# Clowns in a Very Small Car

## On Increasing Bandwidth & Reducing Latency: Fault-Provable Extended Blockspace for the Solana Virtual Machine

Praetorian
verity.dev

**Abstract.** Programs written for the Solana Virtual Machine share one global ledger. Applications that need many small transactions at low latency, such as central limit order books, pay for that contention in fees and unpredictable inclusion. We describe Verity, a rollup that executes unmodified Solana programs in 87 millisecond blocks and settles to Solana. Blocks are produced on a fixed deadline. They are executed in parallel by an engine whose result is defined to equal serial execution, and committed under a state root taken after every transaction. Batches of blocks are anchored on Solana with their state commitments. Any party may dispute a commitment through an interactive bisection game that narrows the disagreement to a single RISC-V instruction, which a Solana program executes to decide the outcome. No trusted adjudicator and no zero-knowledge proof system are involved. Users can force transactions into blocks through Solana, and anyone with stake can take over block production if the sequencer stops posting. We report the problems found while building and operating the system, the mechanisms adopted in response, and measurements from development benchmarks and the public testnet.

### 1. Introduction

Solana gives programs a single, fast, shared state. That is its strength, and it is also the limit for one class of application. A perpetual futures exchange with an on-chain order book writes the same few accounts in nearly every block. These include order books, oracle pages and settlement queues, written by traders, market makers and keepers who all want them in the same slot. Each such application competes with every other program on the network for inclusion, and it cannot choose its own block time, fee schedule or admission policy.

An execution layer above Solana can make those choices, provided users lose nothing that makes Solana worth settling to. Users must be able to check the layer's state transitions, reach their funds without the operator's cooperation, and get transactions included without the operator's permission. Proving execution with zero-knowledge systems answers the first requirement at a cost that grows with every instruction executed. Optimistic designs avoid that cost but need a way to settle disputes. An administrator key that adjudicates challenges is not acceptable for this purpose. Our earliest settlement design had one, and removing it shaped most of the system described here.

Verity is an optimistic rollup that runs the Solana Virtual Machine (SVM) [1] and settles to Solana. A sequencer seals a block every 87 milliseconds. A Merkle commitment to the resulting state is anchored on Solana, together with the batch of blocks that produced it. A commitment becomes final after a challenge window. During that window any staked party may dispute it through an interactive bisection game in the line of [2][3]. The game descends from a batch to one block, one transaction and finally one machine instruction of a RISC-V program [4]. That program is the chain's state transition function, compiled from the same source as the node. Solana executes the disputed instruction and the loser is slashed. Forced in-

clusion and an escape hatch, both driven from Solana, bound what a faulty or absent operator can withhold.

The sections that follow describe each part of the system in the order a transaction meets it. In each part we record the failures we found while building and running it, because several of them changed the design.

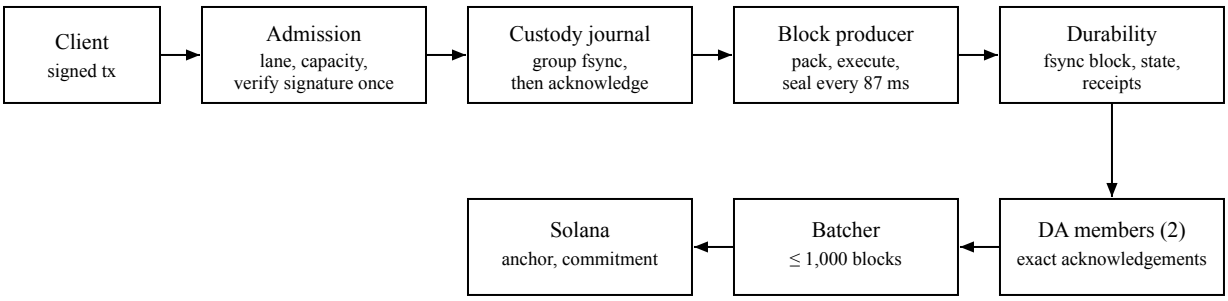
## 2. Transactions and Custody

A transaction is a standard Solana transaction, and wallets, SDKs and programs need no changes to produce one. The node admits it in a fixed order:

1. decode;
2. check the rate lane;
3. reserve capacity in custody;
4. verify signatures;
5. append to the custody journal;
6. acknowledge.

Every refusal happens before any signature work, so an overloaded node does not spend its CPU on transactions it will reject. Signatures are verified exactly once, at admission, and execution never repeats that work.

The custody journal is the mempool. Each admitted transaction is written as its exact wire bytes plus a 13-byte header, in group commits to preallocated segments. An acknowledgement from the node means those exact bytes are on disk. Custody holds up to 500,000 transactions in 512 MiB, with 50,000 places reserved for ordinary users. If the node restarts, it re-verifies every recovered record before producing again; after a forced kill holding 500,000 acknowledged transactions, this took 8.5 seconds.




Two findings shaped custody. The first concerned durable appends. An earlier design gave every acknowledgement its own durable journal append, which held ingest near 1,500 transactions per second regardless of hardware. Group commit, one journal group per batch request, removed that ceiling.

The second concerned how long custody holds a transaction. An earlier design kept each acknowledged transaction until its batch was final on Solana, and released it through a pass that replayed state from the write-ahead log once per transaction. With 165,000 transactions pending, that pass held the engine lock for three seconds out of every five, and blocks took 6 to 26 seconds. A bounded journal released only on Solana finality has a hard ceiling: 500,000 slots spread over a 755-second finality horizon is about 660

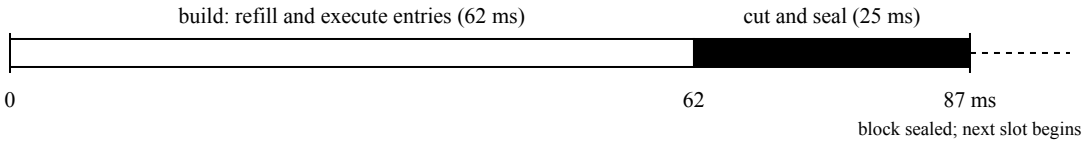
transactions per second. Custody now retires a transaction once its block is durable on the producer and acknowledged by a data availability member on a separate machine. That point arrives within seconds of execution rather than minutes.

