D-Chain Deposit/Withdraw Consensus-Wire Activation: Proposer-Relayed-Once Custody Acknowledgements Carried ~~in the ZAP-Proxy~~ Block

Zach Kelling

Lux Industries

Abstract

The D-Chain is the order-book venue: it owns balances, the book, matching, and settlement. The public-network DEX VM (`chains/dexvm`) is a stateless atomic ZAP proxy — it holds no order, pool, position, or balance keys; it moves value in and out of the venue atomically via cross-chain shared memory and relays order traffic. LP-223-adjacent work (the *carried-fills* consensus fix) established that a moving venue ledger makes per-validator relay non-deterministic: if each validator independently relayed a `clob_submit` during block verification, each would observe a different ledger and derive different fills, forking the state root. The fix makes the *proposer* relay the submit once at `BuildBlock` and *carry* the resulting fills in the block and DAG-vertex bytes, so every validator settles from identical bytes and reaches an identical state root. That closed the order-submission path. It deliberately did *not* close the deposit/withdraw custody path, which is proven against the D-Chain VM directly and against the proxy atomic rail (real `atomic.SharedMemory`) but is *not* yet carried in the proxy consensus block. This LP specifies that follow-up. The proxy block and DAG vertex carry *deposit acknowledgements* (the import-then-credit result) and *withdraw-realized / export commitments* (the debit-then-export result) so all validators apply the identical custody delta. We specify replay protection (consumed-UTXO set ∩ per-transaction idempotency), a network-upgrade activation height H with strict pre-/post-H rejection (no ambiguous window), reuse of the reserved fill-signature field that the carried-fills layout already carved for the trustless path, the conservation invariant that gates release, the proposer-trust security surface (conservation-bounded, no supply inflation), and the multi-validator acceptance gates (deposit, order-lock, match, withdraw — all validators reach an identical state root). This is a design specification; it does not implement the wire.

Contents

1 Motivation

1.1 The decomplected venue

The DEX is decomplected into orthogonal homes with one concern each. The D-Chain is the venue: it owns the per-account ledger, the order book, the matcher, and settlement. The public-network DEX VM (`chains/dexvm`) is a *stateless atomic ZAP proxy*. The proxy holds no order, pool, position, or balance keys; its only jobs are (i) moving value into and out of the venue atomically across chains via shared memory, and (ii) relaying order traffic to the venue. The venue is the single source of truth for money and matches; the proxy is transport. The atomic value movement reuses the cross-chain all-or-nothing discipline of LP-211 [5]; the relay rail rides the ZAP universal wire of LP-201 [2]; the proxy block is a DAG vertex in the mempool of LP-208 [3].

The custody model, in one line per phase, is:

- **Deposit** = `ImportTx` (an atomic debit of a C-chain or X-chain UTXO, consumed exactly once) *then* credit of the D-Chain account's `available` balance.
- **Order** = move from `available` to `locked` (buy locks $\lceil \text{size} \cdot \text{price} \rceil$ quote units; sell locks size base units).
- **Match** = settle `locked` against `available` on *both* legs, wei-exact, plus fees; every unit leaving one account's `locked` arrives in a counterparty's `available`.
- **Withdraw** = debit the D-Chain account's `available` *then* `ExportTx` (an atomic release back to the originating chain).

1.2 Why per-validator deposit/withdraw relay would fork

The carried-fills work established the governing fact for order submission: **a moving venue ledger makes per-validator relay non-deterministic**. If each validator independently dialed the venue during `Block.Verify` to relay a `clob_submit`, each would observe the venue ledger at a different instant and derive a different fill set, forking the state root. The cure was to make the *proposer* relay the submit once at `BuildBlock` and carry the resulting fills in the block and DAG-vertex bytes; every validator then re-derives the same state transition as a pure function of (height, carried timestamp, transaction bytes, carried fills), so all validators reach a byte-identical state root and zero validators dial the venue during verification.

