voluntary close uses, a bankrupt close settles at zero with the shortfall absorbed by the pool, and the whole cycle is wired through the record and operation types and exercised by tests (`market.go:325,348`, `recover.go:238`).

So the precise statement is: **hybrid margin — the claim that health and liquidation become shard-local — is demonstrated and tested in the executable specification; no circuit yet proves that a mark update or a liquidation was legitimate.** Closing that is what would make the hybrid-margin argument complete rather than merely convincing.

The same gap has a second, sharper edge. Because there is no settle-side circuit at all, nothing binds a settlement's *destination* to the escrow that caused it (Section 3.3). Value is conserved whoever receives it, so a misrouted settlement produces a perfectly valid epoch. Value cannot be created; it can be misdelivered, and only the first of those is currently proven impossible.

The full perpetuals dynamics — funding, premium, assignment, deleveraging — live in the exchange's existing v3 circuit family, which is a separate and older stack with its own settlement contract. Numbers from the two must not be mixed, and this paper quotes none from the v3 family except where explicitly labelled.

Funding accrual is absent from both. To be exact about the previous point: perpetual funding-rate accrual appears in neither the scaling specification nor the scaling circuits. Where this paper says health becomes local, it means mark-to-market and liquidation. Funding is a market-local computation and there is no reason to expect it to be otherwise, but it is not implemented, and the claim is not made.

Auto-deleveraging is flagged, not implemented. A settlement that would overdraw the pool is refused outright rather than silently netted. That condition is an insurance shortfall requiring

auto-deleveraging, and the refusal is a deliberate placeholder for a mechanism that does not exist.

Multi-market data availability is ambiguous. The `RecOpen` record carries no market field, so reconstructing a grid run maps a block's K opens to markets by ascending order, relying on a convention rather than on the data. A general multi-market published format needs a market field on that record — which changes the data-availability hash, every shard circuit, and the conformance vectors. It is invasive, it is known, and it is not done.

7.4 What the security rests on

Two of the three items an earlier draft of this paper listed here have since been closed, and it is worth being explicit about which, because the remaining one is of a different kind.

1. **The challenge field — closed.** The grand product ran over a 2^{64} field with a grindable transcript. It now runs over $\mathbb{F}_{p^2}$, at a cost of 144 constraints in the settlement circuit and about 9% of a conservation node's proving time (Section 4.9).
2. **The multi-epoch binding — closed.** The settlement now pins the prior settled root to the global root over the ranges' starting roots, so each epoch provably continues from the last. The frontier's carry follows from that plus the cumulative-outbox structure, with no separate mechanism (Section 4.10).
3. **Settle-side proving — open.** Binding a settlement's destination to its originating escrow, and proving that mark updates and liquidations were legitimate, are unbuilt. This is the remaining item, and it is an unbuilt *extension* of what the paper models rather than a gap in the argument it makes.

The first two were open items against this paper's own soundness argument; both were found while writing it and closed before publication, and both analyses are kept in Section 4 because the reasoning is what makes the current design the right one rather than an arbitrary one.

The challenge field, and what it now rests on. The grand product runs over $\mathbb{F}_{p^2}$, giving $6N^2/p^2$ — 2^{-85} at 2^{20} messages per epoch, beyond any grinding budget. What that rests on is worth stating: Fiat–Shamir in the random-oracle model for the challenge derivation, and Schwartz–Zippel over the extension for the product identity. A uniform $\geq$ 100-bit claim across arbitrarily large epochs rather than realistic ones would want $\mathbb{F}_{p^3}$; nothing in the current design forecloses that, and nothing currently requires it.

The wrapping chain, and a borrowed setup. On chain, only the BN254 pairing check runs. The settlement's Goldilocks soundness is carried by the wrapper and the gnark PLONK compilation beneath it — the same trust base as the exchange's existing range path, but a longer chain than a reader might assume from "the contract verifies the proof".

Separately, the wrap used an existing structured reference string rather than one generated for this circuit. A production deployment wants its own ceremony.

Sequencing is journal-attested, not chain-enforced. The settlement proof reproduces the committed root, so a wrong root cannot be anchored. But *which* transactions a shard sequenced is attested by its journal and the on-chain data hash, not by an on-chain forced-inclusion queue. The exchange's existing settlement contract has that queue; the scaling anchor only settles. The forced-inclusion

path that Section 2 leans on for liveness — the thing that turns "eventually" into a guarantee — is architecture, not deployed code, in the sharded configuration.