### 3. Block Production

Blocks are produced on a deadline grid. One block is sealed every 87 milliseconds, even when it is empty, and a late block shifts the grid rather than triggering a burst of catch-up blocks. Within a slot, 62 milliseconds are spent filling the block and 25 are reserved for sealing:

- executing the last entry;
- encoding and signing;
- applying the result under the engine write lock;
- a safety margin.

Timestamps carry milliseconds, because an 87-millisecond chain cannot be described in seconds. A block holds at most 8,192 transactions and 2 MiB.




All durable work is moved off this path. The write lock covers only in-memory work, about 11 milliseconds for a full block. A separate durability thread group-syncs the block log, state, receipts and custody, then publishes a *durable height*. Nothing external reads above that height: not anchoring, not data availability, and not custody retirement. Production therefore never waits for disk, and never waits for Solana. In earlier builds the syncs ran inline, and under disk pressure they stretched the write lock to 300 milliseconds.

A recent blockhash stays valid for 690 blocks, or 60 seconds. The lifetime is counted in blocks, not time, so the cadence is held fixed to keep the two in step. That decision came from an incident described in Section 15. In it, clients retrying stale transactions made blocks arrive faster. A window of 150 blocks then shrank in wall time, which expired still more transactions and fed further retries. A fixed grid, with empty blocks sealed on schedule, removes the coupling between offered load and block rate.

Confirmation latency is measured on the server, from the node receiving a transaction to the transaction being visible over RPC. On the public testnet under mixed live traffic it is 78 milliseconds at the median, 131 at the 95th percentile and 164 at the 99th. The same measurement on the morning those changes began gave 147 and 586 milliseconds at the median and 99th percentile. Three changes account for the difference:

- The producer previously took candidate transactions only at the start of a slot, adding about 55 milliseconds of waiting at the median. It now refills the block until the cut.
- Accounts written every block by oracle and crank transactions overflowed the build window. Oracle pushes are now made per market on a 3 basis point move or a two-second heartbeat, and cranks are coalesced. The share of blocks that deferred transactions fell from 51% to 6.5%.

- The estimator that schedules the cut learned its per-transaction cost only from blocks of 256 transactions or more, so it underestimated small bursts. It now learns the base cost from nearly empty blocks and the marginal cost from the rest. Under an offered load of 8,000 transactions per second, the block interval holds at 87.0 milliseconds at the median and 91.6 at the 99th percentile.

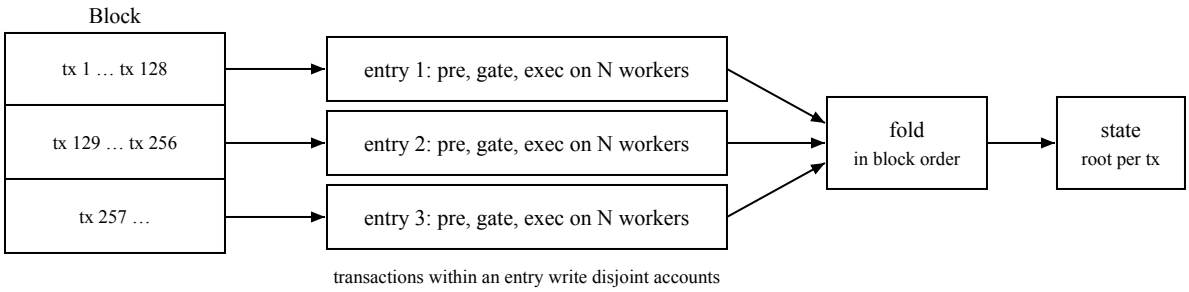
### 4. Parallel Execution

The executor is the Agave SVM, used unmodified, over a state interface specific to Verity. The same crate is compiled twice: natively for the node, and to RISC-V for the fault proof guest (Section 8).

The packer divides each block into *entries*, each holding up to 128 transactions whose account write sets do not conflict. Each entry runs in four phases across a pool of workers:

- **pre**: per-transaction checks;
- **gate**: admission against block limits;
- **exec**: SVM execution;
- **fold**: applying results to state in block order.

By construction the result equals executing the block serially, in order. The producer commits the results it executed and never runs a block twice.




Gas is a consensus rule, not a price. It bounds how much work one block and one disputed step can contain:

$$gas = 64 \cdot CU + 2^{20} \cdot signatures + 4 \cdot bytes\ loaded + 64 \cdot bytes\ hashed\ into\ the\ state\ tree + 2 \cdot bytes\ of\ program\ code\ verified + 4 \cdot bytes\ copied$$

A block admits up to $2^{35}$ gas of ordinary transactions plus a reserved lane of $2^{27}$ for forced transactions (Section 10). A single transaction is limited to $2^{29}$. The alt_bn128, Poseidon and curve25519 group syscalls are disabled, because no program on Verity needs them and pricing them for the fault proof was not worthwhile.

Building this engine surfaced four findings.

**Inherited CPU affinity.** Helper threads spawned by the pinned block thread inherited its single-CPU mask. The parallel commit, transaction preparation, receipt encoding and the block publisher therefore all time-shared one core. Correcting the masks cut commit cost from 9.9 to 3.7 microseconds per transaction, and raised throughput on the same hardware from 24,000 to 29,000 transactions per second.

**Program cache divergence.** A differential harness compares the parallel engine with serial execution block by block. It found that the sequence *invoke P, upgrade P, invoke P* within one block failed the first

invocation on a warm shared cache, while serial execution landed it. A second case involved a program created in one transaction and deployed in a later one. Either case would have let any owner of an upgradeable program produce a state root that honest replicas disagree on. The cache is now purged per transaction where programs change. Cold-cache and warm-cache execution are compared in testing as a separate mode.