The deposit and withdraw flows have the *same* hazard for the *same* reason. Both mutate the venue ledger, and both depend on the venue's response to a relay:

- **Deposit fork.** `executeDeposit` performs the atomic import (consume-once) then relays `clob_deposit` to the venue, which credits `available` and refuses if the credited amount does not equal the imported amount. The credited amount, the resulting account state, and the deposit acknowledgement are venue responses. If each validator relayed independently during verification, each could observe a different ledger snapshot and apply a different credit (or observe a different idempotency outcome). The deposit acknowledgement is exactly the kind of "venue said this happened" datum that the carried-fills argument forbids re-deriving per validator.
- **Withdraw fork.** `executeWithdraw` relays `clob_withdraw`, which debits `available` and returns the *realized* amount clamped to the actually-available balance, then performs the atomic export of exactly that realized amount. The realized amount is a function of the venue ledger at relay time: two validators relaying independently against a moving ledger could compute different realized amounts, hence different export commitments, hence different state roots — and worse, different on-chain releases.

1.3 The consequence the carried-fills work avoided, restated

For order submission the divergence was a forked state root. For custody the divergence is strictly worse: a withdraw realized amount that differs across validators is a divergence in *how much value leaves the system on the originating chain.* The deposit acknowledgement that differs across validators is a divergence in *how much value the venue believes entered.* A consensus that cannot agree on these numbers cannot agree on the conservation invariant of Section 3, and the venue is not safe to decentralize.

1.4 The fix, by analogy

This LP applies the carried-fills mechanism to custody. The proposer performs the irreversible, non-deterministic venue interaction *once* at `BuildBlock` — the import-then-credit for deposits, the debit-then-export-commitment for withdrawals — and carries the *results* in the block and DAG-vertex bytes. Every validator then applies the carried custody delta as a pure, deterministic function of the block bytes: no validator dials the venue, no validator re-samples a moving ledger, and the block's effect on the state root is identical network-wide. The venue ledger remains the source of truth; the proxy block becomes a faithful, replayable record of the custody transitions the proposer already executed against it. The pattern is the prepare-then-commit shape of two-phase commit [1], specialized so the "prepare" is a single proposer relay and the "commit" is the deterministic application of carried results by every validator.

This is deliberately the smallest change that closes the fork: it reuses the existing carried-payload framing, the existing consume-once/idempotency machinery, and the reserved signature field already present for the trustless path. It introduces no new venue semantics — only a deterministic way for every validator to agree on the custody transitions the venue performed.

2 Specification

This section specifies the carried custody payloads, their placement in the block and DAG vertex, replay protection, the network-upgrade activation height, and the strict backward-compatibility regime. The specification is additive: the four frozen CLOB ZAP frames (`clob_ensure_market`, `clob_place`, `clob_cancel`, `clob_submit`) are untouched, and the carried-fills section appended by the order-submission work is unchanged. Custody uses the additive ZAP frames `clob_deposit`, `clob_withdraw`, and `clob_open_market`, which already exist on the venue rail; this LP does not add or alter any ZAP frame — it adds a *carried block payload* that records the results of those relays.

2.1 Carried-payload model

The proxy block already carries a self-delimiting fills section appended after the transaction bytes, and the block identifier and the DAG-vertex identifier already commit to that section (so tampering is detectable). This LP adds two further carried sections, governed by the same rules:

1. The proposer produces the section once at `BuildBlock`, as the result of the venue relay it performs there.
2. The section is appended to the block bytes and to the DAG-vertex bytes, length-prefixed and self-delimiting.
3. The block identifier and the DAG-vertex identifier commit to the section (it is inside the hashed pre-image), so a proposer cannot carry one set of custody results in the bytes and a different set in the identifier.
4. Every validator applies the section deterministically at `Block.Accept`; no validator relays to the venue during `Verify` or `Accept`.

