

# Hourglass: physically delivered forwards on $8\times H100$ node-weeks

A call-auction market for one standardized compute unit, prepaid into USDC escrow, backed by seller collateral, certified hour by hour and settled on Solana.

# Contents

|    |                                  |
|----|----------------------------------|
|    | Abstract                         |
| 1  | Introduction                     |
| 2  | The contract: HG-H100x8          |
| 3  | Participants and eligibility     |
| 4  | Series and the calendar          |
| 5  | The auction                      |
| 6  | Collateral                       |
| 7  | Claims and transfer              |
| 8  | Delivery                         |
| 9  | Evidence and the committee       |
| 10 | Faults, disputes and arbitration |
| 11 | Settlement                       |
| 12 | Fees                             |
| 13 | Credit                           |
| 14 | Program architecture             |
| 15 | Security design                  |
| 16 | Service infrastructure           |
| 17 | Interfaces                       |
| A  | Parameters                       |
| B  | Glossary                         |

# Abstract

Hourglass is a market for physically delivered forwards on one standardized unit of compute: the node-week, one NVIDIA HGX node with eight H100 SXM GPUs for one calendar week, 168 node-hours or 1,344 GPU-hours. Each delivery week and region is a series. Its node-weeks are sold at uniform-price call auctions held weekly from eight weeks to one week before delivery, quoted in USDC per GPU-hour at a tick of 0.01 USDC.

Buyers prepay the full price into escrow when they bid. Sellers post USDC collateral sized to the notional and the weeks remaining to delivery, and register accredited nodes. A filled bid becomes a claim: a Token-2022 token per series, frozen by default and transferable only between approved wallets until notice. At notice, 72 hours before delivery, claims become delivery tickets. Each ticket is assigned a node, its access key is handed over encrypted to the seller, an independent inspector accepts the node, and the buyer confirms access.

During the week, a Delivery Agent on the node signs a heartbeat every minute and the seller's Provider Agent signs BMC and DCGM telemetry. A committee of three keys, two of which must sign, aggregates the evidence into an hourly Merkle root and a bitmap of healthy tickets, committed on chain once per hour, and certifies each ticket after a 24-hour challenge window. Settlement then pays the seller from escrow for every certified hour, refunds the buyer for the rest, and pays the buyer liquidated damages from collateral for hours short beyond a 1% tolerance: the larger of 25% of the contract price and the reference spot price above it. A seller can pledge an obligation to a lender, who advances up to a set share of the escrowed notional and is repaid first from escrow releases, behind buyer refunds. Every fee the protocol keeps is reserved for buying back and burning \$HOURGLASS, the protocol's token: a fixed supply of 1,000,000,000, launched in full to the public on equal terms.

This paper specifies the contract, every deadline of a series, the auction's admission and clearing rules, collateral, claims, the delivery and evidence protocols, faults and arbitration, settlement and fees, credit, the three Solana programs and their accounts, the security mechanisms, and the service that runs the market. Every number in it is a protocol constant or a launch parameter of the deployed system; worked examples use example prices and say so.

# 1 Introduction

Hourglass is a market for forward contracts on compute that settle by delivery. The traded unit is one node-week: 168 consecutive hours of one NVIDIA HGX H100 SXM node with eight GPUs, delivered as bare metal in a named region during a fixed Monday-to-Monday UTC window. A buyer pays for the node-week in USDC before it starts. A seller posts USDC collateral against delivering it. Every hour of the week is certified healthy or unhealthy, and at the end of the week the program pays the seller for the certified hours, refunds the buyer for the rest, and pays the buyer liquidated damages from the collateral when the shortfall exceeds the contract's tolerance.

The service consists of three Solana programs, a public web application and API, and a worker that runs the indexer, the auction crank, the series scheduler and the committee aggregator. Participants act with their own browser wallets. Every transfer of value is an instruction of the market program, signed by the participant whose funds or rights it moves, or open to anyone when it only executes an outcome the program has already fixed.

Hourglass has one token of its own, \$HOURLASS: a fixed supply of 1,000,000,000, launched in full to the public on equal terms, which the protocol's fees exist to buy back and burn (\$12 Fees). Separately, each series issues claim tokens: Token-2022 position tokens with zero decimals, one token per node-week, which are the market's positions.