**Hot accounts.** Parallelism ends where write sets meet. A program whose every trade wrote one treasury account measured 1.0 millisecond per transaction in 128-transaction blocks and 2.5 milliseconds in 350-transaction blocks. Lock contention grew faster than load. An exchange design that kept all markets in one 444 KB account serialized every trade, and also made each trade expensive to prove. The exchange program was rebuilt with a 6 KB book per market and 1.3 KB oracle pages of 16 markets each. Application layout, not the engine, sets the ceiling here, and we document it for developers as a first-order design rule.

**Where time goes.** A plain transfer costs the worker pool about 75 microseconds of CPU in total. About a third goes to account reads, 14% to formatting log strings in base58, and 13% to per-transaction setup in the SVM. These are the next optimisation targets. The serial floor of the current engine is 23 to 27 microseconds per transaction. The serial engine it replaced spent about 630, because it executed every transaction twice and checked signatures three times.

### 5. State Commitment

State is committed by a compact binary Patricia tree [7] over account keys, with account data hashed as 32-byte words [6]. Over 935,520 live keys, the mean leaf depth measured 21.2, so a typical opening is about 22 hashes. Deliberately grinding keys can lengthen one opening, but only to 256 siblings (8.4 KB), and cannot break it. Protocol leaves hold:

- the block height;
- the forced-inclusion cursor;
- gas and work counters;
- the millisecond timestamp;
- signatures, for replay protection.

The state transition is defined over *segments*: a block prologue, each forced entry, each transaction and a block epilogue. A state root is taken after every segment. That is what lets a dispute isolate one transaction, and it is the main execution cost of being fault-provable. Each block also carries the hash of its parent's output:

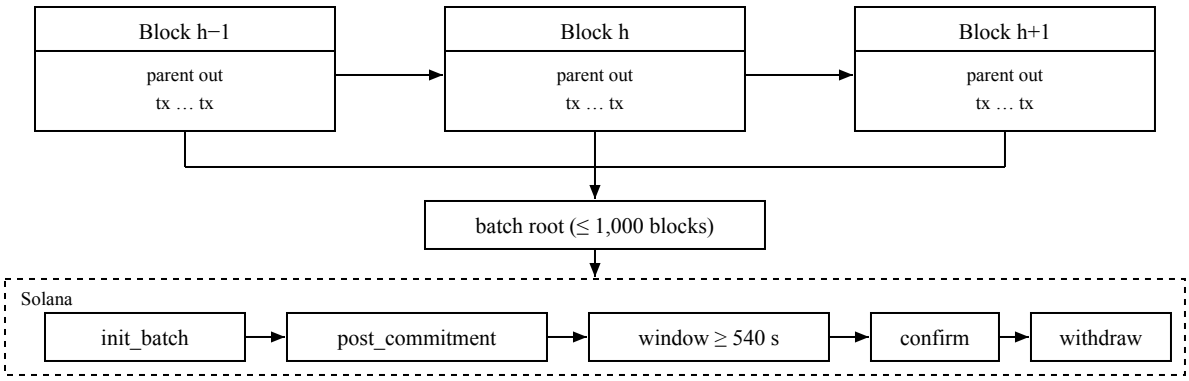
$$out = SHA\text{-}256(domain \parallel S \parallel W \parallel cursor)$$

Here S is the state root, W is the root of the withdrawal accumulator and cursor is the forced-inclusion position. The tree is split into 256 shards that are rehashed lazily and in parallel at commit. A full block of 5,655 transfers commits in 10.5 milliseconds. A fully expanded 256-level sparse tree, which we evaluated first, would have needed about 10 GB of memory and 50 microseconds per write.

### 6. Settlement

The batcher cuts a batch of up to 1,000 consecutive blocks, never past the durable height or the prefix acknowledged by the second data availability member. The batch root is a Merkle root over block roots, and each block root binds the block's bytes and length. Settlement on Solana takes four instructions:

- `init_batch` anchors the batch: its root, byte length, declared work counters and a snapshot of the forced-inclusion queue.
- `post_commitment` records (S, W, cursor, out) for the batch's final block.
- `confirm` is open to anyone once the challenge window has passed with no dispute or availability request outstanding. Batches confirm strictly in order.
- `finalize_withdrawal` pays a withdrawal on Solana against a 32-level Merkle proof into W.




The settlement program is built reproducibly. The native build is byte-identical to the build made in the CI container. Its expensive instructions are bounded and measured:

| Instruction | Compute units |
|---|---|
| init_batch | ≈ 12,000 |
| post_commitment | ≈ 15,000 |
| prove_step (worst case measured) | 22,900 |
| write_da_chunk | 34,052 |
| finalize_withdrawal | ≈ 10,000 |

Settlement produced the most consequential findings in the project.

**A stranger could halt settlement.** The account address of the *next* batch was predictable. Recovery logic walked history on startup, so a single ordinary transfer to that address made it fail, and no further batch could anchor. Halting settlement cost one transaction fee. Recovery now treats any batch number at or beyond the next expected one as absent before reading history.

**Execution must not read settlement time.** An earlier execution policy wrote the Solana anchor timestamp into state. Solana assigns that timestamp after the blocks already exist, so the producer's provisional value never matched. Every batch had to be re-executed and rewritten: 185 of 185 in one sample. Each rewrite made RPC unavailable for about 26 seconds and executed every transaction three times. The same policy also capped useful work at roughly ten transactions per second. Execution is now *context-free*: a block's result depends only on inputs the chain itself carries. Because the policy was fixed at program initialization on Solana, removing it meant deploying a new settlement program.

**Restarts between anchor and commitment.** A restart that fell between `init_batch` and `post_commitment` silently dropped every block executed above the anchored batch. One drill dropped 11,303 blocks. Restart now re-derives that window on a copy of state, and a per-block parent-output check guards the result.