2.2 Deposit acknowledgements

A `DepositAck` records one import-then-credit. It is the datum that lets every validator apply the identical credit to the venue view without re-relaying.

Field	Type	Meaning
`account`	`[16]byte`	D-Chain account the credit lands in
`asset`	`[8]byte`	asset identifier (matches ledger key)
`credited`	`uint256`	amount credited to `available` (wei-exact)
`source_chain`	`uint32`	originating chain (C or X)
`source_utxo`	`[36]byte`	consumed UTXO reference (`txID[32]‖outIdx[4]`)
`tx_index`	`uint32`	index of the deposit tx within the block

Table 1: `DepositAck` carried fields. The credit applied by every validator is `(account, asset) += credited`; the import leg is identified by `(source_chain, source_utxo)` and is consumed exactly once (Section 2.6).

The proposer obtains a `DepositAck` by performing `executeImport` (the atomic, consume-once debit of `source_utxo`) followed by the `clob_deposit` relay, which returns the credited amount and *refuses if credited does not equal imported*. The invariant `credited == imported` is enforced at relay time by the venue and re-checkable by any validator against the consumed UTXO's value (Section 3). The carried `credited` is therefore not a free parameter: it is pinned to the value of the consumed UTXO.

2.3 Withdraw-realized / export commitments

A `WithdrawCommit` records one debit-then-export. The export is *committed* in the block: the carried record names the exact amount that will be released on the originating chain and the identifier of the exported UTXO, so every validator applies the same debit and the same export.

Field	Type	Meaning
`account`	`[16]byte`	D-Chain account debited
`asset`	`[8]byte`	asset identifier
`realized`	`uint256`	amount debited from `available` and exported (wei-exact)
`dest_chain`	`uint32`	destination chain (C or X)
`export_utxo`	`[36]byte`	exported UTXO id produced on `dest_chain`
`tx_index`	`uint32`	index of the withdraw tx within the block

Table 2: `WithdrawCommit` carried fields. The debit applied by every validator is `(account, asset) -= realized`; the export leg releases exactly `realized` to `dest_chain` as `export_utxo`.

The proposer obtains a `WithdrawCommit` by relaying `clob_withdraw`, which debits `available` and returns the *realized* amount *clamped to the actually-available balance* (`realized` $\leq$ `requested`, and `realized` $\leq$ `available` at relay time), then performing `executeExport` of exactly `realized`. The export leg *refuses if it would release more than realized*. Because the realized amount is fixed in the carried record, every validator applies the identical debit and the identical export; the moving-ledger divergence of Section 1 cannot arise.

2.4 Block / vertex layout

The custody payload is two length-prefixed sub-sections appended after the carried-fills section, preserving the existing self-delimiting framing:

```
block.Bytes() =
    txCount[4] | txBytes...                  // existing, self-delimiting
```

```
  | <carried-fills section>                  // existing (order-submission LP)
  | depositCount[4]   | depositCount  x DepositAck
  | withdrawCount[4] | withdrawCount x WithdrawCommit
  | sigLen[4] | sig                         // RESERVED (see below)
```

The DAG-vertex bytes carry the identical custody sub-sections, and the vertex identifier computation folds them into the digest exactly as it already folds the fills section. `DepositAck` and `WithdrawCommit` are fixed-width, so each sub-section is fully described by its `uint32` count prefix; the parser is self-delimiting and rejects trailing or truncated bytes.

2.5 Reuse of the reserved fill-signature field

The carried-fills layout already carved a single trailing `sigLen[4] | sig` field — empty in the default operator-venue mode, reserved for the trustless path in which the venue signs its emitted results and a P3Q [7, 8] / STARK-FRI [9] verifier checks them, so that no *further* wire bump is needed when the trustless path activates.