## 1.1 What Hourglass settles

Each delivery week  $W$  in each region is a separate series with its own accounts and its own claim mint. A series trades in weekly call auctions. Orders accumulate in the series' on-chain book during the week, and at the scheduled Thursday cutoff one permissionless `clear_auction` instruction fills them at a single uniform price. Auctions run every Thursday from eight weeks before  $W$  to one week before  $W$ .

A bid is prepaid. `place_order` moves the full notional of the bid, plus the 0.5% trade fee, from the buyer's wallet into the series funding vault before the order enters the book. When a fill is materialized, the filled notional moves into the series escrow vault, where it stays until settlement. Price improvement and unfilled quantity return to the buyer at the same step.

An ask is collateralized. A fresh ask reserves unobligated accredited capacity and a collateral budget from the seller's free collateral. When the ask fills, the program locks collateral equal to the filled notional times 20% plus 1.5 percentage points for every week remaining until delivery, measured from the scheduled auction. A filled ask becomes an `Obligation`: the seller's short position in the series, which is a program account and never a token.

A filled bid becomes claim tokens. Holders may sell them into later auctions of the same series, or transfer them to another eligible wallet, until the notice deadline at  $W - 72$  hours. By that deadline a holder converts a claim into a `DeliveryTicket` with `request_delivery`, which burns

one claim per ticket. Claims not converted by notice are waived: the corresponding escrow goes to the seller (to the seller's lender first, when the obligation is pledged) and the collateral behind them returns to the seller's free balance.

During the delivery week the buyer's Delivery Agent signs a heartbeat every minute and the seller's Provider Agent signs a BMC and DCGM observation every minute. The aggregator builds one Merkle root per series per hour, and a committee of three inspectors certifies, with two signatures, which tickets were healthy in that hour. The program records each hour with `commit_epoch`. After a 24-hour challenge window the committee signs one `DeliveryCertificate` per ticket and anyone may call `settle_ticket` to execute the payout.

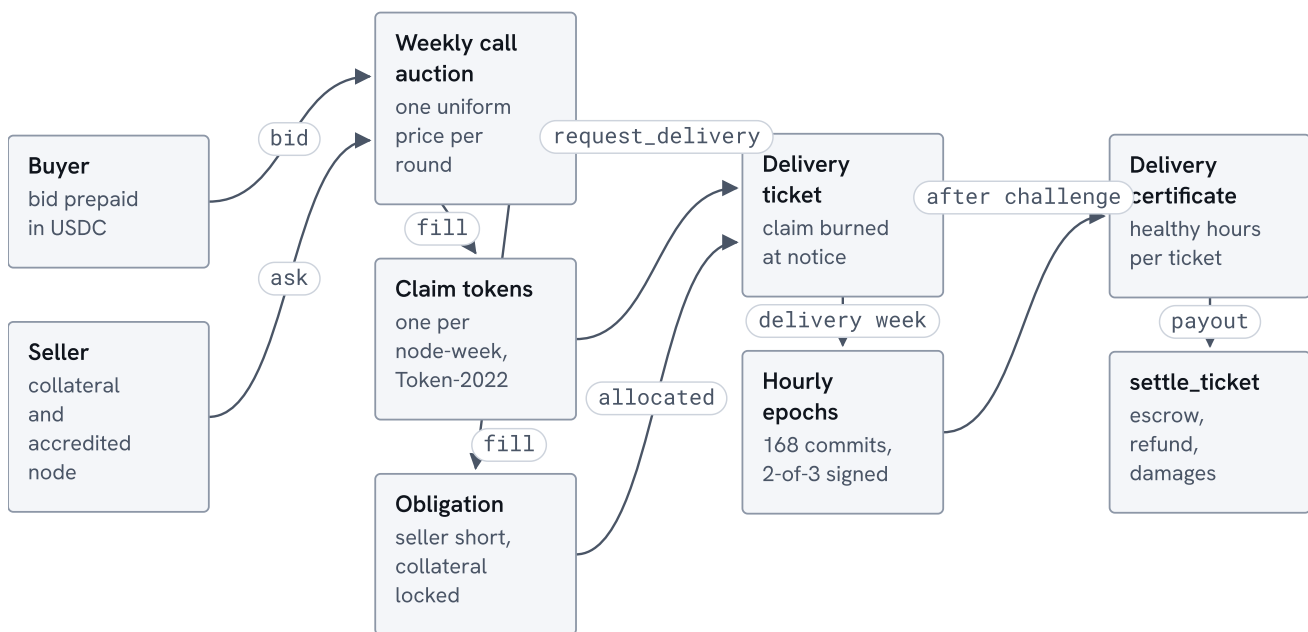


Figure 1 One node-week moves from a prepaid bid and a collateralized ask through the auction, claim conversion and hourly certification to settlement.