Domain separation is structural. As Section 3.11 sets out, the leaf and node hashes are not domain-tagged; their unambiguity comes from fixed tree depths and circuits that fix each hash's shape. This is sound as written and fragile to future change.

7.5 Open questions

Several questions are genuinely open rather than merely unimplemented, and the design does not yet have answers:

Rebalance semantics. What does an explicit cross-margin rebalance look like to a trader, and how slow is acceptable? Hybrid margin makes this a product question, not just an engineering one.

Refund and timeout parameters. How long before a stranded credit escalates to the forced-inclusion path, and who pays for it?

Shard assignment and resharding policy. The split mechanism exists; the policy that decides when to invoke it does not.

Ordering across markets. Oracle and mark-price updates that gate liquidation have no single global instant across shards. How consistent do they need to be? This is the one open question that could reach back into the conservation argument, because it concerns state that liquidation reads.

Admission over many positions. With one position per account, admissibility is a function of the message alone. Once an account may hold several, "already open here" becomes part of admission — best absorbed by making *accept* total over that state, so *reject* stays exactly "the message is inadmissible", keeping the complementarity of Section 2.2 intact.

7.6 On the measurements themselves

Every number in this paper comes from one ten-core machine. The execution figures are the best of three runs and reproduce to within a few percent; the proving figures are single runs of an opt-in benchmark, and the critical-path row is corroborated by arithmetic rather than by repetition — fifteen node proofs at 2.787 s is 41.8 s in series and 11.1 s over four levels, which is what was measured.

Two structural caveats. The fleet figures are a *model* of a fleet — computed as depth times a measured node cost — not a measurement of one; the multi-process fold demonstrates that proofs pass between processes as bytes, but running the levels on genuinely separate machines has not been done. And the throughput ceiling reported here is a ten-core ceiling: at $K = 8$ the pairs and grid topologies are core-saturated, so those points measure the machine, and whether the curves stay near-linear at larger K on larger hardware is an untested extrapolation.

8 Conclusion

The claim this paper set out to defend was that an exchange's state can be partitioned into subtrees that advance independently on separate machines, and that the result can be *proven* to be the state a single machine would have produced — conservation of value included — at linear proving cost and

logarithmic proving latency.

What the implementation and its measurements establish:

The partition is real, and it is nearly free. A cross-shard escrow costs 81 μs against a local account write's 24 μs, and only 8.6 μs of that — about eight hash compressions, 11% — is the boundary mechanism itself. The rest is state-write work that a single machine would have to do anyway. The execution layer is bound by hashing its own state tree, at 1.08 μs per compression, which is a cost of being a rollup rather than a cost of being sharded.

Independent shards scale until the hardware stops them. Throughput is 93% of linear at four shard pairs and 59% at eight, where sixteen processes contend for ten cores and the ideal is 62.5%.

Any shared shard is a wall, and sharding it takes the wall down. Eight markets behind one accounts shard buy 1.77× the throughput of one, on a machine with cores to spare — a design limit, not a measurement artefact. Splitting the accounts tree eight ways runs the *identical* workload 2.94× faster. Both curves survived the discovery that the original harness was measuring its own submitter.

Conservation composes. The balanced multiset identity **sent** $\uplus$ **inFlightIn** $\equiv$ **received** $\uplus$ **inFlightOut**, discharged as a grand product, holds continuously rather than only at drain points, is order-independent so it folds in any order, and is multiplicative so one non-conserving channel anywhere leaves the root unprovable. Three bindings make it mean what it says, and all three are implemented.

The recursion is linear and its latency is logarithmic — on a fleet. Every aggregation node costs a flat 1.24 s regardless of tree width. But proving a level's nodes concurrently on one machine is marginally *worse* than proving them in series, because a single node proof already saturates the cores. Sixteen channels is 41.8 s in series and 11.1 s across four levels of a fleet. The log depth is a property of distribution, not of parallelism, which is why permissionless proving is where this architecture's headline latency actually lives.

It settles. One proof carries the committed global root, the conservation of the same epoch's messages, the binding that forces them to be the same epoch, and the binding that forces the epoch to continue from its predecessor — twelve public inputs, an 864-byte proof after the BN254 tail, verified by a contract that chains settled roots. It is anchored on a STRATO testnet, with no human in the loop.

What the argument cost, and what is still not established

Two of the claims above were not true of the first version of these circuits, and saying so is part of the result rather than an aside.