**Two components each waiting on the other.** Production ran on the testnet for 21 hours without settling. A block re-stamped after an outage kept the producer's provisional context, so the internal shadow verifier disagreed inside a batch that was anchored but not yet committed. The tie-break refused to adopt the batch until the commitment was posted, and the commitment waited for the disagreement to clear. Neither could move. The node now re-executes an uncommitted top batch under its Solana context, and serial replay decides.

**Solana RPC is a dependency to budget.** Daemons polling Solana once a second, each re-reading a full window, reached 388 requests per minute and exhausted provider quotas. When every endpoint refused, settlement stopped and the chain stopped with it. Reads now come from a mirror fed by WebSocket and a caching, rate-limiting proxy, within a design budget of 70 requests per minute per provider per host.

**Program addresses are part of the interface.** Several accounts are derived from the settlement program's address. When that address changed, a client interface file updated the program and configuration but kept five derived accounts from the previous address, and every bridge deposit failed. Every fixed derived address is now re-derived, and a test asserts that each one is a native program or derives from the current program.

## 7. Data Availability

A challenger can dispute a batch only if it can read the batch. Two data availability members, on separate machines, store every block by height and root. Blocks are streamed to them only up to the durable height, compressed, with four in flight per member. Their acknowledgements are exact and form a contiguous watermark.

For batches of 16 MiB or less, availability does not depend on trusting the members. Anyone may pay to request a batch on Solana. The poster must then write the batch's bytes into Solana in chunks of 3,744 bytes before a deadline of two hours, extended in proportion to the bytes outstanding. A poster that misses the deadline is slashed 2 SOL and its commitment is rejected; two missed deadlines void the batch. The requester pays 0.002 SOL plus 0.0001 SOL per chunk into escrow, and is refunded the difference between that payment and the actual cost of the chunks. The limit of 16 MiB follows from the deadline. At roughly 2,500 bytes per second of Solana throughput over two hours, about 18 MB can be published, so any batch at or below the limit can always be produced on demand.

Batches larger than 16 MiB require a certificate signed by both members and checked on Solana with the ed25519 precompile [5]. For those batches availability rests on the committee. Members delete a height one hour after its batch confirms. An archive keeps closed segments and confirmed snapshots for fourteen days so new nodes can join.

We chose this split after reviewing an earlier design that posted only block headers to Solana and kept data elsewhere. A sequencer withholding data defeats every challenge [9], however sound the dispute game. Making every disputed batch publishable on the settlement layer closes that gap below the size limit and states it plainly above it.

An archive mirror of block headers requested only the first page of an index and so copied 1,000 rows. A restore from it would have lacked nearly every header. Mirrors now page until exhausted and verify the row count against the source.

## 8. Fault Proofs

The state transition function is compiled to rv32im, the 32-bit RISC-V base integer instruction set with multiply and divide [4]. It runs on a minimal execution environment of our own. The resulting program, the *guest*, is identified by the hash of its ELF file. That hash is fixed in the settlement program's configuration and embedded in the genesis. Because the node and the guest are built from one source, an independently written verifier cannot drift from the node.

The guest's memory is 4 GiB of 32-byte leaves under a depth-27 Merkle tree. A machine state is 177 bytes: the memory root, program counter, 32 registers, a halt flag, the output pointer and the step count. The guest reads outside data only through four calls:

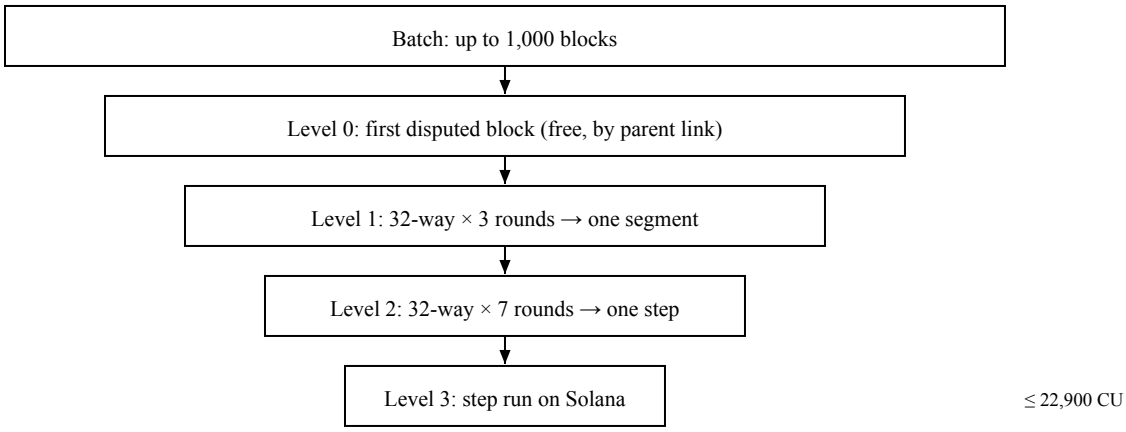
- a SHA-256 preimage;
- a word under a Merkle path;
- a tree node;
- a forced-inclusion entry from Solana, under the queue snapshot taken by `init_batch`.

Each call is checkable on Solana.

A dispute descends through four levels:

- **Level 0**: the challenger names the first block whose parent-output link it disputes. This is free, because blocks are linked by parent output.
- **Level 1**: the two parties bisect, 32 ways per round, over the segment outputs of that block (up to $2^{15}$ segments, three rounds).
- **Level 2**: they bisect over machine states within the disputed segment (up to $2^{35}$ steps, seven rounds).
- **Level 3**: `prove_step` executes the single disputed instruction on Solana from the agreed prior state and its Merkle witnesses.

A segment that exceeds $2^{33}$ steps or faults takes a pre-declared fallback output, so every segment has a defined result.