## 1.2 Why one grade and one-week windows

A forward trades only if one seller's delivery can stand in for another's. Hourglass defines the deliverable as a single HGX node because the topology inside one node is fixed by NVIDIA's reference design: eight GPUs on NVSwitch. Whether a node meets the grade is therefore a pass/fail question that an inspector answers once, at accreditation, and again at acceptance. Any accredited node from any seller in the region delivers the same contract, so claims of a series are fungible and each auction prints one price.

The single node is also the unit that inference and single-node fine-tuning buyers rent. Delivery across several nodes depends on fabric design, topology and scheduling, which differ between providers and cannot be assembled from several sellers. The grade's definition therefore stops at the node boundary: the back-end fabric and connectivity between nodes are not part of HG-H100x8.

The window is one week, starting Monday 00:00:00 UTC. Every node-week in a series covers exactly the same 168 hours, so a series needs one claim mint, one escrow vault and one hourly evidence ledger. A longer requirement is a strip of consecutive weekly series. The weekly window also sets the cadence of everything else: one auction a week, one notice deadline, one evidence ledger of 168 hourly roots.

### 1.3 Why Solana

Hourglass uses four specific capabilities of Solana.

**Permissioned claims without a custom token program.** Each series mints its claims with Token-2022 and six extensions. `DefaultAccountState(Frozen)` makes every new claim account frozen until the market program thaws it for an approved wallet; the participant whose transaction first needs the account pays for it, and the same transaction thaws it.

`TransferHook` routes every transfer through the separate `hourglass_hook` program, which re-reads the current eligibility record of both parties and refuses transfers after notice or while the series is paused. `Pausable` lets the administrator halt a series. `MetadataPointer` with `TokenMetadata` records the series, the specification hash, the delivery window and the region on the mint itself. `PermanentDelegate`, held by the series, lets retirement burn the claims left after their units were waived, and `MintCloseAuthority` then closes the empty mint and returns its rent.

**Committee signatures checked natively.** Inspectors sign epochs, certificates and arbitration awards off-chain with Ed25519 keys. The transaction carries a native Ed25519 precompile instruction immediately before the Hourglass instruction. The precompile cannot be called through CPI, so the program reads the instructions sysvar and checks that the preceding instruction verified two distinct keys from the current committee over exactly the bytes the program reconstructs. Each message binds the program, the `Config` account, a genesis-derived signature domain and the series.

**USDC as the settlement asset.** Escrow, collateral, fees, bonds and payouts are SPL USDC balances held in vaults whose authorities are program-derived accounts. `init_config` rejects a mint without six decimals. Every amount is a `u64` count of base units, every product uses checked `u128` intermediates, and every rounding direction is fixed in the code.

**Cheap, frequent commitments.** A series records 168 hourly commitments per delivery week. One `commit_epoch` carries the hour, a 32-byte Merkle root, a 64-byte bitmap covering up to 512 tickets and a 32-byte evidence hash; the two committee signatures share one message in one Ed25519 instruction, so the transaction fits in the 1,232-byte packet. Large state lives in zero-copy accounts grown in steps of 10,240 bytes: the 65,544-byte auction book, the 32,776-byte delivery allocation registry and the 7,520-byte evidence ledger.

The three programs divide the work. `hourglass` holds the contract specification, series, auctions, escrow, delivery, evidence, disputes and settlement. `hourglass_credit` holds lender pledges and is the only other program that can redirect a seller's escrow releases.

`hourglass_hook` holds the claim transfer rules; it is separate so that Token-2022 never calls back into the market program during a transfer.