This LP **extends the meaning of that same reserved field to cover the custody sub-sections**: when populated, the signature is over the concatenation of the fills section *and* the `DepositAck`/`WithdrawCommit` sub-sections, i.e. over all proposer-relayed venue results in the block. No new signature field is introduced. In the default mode the field stays empty and the block is accepted under the proposer-trust surface of Section 4; in the trustless mode — a strictly higher cert profile in the sense of LP-217 [6] — the field carries the venue's signature and validators verify it before applying any carried fills *or* custody deltas. This keeps the trustless escape hatch a single field for the whole carried-payload family — one and only one way to make the carried results trustless.

2.6 Replay protection

Custody must be safe against two replays: crediting a deposit twice and exporting a withdraw twice. Both reuse machinery already present; this LP only binds the carried payloads to it.

Deposit double-credit. The import leg consumes `source_utxo` from cross-chain shared memory exactly once: a consumed UTXO cannot be consumed again, and the consumed-UTXO set is part of the state the block's state root commits to. A `DepositAck` whose `source_utxo` is already in the consumed set is invalid; a block carrying it is rejected. Additionally, the venue's per-transaction `seen:<txID>` idempotency record (content-addressed, established by the order-submission work and the custody rail) makes a re-relayed `clob_deposit` return the *original* outcome rather than crediting again. The two mechanisms are complementary: the consumed-UTXO set prevents replaying the *import* (the value source), and `seen:<txID>` prevents replaying the *credit* (the venue effect). A deposit is therefore credited at most once even across proposer re-builds, vertex re-issue, or crash-before-commit.

Withdraw double-export. The withdraw leg is bound the same way from the other side. The debit of `available` occurs once on the venue under `seen:<txID>`; a re-relayed `clob_withdraw` returns the original realized amount rather than debiting again. The export leg produces `export_utxo` exactly once into shared memory; the same `export_utxo` cannot be produced twice, and a `WithdrawCommit` that names an already-produced `export_utxo` is invalid. A withdraw is therefore exported at most once, with realized amount fixed in the carried record.

Determinism of replay outcomes. Because both the consumed-UTXO set and the `seen:<txID>` records are inputs the state root commits to, two validators that disagree on whether a deposit/withdraw was already applied would produce divergent roots — i.e. a

replay-handling divergence is detected by the same state-root check that detects every other custody divergence. There is no path by which one validator credits/exports and another does not while both accept the same block.

2.7 Upgrade activation height

The carried-custody payload format is a consensus-breaking change to the block and DAG-vertex bytes and to the identifier pre-image. It activates at a network-upgrade height H, exactly as other Lux network upgrades are gated. H is a single fixed height (per network) at which all validators switch in lockstep; it is not negotiated per block and not enabled by individual validator flags.

- For block height $h < H$: the carried-custody sub-sections MUST be absent. Deposit/withdraw proceeds as it does today — proven against the venue VM directly and against the proxy atomic rail — without being carried in the proxy consensus block.
- For block height $h \geq H$: every deposit/withdraw effected by the block MUST be represented by a `DepositAck` / `WithdrawCommit` in the carried sub-sections. A post-H block that effects a custody transition without carrying its record is invalid.

The activation height is published in the network upgrade descriptor and is the same on every node; nodes that have not adopted the upgrade descriptor cannot follow the chain past H, which is the intended forcing function for a lockstep validator upgrade.

2.8 Backward compatibility and pre-upgrade rejection

There is no ambiguous window. The format is gated strictly on H:

Height	Carried custody present	Validator action
$h < H$	yes	**REJECT** (format not yet active)
$h < H$	no	accept (legacy framing)
$h \geq H$	no (but block effects custody)	**REJECT** (record required)
$h \geq H$	yes	accept (verify framing, identifier commitment, replay)

Table 3: Strict pre-/post-H gating. Pre-H blocks MUST reject the new carried-custody payloads; post-H blocks MUST require them whenever a custody transition occurs. No height accepts both shapes.