Each side runs on a chess clock of two hours. A dispute requires a bond of 1 SOL and a filing fee of 0.01 SOL. The loser is slashed 2 SOL; the winner receives 0.02 SOL and the rest is burned. A timeout needs both elapsed wall-clock time and elapsed Solana slots, so a pause in Solana cannot be turned into a forfeit. At most eight games are active at once. Further disputes wait in a first-in, first-out queue, can be activated

by anyone, and still block confirmation while waiting. Without the queue, a flood of disputes could make an honest proposer lose on time.

Measured results:

- `prove_step` costs at most 22,900 compute units, with witnesses of at most 2,996 bytes.
- One million fuzzed steps produced no disagreement between the on-chain stepper and the reference emulator.
- The emulator that computes checkpoints runs at about 28 million guest steps per second.
- An order-book trade, the heaviest transaction we run, takes under $2^{28}$ steps, and its guest output matches the native output bit for bit.
- Differential runs over 1,000,520 transactions in 9,472 blocks found no mismatch between native and guest execution.

Every on-chain step of the game was exercised end to end on a local Solana network against the real settlement program. In that run:

- A false commitment was caught in 27 moves, its poster was slashed and locked out, and the honest commitment was re-posted and confirmed.
- A dishonest challenger lost.
- Verifier processes killed mid-game resumed and finished.
- A defender cut off from the network for six minutes mid-game still won.

Those drills also found a defect that would have made an honest defender lose on time. The trace builder for Level 2 was constructed without the authenticated step schedule, so it answered every proof query with "unsupported".

Six rounds of design review closed more than twenty soundness gaps before deployment. Examples:

- tree depth taken from an untrusted register;
- an unbounded address search for forced-inclusion entries;
- a ruling instruction that required every challenger account in one transaction, which a swarm of accounts could have frozen.

Rulings now use one account per dispute.

### 9. Verifiers and the Challenge Window

Two verifiers re-execute the chain:

- **Designated verifier**, on a separate machine. It replays every batch with a serial engine, a code path independent of the producer's parallel engine, while it is within 60 seconds of the head. Further behind, it switches to the parallel engine and re-checks one batch in sixteen serially, chosen by a keyed hash the producer cannot predict. Disputes are filed only on a serial result. A disagreement between its two engines disables the parallel path and raises an alert.
- **Second verifier**, beside the producer. It replays in parallel and samples one segment per batch on the exact guest binary.

A challenge window is only meaningful if verifiers can replay within it. Each `init_batch` therefore declares cumulative work counters, and the guest refuses work beyond what was declared, so an understate-

ment is itself provable fraud. The settlement program converts declared work into verifier time and stretches the window when the backlog would exceed it:

$$t = \lceil \Delta work_{par} / (c \cdot g) \rceil + \lceil \Delta work_{ser} / g \rceil$$
$$backlog' = max(0,\ backlog - elapsed) + t$$
$$window = max(W,\ backlog' + 60\ s + 120\ s)$$

Here c is the configured number of verifier cores, g the configured verifier speed and W the base window, 540 seconds on the testnet. Settlement speed is thus honestly tied to verifier hardware. The cost of that honesty showed when the speed g was first set too low. Ordinary load then charged 2.5 seconds of verifier time per real second, and the window grew past 4,000 seconds while the verifier sat nearly idle. Raising g is timelocked by design, so the value is now set at initialization.

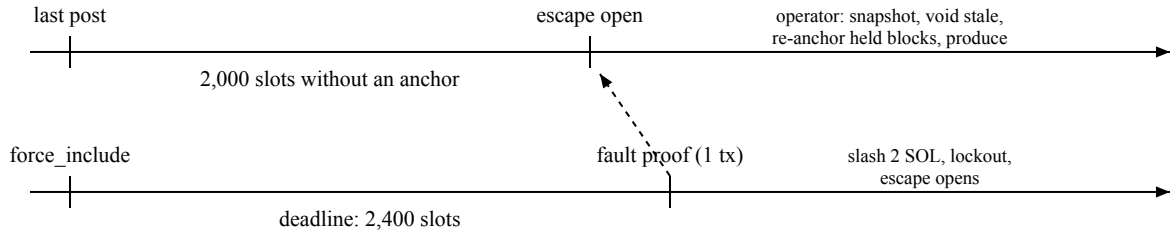
Restarting the designated verifier originally took two hours, because it re-read every commitment on Solana one request at a time. Reading them in groups of one hundred brought a walk of 19,900 batches to 75 seconds, and restart to first attestation to six minutes.

**10. Forced Inclusion and Escape**

A user who is being censored calls `force_include` on Solana. Every block reserves gas for forced entries, so at least two can always be drained per block. If a posted batch leaves an entry older than 2,400 Solana slots undrained, anyone can prove the fault in one Solana transaction. The poster is slashed 2 SOL and, if it is the designated sequencer, locked out. Deposits that fail to execute are parked and retried each block, so they never hold the queue. The fault deliberately ignores the kind of entry, because exempting deposits would let a sequencer stall the queue on one. Forced transactions use a blockhash derived from the genesis, so they cannot be replayed on another chain.

If no batch is anchored for 2,000 Solana slots, about eight minutes, or if a censorship fault has been proven, escape opens. Anyone with stake may then anchor and post. An escape operator, kept cold on a separate machine with stake deposited in advance, starts within 30 seconds of escape opening. It then:

1. loads the designated verifier's latest snapshot;
2. voids stale anchored batches;
3. re-anchors the departed sequencer's blocks held by the data availability members, unchanged;
4. produces new blocks, draining overdue forced entries first.




In drills on a local Solana network, both the sequencer and the primary data availability member were killed outright. The escape operator anchored within the escape delay plus 20 slots, forced withdrawals completed, and the original sequencer later rejoined as a follower. The forced-inclusion fault was proven within 2.5 seconds, and 215 of 215 acknowledged transactions landed. Two defects surfaced:

- Re-anchored blocks were applied under one provisional timestamp instead of each block's own, so only the first block linked.
- A replica chose between data availability copies by recency, so it served the dead sequencer's version of a height after takeover. Copies are now selected by batch root.