## 1.4 What this paper covers

- §2 The contract: HG-H100x8 — the grade, the unit, healthy hours, tolerance, cure, damages and accreditation.
- §3 Participants and eligibility — the roles, what each signs and holds, and the on-chain eligibility record.
- §4 Series and the calendar — how series are listed and every deadline relative to the delivery week.
- §5 The auction — order admission, the clearing rule and how fills are allocated.
- §6 Collateral — how collateral is reserved, locked, allocated per node-week and released.
- §7 Claims and transfer — the claim mint, transfer rules, resale and conversion at notice.
- §8 Delivery — ticket allocation, access handoff, assignment, acceptance and access confirmation.
- §9 Evidence and the committee — minute evidence, hourly roots, epochs and certificates.
- §10 Faults, disputes and arbitration — bonds, cures, probes, holds and arbitration.
- §11 Settlement — the payout of each ticket and the rounding of every amount.
- §12 Fees — the trade, delivery and pledge fees and where they are charged.
- §13 Credit — pledges, lender advances and the repayment waterfall.
- §14 Program architecture — the three programs, their accounts and instructions.
- §15 Security design — the checks each instruction enforces.
- §16 Service infrastructure — the API, the worker, the indexer and the read model.
- §17 Interfaces — the public API and the transaction path for wallets.
- Appendix A lists every parameter; Appendix B is the glossary.

## 2 The contract: HG-H100x8

HG-H100x8 is the single contract Hourglass lists. It defines what a seller delivers, the unit in which it is traded and quoted, what counts as a delivered hour, how much shortfall is tolerated, how a failure can be cured, and what the seller owes when delivery falls short. This chapter gives the full specification at version 0.1.0.

### 2.1 The specification on chain

The human-readable specification is a fixed text. Its SHA-256 digest, computed over the UTF-8 bytes with line endings normalized to LF, is stored in the `ContractSpec` account at the address derived from the seeds `["spec", "HG-H100x8"]`. A changed specification is a new specification with a new digest; the text behind an existing digest never changes.

`create_spec`, signed by the administrator, writes the digest together with the numeric terms the program enforces. Each `Series` copies the digest and the economic terms from `ContractSpec` and `Config` when it is created, so a later change to configuration never alters a series that already trades.

```
pub struct ContractSpec {
    pub spec_hash: [u8; 32],
    pub gpu_count: u8,           // 8
    pub gpu_memory_gb: u16,      // 80
    pub ram_gb: u16,             // 1000
    pub nvme_tb: u16,            // 20
    pub ethernet_gbps: u16,      // 200
    pub tolerance_bps: u16,      // 9900
    pub damages_floor_bps: u16,  // 2500
    pub collateral_base_bps: u16, // 2000
    pub collateral_week_bps: u16, // 150
    pub cure_hours: u8,          // 4
    pub challenge_hours: u8,      // 24
    pub bump: u8,
}
```

### 2.2 The par grade

The deliverable is one node that meets every line of the par grade.

| Field         | Requirement              |
|---------------|--------------------------|
| GPU           | NVIDIA H100 SXM          |
| Architecture  | NVIDIA HGX with NVSwitch |
| GPUs per node | 8                        |
| GPU memory    | at least 80 GB per GPU   |

| Field             | Requirement                                           |
|-------------------|-------------------------------------------------------|
| Host memory       | at least 1 TB ECC RAM                                 |
| Host CPU          | dual-socket current-generation AMD EPYC or Intel Xeon |
| Local storage     | at least 20 TB NVMe                                   |
| Front-end network | at least 200 Gbps Ethernet                            |
| Public IP         | required                                              |
| Delivery mode     | bare metal                                            |
| Operating system  | stock Ubuntu 22.04 or 24.04                           |

The seller may deliver an HGX H200 node with eight 141 GB GPUs on NVSwitch in place of the par grade. The substitute is delivered at par: the contract price, the escrow and the damages terms are those of the H100 contract. The election is the seller's, made by the node it accredits and assigns.

The grade is defined on one node. A back-end fabric and connectivity between nodes are not part of it: a delivered node may carry InfiniBand or another fabric, and the contract states nothing about it. Two node-weeks bought in the same series are two independent deliveries that may come from different sellers and facilities.