Because the block and vertex identifiers commit to the carried sections, a pre-H node and a post-H node compute *different* identifiers for any block that carries custody payloads — so the upgrade cannot silently half-apply. The forced divergence at H is exactly the lockstep behaviour Lux network upgrades rely on: there is no window in which a custody transition is ambiguous as to whether it must be carried.

2.9 Forward compatibility with the trustless path

The reserved signature field (Section 2.5) means the trustless custody path does not require a second wire bump after H: moving the venue from operator-trusted to signature-verified populates the existing field. The activation of *signature requirement* (as opposed to format) is itself a later policy gate; this LP specifies the format that makes that gate a flag flip, not a wire change. One wire, two policies (format at H, trust later) — no third format.

3 Conservation Invariant

The release gate for the consensus-wired custody path is a value conservation invariant that MUST hold across the full deposit → trade → withdraw lifecycle, with every validator agreeing

on each term. We state it formally, fix the sign convention so it balances, and then show why the carried-payload mechanism preserves it.

3.1 Ledger terms

Fix an asset and a window of blocks $[0, h]$. All quantities are non-negative integers in the asset's smallest unit (wei-exact). Define, summed over the window:

- I — value *imported* from C/X, i.e. the sum of consumed source-UTXO values across all `DepositAck` records. Each import consumes a UTXO exactly once (Section 2.6).
- A — the D-Chain `available` balance currently held across all accounts.
- L — the D-Chain `locked` balance currently held across all accounts (reserved by resting orders).
- F — fees collected (or burned) by settled matches.
- E — value *exported* back to C/X, i.e. the sum of `realized` across all `WithdrawCommit` records.

3.2 The invariant

The balanced conservation identity is

$$\boxed{I \;=\; A \;+\; L \;+\; F \;+\; E} \tag{1}$$

i.e. every unit that entered the system via an import is, at every height, accounted for in exactly one of: still available, still locked, taken as fees, or already exported. Equivalently, writing the C/X-side net custody as $(\text{released } E) - (\text{consumed } I)$, the form stated in the work item

$$\underbrace{(E - I)}_{\text{C-chain released}-\text{locked/consumed}} \;+\; \underbrace{A}_{\text{D-chain available}} \;+\; \underbrace{L}_{\text{D-chain locked}} \;+\; \underbrace{F}_{\text{fees}} \;=\; 0$$

is the rearrangement of (1) with the C/X custody change moved to the left; before any withdrawal ($E = 0$) it reduces to the "Σ available $+ \Sigma$ locked + fees = initial deposited" form, with I the initial deposited total. We treat (1) as the canonical statement because every term is a non-negative quantity the state root already commits to.

3.3 Per-transition deltas (why each phase preserves it)

Each custody/match transition moves value between the terms of (1) without changing the total:

3.4 Why the consensus wire preserves it across validators

The novelty this LP must defend is not the single-venue invariant — that is proven against the venue VM directly and against the proxy atomic rail. The novelty is that *every validator* must agree on each term of (1) once the transitions are carried in the proxy consensus block. This follows from three facts established in Section 2:

1. **Each term is a deterministic fold of carried records.** I is the sum of `DepositAck.credited`; E is the sum of `WithdrawCommit.realized`; A, L, F evolve by the per-transition deltas of Table 4, all driven by carried records and carried fills. No term depends on a per-validator venue query.
2. **Carried records are pinned, not free.** The carried `credited` is pinned to the consumed-UTXO value (the import refuses otherwise); the carried `realized` is pinned $\leq$ `available` at relay and the export refuses to release more. A proposer cannot carry an inflated credit

Transition	Effect on (1) terms
Deposit	$I+ = v$, $A+ = v$ (with carried `credited` $= v =$ consumed-UTXO value). Both sides grow by v; identity preserved.
Order-lock	$A- = r$, $L+ = r$ (reserve $r = \lceil \text{size} \cdot \text{price} \rceil$ quote, or size base). Internal $A \to L$ move; total unchanged.
Match	for each filled unit, $L- = x$ on one account and $A+ = (x - f)$ on the counterparty, $F+ = f$. Both legs settled; L down by x, $A+F$ up by x; total unchanged. (The maker leg is credited, not dropped.)
Refund of unspent lock	$L- = u$, $A+ = u$ for the unfilled remainder. Internal $L \to A$ move; total unchanged.
Withdraw	$A- = w$, $E+ = w$ (with carried `realized` $= w \leq A$ at relay). A down by w, E up by w; total unchanged.