### 11. Finality and the Bridge

A transaction passes through five levels of finality:

1. acknowledged: durable in custody;
2. executed: in a block, typically within 87 milliseconds;
3. durable on both data availability members;
4. anchored and committed on Solana;
5. confirmed, after the challenge window.

Only the last is final against every failure, and the node reports it explicitly. Nothing below the confirmed watermark can be reorganised. To reorganise anything above it, the sequencer would have to withhold data through two availability deadlines, at the cost of two slashes and hours of publicly visible silence.

The bridge moves SOL from Solana into Verity as vSOL, the chain's gas asset. A deposit on Solana writes the vault transfer and a queue entry in one instruction, and the node reads the queue at Solana finality and credits the account. A withdrawal burns on Verity, enters the withdrawal accumulator and is paid on Solana once its batch confirms. An uncontested round trip from a Solana deposit to a Solana payout is bounded at about 568 seconds:

| Step | Seconds |
|---|---:|
| deposit read at Solana finality | 3 |
| credit and withdraw on Verity | 0.3 |
| batch close | 10 |
| anchor and commitment | 5 |
| challenge window | 540 |
| confirm and finalize | 10 |

The bridge interface signs Solana deposits with a plain sign request, so a deposit is never routed through Verity. At every financial step it re-checks the genesis identities of Verity and Solana in parallel. The swap is the second entry point. It routes between pools directly, or atomically in two hops through vSOL. Before signing, it re-reads every account it priced, and it gives each transaction a unique memo so two identical swaps cannot share a signature.

### 12. Node Reliability

The failures that stopped the node outright were not in consensus. They were in how threads, locks and processes interact under load. Every item below was found in operation:

- **A recursive read lock.** An RPC path holding an engine read guard called a helper that took the read lock again. The standard library's lock gives priority to waiting writers, so the second read queued behind the producer, which was itself waiting on the first. The node deadlocked until its watchdog aborted it. Rule: never re-lock under a held guard.

- **Blocking work on asynchronous workers.** RPC handlers that took engine locks ran on the asynchronous scheduler. Four waiting requests on a four-core host left no thread to accept connections, so the node looked down while healthy. Handlers that touch the engine now run on a blocking pool.
- **A stalled log consumer.** When the reader of standard output stopped reading, the logger's global mutex froze every thread in the process.
- **A memory ceiling below the working set.** The ceiling was 24 GB against a working set of about 25 GB. The node was throttled into a livelock of repeated re-stamping, and produced no blocks for 7.4 hours.
- **Index growth under the write lock.** A hash map resized under the index lock stalled the producer for 300 to 390 milliseconds at each doubling. It is now sized in advance.
- **Allocator arenas.** Per-thread allocator arenas grew to about 27 GB. Their count is now bounded.

Restart time is independent of history length, because logs are reopened by walking headers. A live restart under load returned RPC in 155 seconds with no acknowledged transaction lost: 485 transactions were landed again and 4,316 replayed. In a crash matrix under load, restarts reconfirmed in 24 to 96 seconds with nothing lost.

### 13. Performance

During development, Verity's execution path processed **78,168 transactions per second**. That was a burst of 102,400 signed transfers from 4,096 payers, executed into blocks in 1.31 seconds on a single ten-core workstation. The run used an earlier engine that did not compute a state root after every transaction. Twelve repeated bursts on that engine stayed above 50,000 transactions per second. With the network path removed, the same execution engine processed 127,428 transfers per second in process.

The fault-provable engine that runs today does strictly more work per transaction, because every transaction leaves an openable state root. On eight cores of a commodity server (AMD Ryzen 7 3800X, 64 GB), it measured:

| Run | Duration | Transactions | TPS |
|---|---:|---:|---:|
| Sustained transfers, replay-verified | 600 s | 15,896,205 | 26,494 |
| Accepted engine benchmark | 60 s | — | 29,600 |
| Small state, replay-verified | ≈ 20 s | 678,600 | 34,355 |
| Full node through RPC ingest | 120 s | 1,268,590 | 8,002 |
| Public testnet, live application traffic | 60 s peak | — | ≈ 870 |

In the 600-second run, every block was re-executed by the parallel engine and every fiftieth by the serial engine, with zero mismatches, zero fallbacks and zero rejected transactions. The slowest ten-second window still averaged 25,448 transactions per second. These engine figures measure execution and commitment. Signatures are verified before the timed window, in the pre-phase that admission performs in production, and that phase measured 47,600 verifications per second on four threads. The full-node figure includes every part of the live path: binary ingest, the 87 millisecond cadence, and an exact ledger check that 1,268,580 acknowledged transactions and ten probes all landed.

The gap between the engine and the full node is where current work is directed. Hot accounts set the ceiling for any single application, independent of the engine, as Section 4 describes.

## 14. Network Services

**RPC.** The node serves Solana-compatible JSON-RPC and WebSocket subscriptions, so existing tooling works unchanged. Verity adds the following:

- Exact decimal strings beside every 64-bit value, because JSON numbers lose precision above $2^{53}$.
- Gas estimates in simulation.
- Account views that page many accounts from one state root and slot.
- An extended context on every read, carrying the genesis hash, state root and output hash.

Unbounded program scans are refused above 4,096 accounts. Services instead keep registries fed by events, and page keys when they must scan.

The public edge is a gateway with no third-party dependencies. In order, it:

- checks keys;
- applies token buckets per client;
- refuses operator methods anywhere in a batch;
- writes the accepted genesis hash into every send, so a plain transaction can only ever execute on this network.

Client addresses are stored only as salted, truncated hashes. Rate tiers exist for third parties; first-party services use internal credentials and are never throttled. Developer access is provided through Keel, Verity's metered RPC platform. Its proxy is written in Rust, revokes keys within one second, and adds about 0.9 milliseconds at the median.