The challenge field. The grand product was instantiated over the Goldilocks base field, where its soundness degrades quadratically in the epoch's message count and a laptop-scale offline grind becomes available above roughly 5×10^4 messages per epoch — inside this design's own throughput envelope, reached by a single second of settlement at the rates measured here. Moving it to $\mathbb{F}_{p^2}$ cost 144 constraints out of 5.2 million and about 9% of a conservation node's proving time.

The multi-epoch binding. Conservation was proven one epoch at a time. An epoch's starting corners were established by nothing, and no circuit asserted that this epoch's state began where the last one ended — so even the anchored chain rested on the prover having honestly continued. One

constraint fixed it, and the frontier's carry then followed from machinery the design already had: because a shard's outbox is cumulative and seeded from the frontier committed at its starting root, state continuity forces the prior corner to the real prior frontier by the same collision resistance that makes the first binding sound. The corner-pinning mechanism that seemed to be required turned out to be already implied.

What is still open. There is no settle-side circuit. Value is conserved whoever receives a settlement, but nothing binds a settlement's destination to the escrow that caused it, and no proof attests that a mark update or a liquidation was legitimate. Value cannot be created; it is not yet proven that it cannot be misdelivered. Hybrid margin — the design decision the whole architecture turns on — is implemented, tested and shard-local in the executable specification, and unproven in circuit. That is the honest boundary of this paper.

The shape of the result

The reason any of this works is worth restating, because it is not a general-purpose result and should not be read as one. General state sharding fails because arbitrary contracts make every boundary artificial and every boundary crossing unbounded. An exchange escapes that not through better engineering but because its state was already partitioned: markets are mutually independent, accounts partition by index, and the only thing that ever crosses a boundary is value, which is one message with thirteen fields.

Everything in this paper follows from that. The conservation argument is tractable because there is exactly one thing to conserve. The proof tree has a shape because the state tree already had one. The cut is cheap because the only traffic across it is the traffic the workload was always going to generate. What a zero-knowledge rollup adds is the last piece: because correctness is proven rather than trusted, the machines on either side of a boundary need not trust each other, and the partition costs no consensus.

Sharding by market scales cleanly. Any shared shard is a wall, and sharding that shard takes the wall down and reveals the next one. A genuinely hot single market is where the sequence ends, and that — the sharded order book, and how price-time priority survives a match that crosses shards — is the question this design leaves open and correctly leaves last.

References

[1] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. https://bitcoin.org/bitcoin.pdf

[2] V. Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. 2014. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

[3] G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum Yellow Paper, 2014–. https://ethereum.github.io/yellowpaper/paper.pdf

[4] A. Yakovenko. Solana: A New Architecture for a High Performance Blockchain, v0.8.13. 2018. https://solana.com/solana-whitepaper.pdf

[5] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO '87*, LNCS 293, pages 369–378. Springer, 1988.

[6] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO '86*, LNCS 263, pages 186–194. Springer, 1987.

[7] J. T. Schwartz. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *Journal of the ACM*, 27(4):701–717, 1980.

[8] R. Zippel. Probabilistic Algorithms for Sparse Polynomials. In *EUROSAM '79*, LNCS 72, pages 216–226. Springer, 1979.

[9] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge. IACR Cryptology ePrint Archive, Report 2019/953. https://eprint.iacr.org/2019/953

[10] A. Gabizon and Z. J. Williamson. plookup: A Simplified Polynomial Protocol for Lookup Tables. IACR Cryptology ePrint Archive, Report 2020/315. https://eprint.iacr.org/2020/315

[11] J. Groth. On the Size of Pairing-Based Non-Interactive Arguments. In *EUROCRYPT 2016*, LNCS 9666, pages 305–326. Springer, 2016.

[12] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT 2010*, LNCS 6477, pages 177–194. Springer, 2010.

[13] P. S. L. M. Barreto and M. Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography 2005*, LNCS 3897, pages 319–331. Springer, 2006.

[14] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, Transparent, and Post-Quantum Secure Computational Integrity. IACR Cryptology ePrint Archive, Report 2018/046. https://eprint.iacr.org/2018/046

[15] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Fast Reed–Solomon Interactive Oracle Proofs of Proximity. In *ICALP 2018*, LIPIcs 107, pages 14:1–14:17, 2018.

[16] Polygon Zero. Plonky2: Fast Recursive Arguments with PLONK and FRI. 2022. https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf

[17] S. Bowe, J. Grigg, and D. Hopwood. Recursive Proof Composition without a Trusted Setup. IACR Cryptology ePrint Archive, Report 2019/1021. https://eprint.iacr.org/2019/1021

[18] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *USENIX Security 2021*, pages 519–535, 2021.