Table 4: Each transition is a value-preserving move between the terms of (1). No transition creates or destroys units; fees are an explicit term, not leakage.

or an over-realized withdraw without the import/export leg refusing — so a malformed carried record does not even produce a valid block.

3. **Identical bytes ⇒ identical terms ⇒ identical root.** Every validator folds the identical carried bytes, so every validator computes identical I, A, L, F, E, so every validator commits the identical state root. The state transition is a pure function of the block bytes — the same determinism discipline LP-210 [4] requires for parallel execution. A divergence in any term would surface as a divergent root and the block would not gain consensus.

3.5 The release gate

Release gate. The consensus-wired custody path is releasable when (1) holds at every height across a multi-validator deposit → order-lock → match → withdraw run *and* all validators commit the identical state root at every height. The single-venue proof of (1) is necessary but not sufficient; the multi-validator agreement on every term is the additional condition this LP exists to make checkable.

The acceptance gates of Section 5 are the concrete multi-validator checks that discharge this gate.

4 Security Considerations

4.1 Proposer-trust surface

The carried-payload model concentrates the irreversible venue interaction in the proposer: the proposer performs the import-credit and the debit-export once at `BuildBlock`, and validators replay the *results*. This is the same trust surface the order-submission work established for fills, and it is bounded the same way.

What the proposer can do. A proposer can build a block whose carried custody records reflect venue relays it performed, and then *fail to win the consensus poll* for that block. If the block does not gain consensus, its custody transitions are not committed to the canonical chain, yet the proposer already mutated the venue ledger (and, for a withdraw, may have produced an export UTXO). This is the same "built-and-relayed but lost the poll" surface as for fills.

What the proposer cannot do. The surface is *conservation-bounded*. A proposer cannot inflate supply:

- It cannot carry a `credited` larger than the consumed-UTXO value — the import leg refuses, so no such block is even valid (Section 3, fact 2).
- It cannot carry a `realized` larger than the account's `available` at relay, nor export more than `realized` — the export leg refuses.
- It cannot double-credit a deposit or double-export a withdraw — the consumed-UTXO set and `seen:<txID>` forbid it (Section 2.6), and both are state the root commits to, so a divergent replay outcome is a divergent root.

Therefore the worst a malicious or faulty proposer achieves is a *stranded* custody transition (Section 4.4), never created or destroyed value. The blast radius of a single bad block is a single deposit or a single withdraw escrow, bounded by that transaction's own value. This matches the accepted single-operator-venue posture today; the trustless path of Section 2.5 removes even the stranding risk by requiring the venue's signature over the carried results.

4.2 Reorg

A reorg that drops an accepted block drops its carried custody records with it. Two cases:

- **Deposit in a reorged-out block.** The import consumed a source UTXO. If the block is reorged out, the consumed-UTXO set entry is rolled back along with the rest of the block's state, so the UTXO is spendable again and the deposit can be re-proposed in the new canonical chain. No double-credit can occur because the credit only exists where the consuming block exists.
- **Withdraw in a reorged-out block.** The debit of `available` is rolled back with the block. The export UTXO, however, is produced into cross-chain shared memory at `Accept`; a reorg that removes the producing block must also remove its shared-memory effect, or the system would have exported value with no backing debit. The withdraw path therefore MUST tie the export's shared-memory production to the same atomic commit point as the block's acceptance, so that reorg removes both or neither.