Product servers do not broadcast transactions directly. They first register the exact signed bytes with the gateway, which verifies the signatures, binds the blockhash and commits the record durably. Only then does it permit the broadcast. An ambiguous outcome is recovered by reading the record again or re-sending the identical bytes, never by signing again. This rule appears throughout the system, in the wallet, the SDK, the faucet and the oracle. An unknown outcome stays unresolved rather than risk a double payment.

**Indexers.** Indexers are pinned to the genesis hash and program identity, check both on every read, and refuse to serve if either differs. Gaps are recorded, never interpolated, and the database refuses writes that would lower a gap count or move a cursor backwards. The exchange program logs every fill as a fixed 160-byte record carrying a per-market sequence number. Full trade history is read from block logs instead of a ring buffer that overwrites itself. Logging costs 840 compute units per fill. In testing, 347 of 347 events matched field for field.

Indexers trail the head of the chain, because we found that a block read at the head could be returned under its final hash without its last transactions. Where receipt history has aged out, readers fetch from an archive. Every slot on Verity has a block, so an empty range is never read as an idle chain. The NFT indexer polls RPC, because the node exposes no streaming plugin. It identifies blocks by the sequencer's signature over the raw block, and reverts derived state when a recent block changes.

**NFT infrastructure.** Six programs provide the NFT layer: an asset standard, compression, minting, a market engine, AMM pools and a bridge vault. Each emits events through a self-invocation signed by its own event authority, so indexers cannot be fooled by forged logs. Sixteen backend services provide a DAS-compatible API, transaction building, proof serving and content screening. None of them holds user keys.

One finding from this work applies to every program on Verity. Upgrading a program in place is priced per changed 32-byte word, each with its own Merkle path, because the fault proof must be able to replay it. A 248 KB program therefore costs about 330 million gas to upgrade, more than one transaction may carry. Until that pricing changes, a new version of a large program is deployed at a new address.

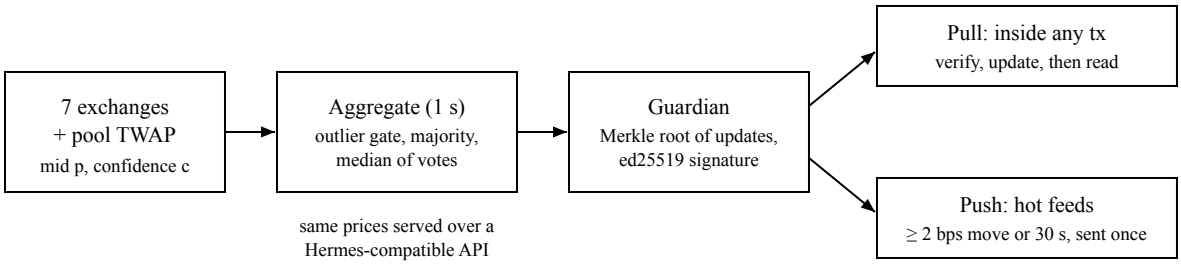
### 15. Oracle

Verity runs its own price oracle as a chain service. The service has seven publishers, each reading one exchange's order book over WebSocket: Binance, OKX, Bybit, Coinbase, Kraken, Hyperliquid and Lighter. An eighth publisher, a five-minute time-weighted price from on-chain pools, counts only above a liquidity floor. Each publisher contributes its mid price, with half the spread as its confidence. Quotes in USDT are converted at a live USDT/USD rate, and are excluded if that rate is stale or outside 0.9 to 1.1.

Once per second, the service aggregates each feed in the manner of [8]:

1. publishers more than 150 basis points from the plain median are set aside;
2. the remaining publishers must form a strict majority;
3. each remaining publisher casts three votes, at $p - c$, $p$ and $p + c$;
4. the price is the median of the votes;
5. the confidence is the larger distance from that median to the 25th or 75th percentile.

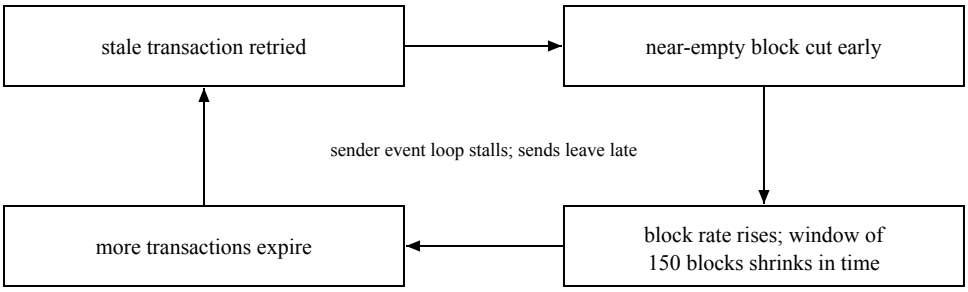
A frozen or wrongly quoted venue is outvoted and never becomes the price.




Each round's updates are hashed into a Merkle tree, and a guardian set signs the root with ed25519. The on-chain program verifies the threshold of guardian signatures through the native ed25519 precompile [5]. It requires every offset in that instruction to point inside the instruction itself, so the bytes checked are the bytes verified. It then checks each update's Merkle proof. Updates that are older, duplicated or below quorum are skipped rather than failed, so a transaction that pulls a price never fails because a push landed first. Any transaction can carry fresh signed prices and use them in the same instruction sequence. Frequently used feeds are also pushed, once per new round when the price moves 2 basis points or a 30-second heartbeat falls due. The same service provides verifiable randomness from a committed hash chain, and a sealed-bid auction for execution rights.

We built the oracle after a third-party price service changed its access terms, lapsed its trial, and left the exchange with no entitled feeds, while its metadata routes still answered as healthy. Licensed sources for equities, foreign exchange and metals were priced at about $10,000 a month, and their terms arguably forbid republication on a public chain. The live oracle serves 44 crypto feeds with about 3.75 million updates a day. The exchange refuses to trade on any market whose price is older than 200 seconds.