[19] L. Grassi, D. Khovratovich, and M. Schofnegger. Poseidon2: A Faster Version of the Poseidon Hash Function. IACR Cryptology ePrint Archive, Report 2023/323. https://eprint.iacr.org/2023/323

[20] ConsenSys. gnark: A Fast Zero-Knowledge Proof Library. https://github.com/Consensys/gnark

[21] M. Al-Bassam, A. Sonnino, and V. Buterin. Fraud and Data Availability Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities. arXiv:1809.09044, 2018. https://arxiv.org/abs/1809.09044

[22] V. Buterin, D. Feist, D. Loerakker, G. Kadianakis, M. Garnett, and M. Taiwo. EIP-4844: Shard Blob Transactions. Ethereum Improvement Proposals, 2022. https://eips.ethereum.org/EIPS/eip-4844

[23] StarkWare. StarkEx: A Scalability Engine for Trading and Payments. https://starkware.co/starkex/

[24] Hyperliquid. Hyperliquid Documentation. https://hyperliquid.gitbook.io/hyperliquid-docs

[25] Lighter. Lighter Documentation. https://docs.lighter.xyz

[26] dYdX. dYdX Chain Documentation. https://docs.dydx.xyz

A Commitment encodings

Every commitment in the system, with its exact element list. H is Poseidon2 over Goldilocks — plonky2's `hash_n_to_hash_no_pad`, rate 8, no padding; an owner is three 56-bit limbs in the account leaf and five 32-bit limbs where a contract reads it, an amount four 32-bit limbs (Section 3.1); the empty digest is the literal zero. Three encodings extend conditionally on their contents and so have more than one arity; each is marked, and Section 3.11 gives the argument that they stay unambiguous.

Commitment	Arity	Element list
Merkle node `gl.go:133`	8	$l[0..3]$, $r[0..3]$
Account leaf `accounts.go:21`	8	$owner[0..2]$, $free[0..3]$, $nonce$ owner packed 3×56 (`U160Packed`) — eight elements is one permutation
Exit leaf `accounts.go:47`	11	$account$, $amount[0..3]$, $dest[0..4]$, seq (one L2→L1 withdrawal; seq is also the leaf index)
Position `market.go:21`	3	$side$, $size$, $entry$
Sub-account `market.go:42`	8	$margin[0..3]$, $pos[0..3]$ (`openSeq`, `owner`, `account` are *not* committed)
Channel leaf `messages.go:171`	18	$nextSeq$, $wantSeq$, out$[0..3]$, in$[0..3]$, emittedValue$[0..3]$, consumedValue$[0..3]$
Channel root step `messages.go:187`	9	$acc[0..3]$, $peer$, $L_{peer}[0..3]$ (from $\mathbf{0}$, peers in ascending order)
Accounts shard root `accounts.go:145`	12	$treeRoot[0..3]$, $channelRoot[0..3]$, $exitRoot[0..3]$
Market shard root `market.go:219`	13 *or* 17	$subRoot[0..3]$, $markPrice$, $pool[0..3]$, $channelRoot[0..3]$ then $bookRoot[0..3]$ on a book-enabled shard; a bookless shard folds nothing extra and its root is byte-identical to before the book existed
Message fold `messages.go:84`	17 *or* 19	$prev[0..3]$, $\vec{v}(m)$ (see below) then $price$, $expiry$ when $price \neq 0$ — so the chain binds a limit escrow's terms even though $\vec{v}(m)$ does not
Message field vector $\vec{v}(m)$ `multiset.go:88`	13	Src, Dst, Seq, $Kind$, $Account$, $Market$, $Amount[0..3]$, $CauseRef$, $Side$, $Size$ fixed at thirteen; $price$ and $expiry$ are deliberately *not* fingerprinted — see Section 4.8

Commitment	Arity	Element list
Record encoding `block.go:55`	19, 32 *or* 34	*Kind*, *Account*, *Side*, *Size*, *Price*, *Owner*[0..4], *Amount*[0..3], *Margin*[0..3], *flag* then $\vec{v}(m)$ when *flag* = 1, then *price*, *expiry* when that message is priced
DA hash step `block.go:91`	23, 36 *or* 38	$h[0..3] \,\|\|\, \text{encode}(r)$ (from **0**)
Block header `block.go:103`	23	*Shard*, *Number*, *PrevRoot*[0..3], *NewRoot*[0..3], *Timestamp*, *OutboxCommit*[0..3], *InboxCommit*[0..3], *DAHash*[0..3]
Epoch commitment step `challenge.go:37`	19	*K*[0..3], *Shard*, *First*, *Last*, *RootBefore*[0..3], *RootAfter*[0..3], *DAChain*[0..3] seeded at H(EPOCHK), ranges in shard order
Challenge transcript `challenge.go:59`	9	CHALBG, *priorSettledRoot*[0..3], *K*[0..3] $\beta = (t_0, t_1)$, $\gamma = (t_2, t_3)$, both in $\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2 - 7)$; see Section 4.9