## 2.3 The unit and the price

The unit is the node-week: 168 node-hours of one node. With eight GPUs per node that is  $8 \times 168 = 1,344$  GPU-hours. Claim tokens have zero decimals and one token is one node-week, so every quantity in the market is a whole number of node-weeks. One order covers 1 to 512 node-weeks, and a series issues at most 512.

Prices are quoted in USDC per GPU-hour with a tick of 0.01 USDC. USDC has six decimal places, so the tick is 10,000 base units, and the program rejects any price that is zero or not a multiple of it. The notional of an order or a fill is exact:

$$N = P \times 1344 \times q$$

Here P is the price in USDC base units per GPU-hour, q the number of node-weeks and N the notional in base units. No rounding occurs. At an example price of 2.20 USDC per GPU-hour, one node-week is  $2,200,000 \times 1,344 = 2,956,800,000$  base units, or 2,956.80 USDC. One tick moves the notional of a node-week by 13.44 USDC.

## 2.4 Regions and the delivery window

There are two region classes: NA (North America), encoded as 0, and EU (Europe), encoded as 1. The seller declares the facility; the contract guarantees the region and nothing narrower. A node's region is fixed in its `NodeRecord` at accreditation, and the program checks it against the

series region when capacity is reserved, when a node is assigned and when a replacement is booked.

The delivery window runs from Monday 00:00:00 UTC to the following Monday 00:00:00 UTC, 604,800 seconds. The program accepts a series only if its delivery start is a whole UTC day that falls on a Monday. Access is by SSH to the bare-metal host: the buyer's public key travels to the seller encrypted to the seller's X25519 key, and only its SHA-256 hash is recorded on the ticket (see §8 Delivery).

## 2.5 The healthy node-hour

An hour of delivery is either healthy or not. A node-hour is healthy when all four conditions hold for that hour:

1. All eight GPUs are visible.
2. The GPU UUIDs match the set recorded in the accredited `NodeRecord`.
3. No gating XID event occurs.
4. The node is reachable for at least 55 of the 60 minutes.

The gating XID list is a named configuration, `HEALTH_GATING_XIDS`, holding the NVIDIA XID codes that disqualify an hour. The service configures the codes 48, 64, 74, 79, 95, 119 and 120.

The evidence for each condition is signed minute by minute. The buyer's Delivery Agent signs a heartbeat that carries the hash of the visible UUID set, the count of gating XID events, GPU visibility and reachability. The seller's Provider Agent signs the same hardware fields together with BMC power state, link state and DCGM health. For each ticket and hour, the aggregator applies a fixed rule:

- The hour is healthy on buyer evidence when no buyer heartbeat shows a hardware conflict (a missing GPU, a different UUID set or a gating XID) and at least 55 heartbeats are clean and reachable.
- Otherwise the hour is healthy on seller evidence when no fault is open, every buyer heartbeat received is clean, at least 55 provider observations are clean, reachable, powered and linked, and no provider observation shows a hardware conflict or a DCGM failure.
- Otherwise the aggregator proposes the hour as unhealthy and flags the ticket for the fault process.

The committee reviews the proposal and signs the hour into the series' evidence ledger with `commit_epoch`; each healthy hour adds 1,000,000 micro-hours to the ticket's counter. Missing heartbeats never move money by themselves: buyer silence with no open fault and clean seller telemetry is certified healthy, and a dispute opens only through a bonded fault (see §9 Evidence and the committee and §10 Faults, disputes and arbitration).