Requirement. The custody records and their cross-chain shared-memory effects MUST be committed at a single atomic point with the block's acceptance — the same all-or-nothing discipline the proxy already uses for imports/exports. The carried record is the in-consensus half; the shared-memory apply is the cross-chain half; they are one atomic step.

4.3 Replay

Replay is addressed structurally in Section 2.6 and is restated here as a security property: there is no sequence of proposer re-builds, vertex re-issues, crash-before-commit, or network re-delivery that credits a deposit twice or exports a withdraw twice, because (i) the import consumes its UTXO exactly once, (ii) the venue `seen:<txID>` returns the original outcome on re-relay, and (iii) both facts are committed to the state root, so any validator that replayed differently would fork the root and lose consensus. Replay safety is thus not a best-effort property of the proxy; it is enforced by the consensus check itself.

4.4 Failed-export recoverability

The only residual hazard is a *stranded* custody transition: a block whose proposer mutated the venue (and possibly began an export) but which did not gain consensus, or whose export leg failed after the venue debit. The governing requirement is that funds are always *unspent and recoverable, never burned*:

- **Failed export after debit.** If the venue debited `available` but the export leg failed, the realized amount MUST be returned to the account (full refund of the would-be-withdrawn amount), exactly as the order-submission failure path full-refunds escrow rather than stranding it. The withdraw then either does not occur (account made whole) or is re-attempted; it never half-occurs.
- **Lost-poll block.** A block that loses the consensus poll does not commit its custody transitions to the canonical chain. The source UTXO of a deposit remains unconsumed on the canonical chain and is re-proposable. For a withdraw, the atomic discipline of the reorg requirement ensures the export UTXO is not produced on the canonical chain if the debit is not committed there — so the funds remain on the venue, available, and recoverable.
- **Never burned.** No failure path destroys units. A stranded transition leaves value either on the originating chain (deposit not finalized) or on the venue (withdraw not finalized), always spendable. This is the conservation invariant of Section 3 holding even under failure: $I = A + L + F + E$ never loses a term to a black hole.

4.5 Determinism as a security property

Finally, the determinism that motivates the whole LP is itself a safety property: because every validator applies byte-identical carried records and reaches an identical state root, there is no state at which honest validators disagree about how much value entered or left the system. Disagreement about custody is, for a value-bearing venue, the most dangerous failure mode; the carried-payload model removes it by construction rather than detecting it after the fact.

5 Test and Acceptance

The single-venue custody model is already proven by two suites: a real-VM conservation test against the D-Chain ledger (deposit → order-lock → match → withdraw with value conservation and restart survival) and a real-shared-memory atomic-rail test (import/export all-or-nothing). This LP requires the *multi-validator analog*: the same hard gates, but with $N \geq 3$ validators consuming identical carried blocks and reaching an identical state root at every height. Acceptance is the conjunction of the gates below; each is the multi-validator lift of an already-passing single-venue gate.

5.1 Hard gates

1. **Deposit gate.** A deposit block carries a `DepositAck` with `credited` equal to the consumed-UTXO value. All N validators apply the identical credit, the consumed UTXO is in every validator's consumed set, and all N commit an identical state root. *Negative:* a block carrying `credited` $\neq$ consumed-UTXO value is rejected by every validator (invalid import leg). *Replay:* re-delivering the deposit block, or a proposer rebuild, credits exactly once across all validators.

2. **Order-lock gate.** An order block moves `available` → `locked` by the exact reserve ($\lceil \text{size} \cdot \text{price} \rceil$ quote for a buy, size base for a sell). All N validators apply the identical lock and commit an identical root. *Negative:* an unfunded order (reserve > `available`) is rejected identically by all validators (custody gate: lock before match).