Domain separators. Only two commitments carry a tag: EPOCHK = `0x45504F43484B` and CHALBG = `0x4348414C4247`, the ASCII strings packed big-endian as field elements. Every other commitment above relies on structural position for its unambiguity, and on arity that is either fixed or — for the three conditional encodings — always above the sponge rate (Section 3.11).

B Circuit public inputs

The public-input layout of each circuit in the stack, in slot order. These are the interfaces the recursion binds against: a parent connects specific slots of its child's public inputs, so the layouts are load-bearing rather than descriptive.

Circuit	Len	Slots
Shard block `shardblock.rs:44`	22	0 rootBefore[4] · 4 rootAfter[4] · 8 number · 9 ts · 10 da[4] · 14 outbox[4] · 18 inbox[4]
Multi-peer consume block `shardblock.rs`	var	… 14 inbox per peer, at $14 + 4j$ for peer index j
Multi-peer emit block `shardblock.rs`	var	… 14 outbox per peer, at $14 + 4j$
Shard range `shardrange.rs:31`	22	0 rootBefore[4] · 4 rootAfter[4] · 8 first · 9 last · 10 daChain[4] · 14 outbox[4] · 18 inbox[4]

Circuit	Len	Slots
Message product `shardproduct.rs:29`	14	0 β[2] · 2 γ[2] · 4 product[2] · 6 chainBefore[4] · 10 chainAfter[4]
Shard epoch `shardepoch.rs:29`	29	0 priorOutbox[4] · 4 product[2] · 6 β[2] · 8 γ[2] · 10 newOutbox[4] · 14 shard · 15 first · 16 last · 17 rootBefore[4] · 21 rootAfter[4] · 25 daChain[4]
Conservation leaf / node / root `conservationtree.rs:44`	12	0 β[2] · 2 γ[2] · 4 P(**sent**)[2] · 6 P(**received**)[2] · 8 P(**inFlightIn**)[2] · 10 P(**inFlightOut**)[2]
Range join `rangejoin.rs`	8	0 globalBefore[4] · 4 globalAfter[4]
Multiset join `multisetjoin.rs`	18	0 β[2] · 2 γ[2] · 4 priorSettled[4] · 8 newOutbox[4] · 12 newInbox[4] · 16 P(**inFlightOut**)[2]
Multichannel join `multichanneljoin.rs:39`	8	0 β[2] · 2 γ[2] · 4 priorSettled[4]
State root `stateroot.rs:27`	12	0 globalRoot[4] · 4 globalRootBefore[4] · 8 epochCommit[4]
Split `split.rs:37`	16	0 oldRoot[4] · 4 leftRoot[4] · 8 rightRoot[4] · 12 marketRoot[4]
Grid settlement `grid.rs`	4	0 globalRoot[4]
Settlement `settlement.rs:32`	**12**	0 globalRoot[4] · 4 priorSettled[4] · 8 β[2] · 10 γ[2]

Every slot marked [2] is an element of the quadratic extension $\mathbb{F}_{p^2}$ — see Section 4.9. That is why the challenge- and product-carrying layouts are wider than a reader of an earlier draft of this paper would expect: `MPPI` grew from 11 to 14, `SEPI` from 26 to 29, `CAPI` from 6 to 12, `MJPI` from 15 to 18, `MCJPI` from 6 to 8 and `SETPI` from 10 to 12. The purely hash-carrying layouts — shard block, shard range, range join, split, grid — are unchanged, because nothing in them touches the grand product.

`STPI` grew for a different reason: it gained `globalRootBefore` at slot 4, the global root folded over the ranges' *starting* roots, which is what the settlement pins the prior settled root against to prove state continuity across epochs (Section 4.10).

The settlement layout is the anchor's whole interface: `SettlementAnchor.anchorEpoch` takes exactly these twelve values plus the BN254 proof, and advances the settled root only when `priorSettled` matches its current one. `build_join_settlement` — the grid path — exposes the identical layout, so drivers and the contract read both the same way.