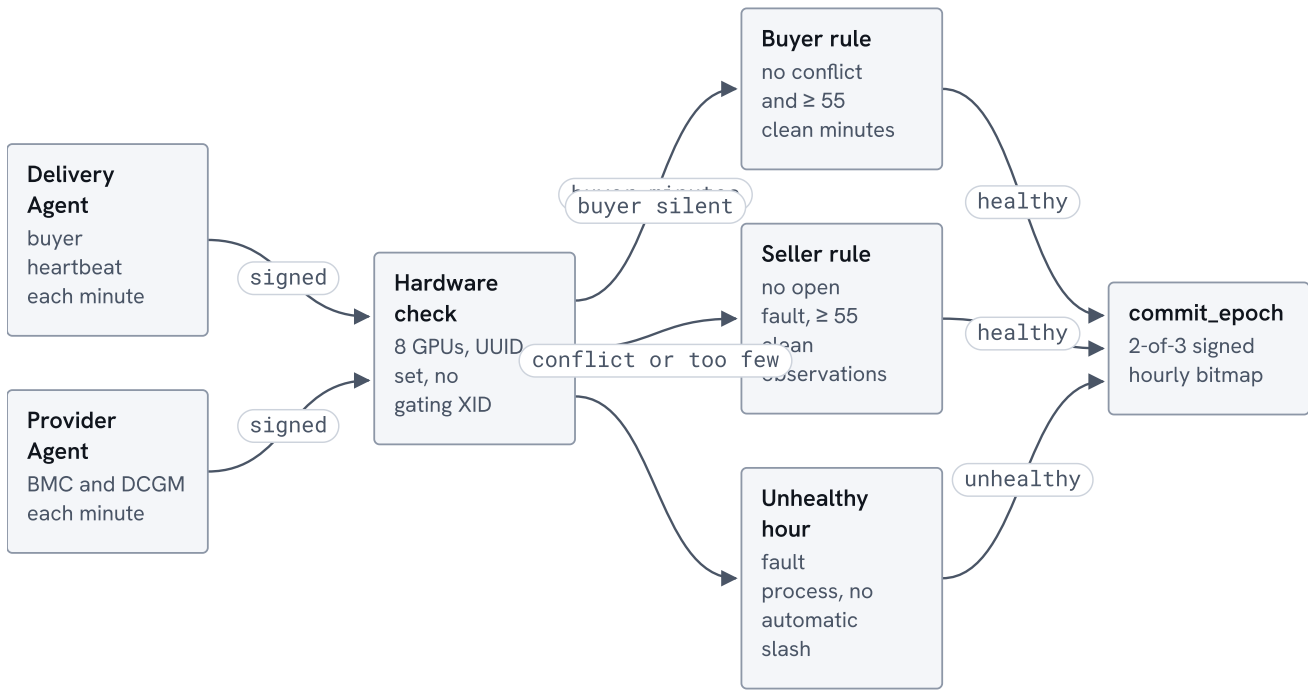


Figure 2 Each hour of each ticket is classified from signed minute evidence and committed to the evidence ledger.

## 2.6 Tolerance

The seller meets the contract when at least 99% of the 168 node-hours are healthy. The program stores the threshold as `tolerance_bps = 9900` and derives the allowance from it:

$$A = 168 \times (10000 - T) / 10000 = 168 \times 100 / 10000 = 1.68$$

T is `tolerance_bps` and A the allowance in node-hours per node-week. The program works in micro-hours, so the allowance is exactly 1,680,000 micro-hours, or 6,048 seconds. The allowance only exempts hours from damages. Every certified undelivered hour, including the hours inside the allowance, is refunded to the buyer pro rata.

## 2.7 Cure

When a buyer files a fault, the dispute records a cure deadline four hours after filing (`cure_hours = 4`). Before that deadline the seller may book a replacement node with `cure_fault`: a different accredited node in the series region, with accreditation valid through delivery end, a capacity reservation for the same week and no other booking for that week. The replacement may be of the par grade or the deliverable substitute. When an independent inspector accepts it with `accept_cure` before the same deadline, the ticket moves to the replacement node and the buyer's fault bond is returned.

An accepted replacement stops the damages clock: from acceptance on, the replacement's hours are certified like any other. It does not erase the past. Hours already certified unhealthy before the cure remain unhealthy, are refunded and count toward damages.

## 2.8 Liquidated damages

When undelivered hours exceed the allowance, the seller pays the buyer liquidated damages from the collateral allocated to the ticket. For each undelivered GPU-hour beyond the allowance the rate is the larger of 25% of the contract price and the amount by which the reference spot price exceeds the contract price. Per ticket:

$$\begin{aligned}H_e &= \max(0, 168 - H_h - H_x - A) \\r &= \max(P \times F, \max(0, S - P) \times 10000) \\D_u &= \lceil r \times 8 \times H_e / 10000 \rceil \\D &= \min(D_u, C)\end{aligned}$$

$H_h$  is the certified healthy node-hours of the ticket and  $H_x$  the undelivered hours excused as buyer-caused by the certificate;  $H_e$  is the damages-eligible node-hours.  $P$  is the contract price and  $S$  the reference spot price, both in base units per GPU-hour.  $F$  is the damages floor, `damages_floor_bps = 2500`, so  $r$  is a rate in base units per GPU-hour scaled by 10,000. The factor 8 converts node-hours to GPU-hours.  $D_u$  is rounded up to one base unit.  $C$  is the collateral allocated to the ticket, and  $D$ , the damages paid, never exceeds it.