3. **Match gate.** A match block carries fills (existing carried-fills section); all N validators settle both legs wei-exact — the maker is credited the taker's payment and the taker is credited the maker's asset, fees taken into F — and refund any unspent lock. All N commit an identical root, and the per-asset sums A, L, F match across validators. *Determinism:* the proposer relays the underlying `clob_submit` once at `BuildBlock`; **zero validators**

dial the venue during `Verify/Accept` (the carried-fills property), and the same must hold for the custody relays.

4. **Withdraw gate.** A withdraw block carries a `WithdrawCommit` with `realized` $\leq$ `available` at relay; all N validators apply the identical debit and observe the identical `export_utxo`, and all N commit an identical root. The exported amount equals `realized` on the destination chain. *Negative:* a block whose export leg would release more than `realized` is rejected by every validator. *Replay:* re-delivery or rebuild exports exactly once.

5.2 End-to-end multi-validator conservation

The capstone acceptance is the end-to-end run with $N \geq 3$ validators:

1. Deposit a maker account and a taker account (distinct assets).
2. Place a resting maker order (lock applied).
3. Cross with a taker order (one or more consensus fills settled in the ledger; maker credited, taker credited).
4. Withdraw both accounts.

Acceptance holds when, *at every block height across the run*:

- all N validators commit a byte-identical state root;
- the conservation invariant $I = A + L + F + E$ (Section 3) holds, with every term computed identically by every validator;
- the per-account `available` and `locked` balances match across all validators;
- the realized withdraw totals equal the deposited totals net of fees ($E + A + L + F = I$, and at the end with all balances withdrawn, $E + F = I$).

5.3 Failure and restart gates

1. **Lost-poll strand.** A block the proposer builds and relays but which loses the consensus poll commits no custody transition to the canonical chain across any validator; the source UTXO remains unconsumed (deposit) or the venue balance remains available (withdraw). No value is created or burned (Section 4.4).
2. **Failed export refund.** A withdraw whose export leg fails after the venue debit results in a full refund to the account; the end-state conservation still holds.
3. **Restart.** After a node restart at the end of the run, the per-account `available` and `locked` balances survive (rebuilt from persisted rows) and match the pre-restart values on every validator — the multi-validator lift of the single-venue restart proof.

5.4 Activation gates

1. **Pre-H rejection.** A block at height $h < H$ that carries custody sub-sections is rejected by every validator (Section 2.8).
2. **Post-H requirement.** A block at height $h \geq H$ that effects a custody transition without carrying its record is rejected by every validator.
3. **Identifier divergence at H.** A pre-H node and a post-H node compute different identifiers for any block carrying custody payloads, confirming the format cannot silently half-apply.

5.5 What "done" means

This LP is dischargeable when all gates above pass with $N \geq 3$ validators on a network that has activated the upgrade at H, with the evidence artifact recording: validator count, commit SHAs and released versions, the activation height H, per-block state roots (identical across

validators), the deposit/withdraw `credited/realized` values and UTXO references, the fill identifiers, before/after `available+locked` per account, and the final conservation output $I = A + L + F + E$. Until those gates pass on a multi-validator network, the format is specified and the single-venue model is proven, but the consensus-wired path is not released — which is precisely the boundary this LP draws.

References

[1] Jim Gray. Notes on database operating systems, 1978. IBM Research RJ2188; the original two-phase-commit treatment.

[2] Lux Industries. Lp-201: Zap universal wire, 2026.

[3] Lux Industries. Lp-208: Dag mempool, 2026.

[4] Lux Industries. Lp-210: Block-stm optimistic parallel execution, 2026.

[5] Lux Industries. Lp-211: Cross-shard atomic polaris cert, 2026.

[6] Lux Industries. Lp-217: Cert profile modes, 2026.

[7] Lux Industries. Lp-218: Zap p3q post-quantum rollup, 2026.

[8] Lux Industries. Lp-219: P3q framework, 2026.

[9] Lux Industries. Lp-221: Stark-fri verifier, 2026.