C Reproduction

Every measurement in this paper was taken on an Apple M4 (10 cores, 24 GB), on the Lambdachain rollup repository at commit `ea2f36d`. These are the commands.

C.1 Execution scaling

The harness stands up K shard pairs over one shared in-memory transport, runs their block loops concurrently, drives a workload, and measures aggregate throughput. It verifies value conservation on every run and refuses to report a number if it fails.

```
# Independent pairs: K accounts/market pairs, each self-contained.
go run ./cmd/shardbench -mode pairs  -sweep 1,2,4,8 \
     -accounts 20000 -marks 20000 -target 256 -interval 1ms

# Fanout: ONE accounts shard behind K markets -- the shared-shard wall.
go run ./cmd/shardbench -mode fanout -sweep 1,2,4,8 \
     -accounts 4000 -marks 4000

# Grid, FIXED total workload: 16,000 accounts split across N accounts shards.
go run ./cmd/shardbench -mode grid   -sweep 1,2,4,8 \
     -markets 8 -total 16000 -marks 2000
```

Each takes seconds. Reported figures are the best of three runs. Efficiency is $\text{tps}(K)/(K \cdot \text{tps}(1))$; with `-total` the last column reports speed-up over $N = 1$ instead, because the workload is held constant.

C.2 Primitive and per-operation costs

The primitive costs of Table 4 are Go microbenchmarks over `gl.TwoToOne` and `gl.SparseTree.SetLeaf` at depth 24, run with `-benchtime 2s`. The marginal per-operation costs of Table 5 are single-pair runs driving one operation class to exhaustion — local credits on a disjoint identifier range so they emit nothing, cross-shard escrows, and local mark updates — with the class count divided into the measured wall-clock, best of five.

C.3 Proving

The proving benchmarks are opt-in, because they prove many recursive nodes and take minutes:

```
cd zk && cargo test --release --test proving_bench -- \
     --ignored --nocapture --test-threads=1
```

This runs four benchmarks in sequence, about four minutes in total:

`aggregation_cost_curve` folds $k = 1 \ldots 16$ conservation channel partials up the recursive tree, reporting the total fold, the per-node cost and the root check (Table 11).

`parallel_aggregation_critical_path` folds the same tree level by level with each level's independent nodes on concurrent threads, reporting this machine's wall-clock against the sequential and

ideal-fleet models (Table 12).

range_fold_cost_curve proves 16 chained credit blocks and folds them into ranges — the within-shard twin of the aggregation curve.

bn254_wrap_cost re-proves under the BN254-hashed configuration. Note that this is the small test wrap, *not* the settlement wrap; see Section 6.6.

The multi-process variant, which folds the tree across separate worker processes exchanging proofs as bytes on disk:

```
cd zk && cargo test --release --test multiprocess_bench -- \
    --ignored --nocapture
```

C.4 The full pipeline

To run the whole thing — shard nodes, the commitment aggregator, and the prover daemon — and watch a committed root become a proven one:

```
# Two shard processes + aggd + scaleproverd, with a circuit-shaped workload.
go run ./cmd/scalepipeline

# The grid: one accounts shard fanning out to K markets.
go run ./cmd/scalepipeline -markets 4
```

Recall from Section 7 that provable runs must be circuit-shaped: one funding credit then one open per accounts block, one escrow-in per market block, and no mark updates. The throughput harness above deliberately produces multi-record blocks, which the job builders reject — so the execution and proving measurements are of the same system but not of the same runs.

C.5 Conformance

What keeps the Go specification and the Rust circuits honest about each other is a set of pinned conformance vectors. Fifteen Go tests — `TestVector...` across `scaling/` — emit the commitments for a chosen scenario: a single credit, a multi-record block, a consume, a multi-peer emit and consume, the escrow join and reject cycle, the grand product, the aggregator root, the challenge derivation, the grid, and the split and merge roots. Those values are hard-coded into the corresponding Rust pipeline tests, which recompute them in-circuit and check equality.

```
go test ./scaling/...          # the executable specification, incl. vectors
cd zk && cargo test --release  # the circuits, checked against those vectors
```

The pinning is manual rather than generated, so its coverage is the coverage of those fifteen scenarios: a change to either side that alters a *pinned* commitment fails here rather than at settlement, and a change to a commitment no vector exercises does not. Extending the vectors as circuits are added is therefore load-bearing maintenance, not optional hygiene.