The oracle also produced the finding we consider most general. A keeper that retries stale transactions creates a self-sustaining wedge. Each retried transaction caused a nearly empty block to be cut early, which raised the block rate, which shortened the wall-clock life of a 150-block blockhash window, which expired more of the retries. The retry storm also stalled the sender's own event loop by several seconds, so its sends left too late even for the shortened window. The loop never recovered on its own, and restarting the process collapsed both halves at once, which made the restart look like the fix. Two changes ended it. The oracle now sends each price once and treats the next round as its retry. The chain seals blocks on a fixed deadline with a blockhash life counted in blocks (Section 3). Client retry storms amplify across a chain; we treat sending once per fresh state as a protocol-level rule for every service we run.




**16. Wallet, SDK and CLI**

**Wallet.** The Verity wallet is a browser extension that exposes its own provider and registers through the Wallet Standard. Keys are encrypted with AES-256-GCM under a PBKDF2 key of 310,000 iterations. Decrypted keys are held only in session storage, which pages cannot read, and the wallet locks after 30 minutes idle. Its parts are separated by duty:

- the page-side provider holds nothing secret;
- the background worker never signs;
- a site's request is signed only in a dedicated approval window.

The extension holds no RPC credential. Reads and submissions pass through the wallet's own server, which admits a fixed allow-list of methods and transaction shapes and rejects any result that names another genesis. The wallet sends only plain transfers itself. Program calls are signed for the application to submit, so the wallet's credential can never become an open relay.

Several of the wallet's defects compounded into one symptom: users retrying a swap two or three times.

- A definitive refusal never cleared the outbox.
- A preflight error was discarded and reported as an outage.
- The recovery path reported "unresolved" while a transaction was merely in flight.
- The browser destroyed the popup on focus loss, orphaning signed transactions mid-submission.

Each has its own fix, and the side panel is now the default surface. A separate defect half-activated wallets in real browsers but passed every test. Extension storage returns objects with sorted keys, so comparing read-back values as raw JSON failed. Comparisons are now canonical, and the test store sorts keys as the browser does.

**SDK and CLI.** The SDK provides network configuration that cannot be changed after creation, exact balances, transfers, submission tracking and a bridge client. It compares the network's genesis and version with the caller's own pins and never adopts them from the endpoint. Its confirm routine resends identical bytes while the blockhash is valid and never re-signs. On timeout it reports an unresolved outcome that carries the original bytes. The `verity` command-line tool manages keys, balances, transfers and the full upgradeable program lifecycle. It pins the genesis hash on first use and refuses to act against another. It uses exact integers throughout, and exits with a distinct code when an outcome is unconfirmed.

### 17. Applications

Two applications define what the chain is built for.

**perps.run** is a perpetual futures exchange whose order books, margin and liquidations run entirely on Verity:

- 170 markets on fully on-chain central limit order books;
- cross-margined portfolios with 1% initial and 0.5% maintenance margin;
- permissionless liquidation;
- a request-for-quote path where a maker's signed firm quote settles both sides in one instruction;
- prices from the oracle of Section 15, with trading refused on any market whose price is stale.

On the public testnet a fill is visible over RPC in 78 milliseconds at the median and 164 at the 99th percentile, under mixed load. Every order, cancel and crank is a transaction, which is the workload that motivated Verity.

**Parry** provides fully collateralised cover for perps.run traders. Each policy's full payout limit is locked in underwriter vaults at purchase. When a covered position is liquidated, perps.run writes the event to an on-chain record, and Parry's program reads that record and pays within seconds, with no claim to file. Cover against system faults is settled by signed attestations followed by a 30-minute challenge window. Cover against faulty liquidations is settled by a bonded assessor panel. Cover transactions never write the exchange's order-book or oracle accounts, so they add no latency to trading. That property was measured before deployment.

Both applications currently run on the public testnet with test collateral.

### 18. Trust Model and Limitations

On its own, a user can:

- enforce the validity of every state commitment through the dispute game;
- obtain the data of any batch up to 16 MiB through Solana;
- force transactions into blocks;
- exit through the escape hatch.

A user trusts the operators for:

- availability of batches above 16 MiB;
- the acknowledgement of transactions not yet anchored;
- the archive used by new nodes;
- program upgrades.

The application programs have not yet been independently audited. Settlement targets Solana devnet during the testnet.

Changes to the deployment will be published as addenda to this paper.

## 19. Conclusion

We have described an execution layer that runs Solana programs unmodified in 87 millisecond blocks without asking users to trust its operators for correctness. Every transaction leaves a state root that can be opened. Every disputed commitment reduces to one RISC-V instruction that Solana executes. Every batch small enough to be challenged can be forced onto Solana. Users can force inclusion and take over production through Solana alone. Building it showed that the hard problems were rarely the cryptography. They were coupling: between the block rate and blockhash lifetime, between execution and settlement time, between custody and finality, and between locks and the threads that wait on them. Removing each coupling produced a rule we now apply everywhere: execute independently of context, retire on durability, send once per fresh state, never re-sign, and treat unknown as unknown.

## References


[1] A. Yakovenko, "Solana: A new architecture for a high performance blockchain," 2018.

[2] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, E. W. Felten, "Arbitrum: Scalable, private smart contracts," In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[3] J. Teutsch, C. Reitwießner, "A scalable verification solution for blockchains," 2017.

[4] A. Waterman, K. Asanović, eds., "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA," RISC-V International.

[5] S. Josefsson, I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," RFC 8032, 2017.

[6] R. C. Merkle, "Protocols for public key cryptosystems," In *Proc. 1980 Symposium on Security and Privacy*, IEEE Computer Society, pages 122-133, April 1980.

[7] D. R. Morrison, "PATRICIA—Practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, 15(4):514-534, 1968.

[8] Pyth Data Association, "Pyth Network: A first-party financial oracle," 2023.

[9] M. Al-Bassam, A. Sonnino, V. Buterin, "Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities," 2018.