The program evaluates hours in micro-hours, so  $H_e$  can be fractional. Hours excused as buyer-caused are only those the certificate proves: unhealthy hours that fall inside a compliance hold caused by the buyer or inside hours covered by a two-of-three arbitration award against the buyer. A ticket whose buyer never committed an access key is certified as a buyer default; it pays the seller the full escrow and charges no damages and no delivery fee.

The reference spot price comes from the certificate. The committee signs the price together with the hash of its source record, and the program rejects a certificate whose reference spot exceeds `Config.reference_spot_max`, a positive ceiling set at initialization. The service records the reference for each region and week as the median of three named providers' published prices; the configured ceiling is 20 USDC per GPU-hour.

## 2.9 Worked example

Take one node-week bought at an example clearing price of 2.20 USDC per GPU-hour in the auction four weeks before delivery, an example reference spot of 3.00 USDC per GPU-hour, and a week in which 165 of 168 hours are certified healthy. The auction is scheduled 584 hours before delivery start, 3.476191 weeks after rounding up to the micro-week, so the collateral locked for the node-week is  $2,956.80 \times (20\% + 1.5\% \times 3.476191)$ , rounded up: 745.536024 USDC.

| Quantity                              | Value                  |
|---------------------------------------|------------------------|
| Notional N                            | 2,956.800000 USDC      |
| Undelivered hours                     | 3                      |
| Allowance A                           | 1.68                   |
| Damages-eligible hours H <sub>e</sub> | 1.32                   |
| Seller gross [N × 165 / 168]          | 2,904.000000 USDC      |
| Pro-rata refund N – gross             | 52.800000 USDC         |
| Damages rate max(0.55, 0.80)          | 0.80 USDC per GPU-hour |
| Damages 0.80 × 8 × 1.32               | 8.448000 USDC          |
| Delivery fee [2.5% × gross]           | 72.600000 USDC         |
| Seller net                            | 2,831.400000 USDC      |
| Collateral released                   | 737.088024 USDC        |

The buyer receives 61.248000 USDC: the refund plus the damages. At the same price and spot, 167 healthy hours leave one undelivered hour inside the allowance: the refund is 17.600000 USDC and the damages are zero. At 166.319999 healthy hours the shortfall exceeds the allowance by one micro-hour, and the damages are  $0.80 \times 8 \times 0.000001 = 0.0000064$  USDC, rounded up to 7 base units. If no hour is delivered, the uncapped damages of 1,064.448000 USDC exceed the allocated collateral, and the buyer receives the full escrow back plus all 745.536024 USDC of collateral.

|                     |                                                                                       |               |
|---------------------|---------------------------------------------------------------------------------------|---------------|
| Escrow              |   | 2,956.80 USDC |
| Refund to buyer     |  | -52.80 USDC   |
| Delivery fee        |  | -72.60 USDC   |
| Seller net          |   | 2,831.40 USDC |
| Collateral          |    | 745.54 USDC   |
| Damages to buyer    |    | -8.45 USDC    |
| Collateral released |    | 737.09 USDC   |

Figure 3 The example ticket's escrow splits into refund, delivery fee and seller net, and its collateral into damages and release.

The rounding directions are fixed: the seller's escrow rounds down, the refund is the escrow residual, damages round up, collateral rounds up and the delivery fee rounds down, each to one USDC base unit. §11 Settlement gives the full payout order.

## 2.10 Accreditation

A node enters the market through accreditation. An inspector runs a burn-in of 16 to 24 hours:

`dcmi diag -r 4` with zero diagnostic failures, zero uncorrectable ECC errors and NVLink bandwidth within 5% of the reference figure. The inspector then signs `accredit_node`, which creates the `NodeRecord` at `["node", seller_profile, node_id]`.

The record holds the hashes of the eight GPU UUIDs, which must be nonzero, distinct and in ascending byte order, the SHA-256 digest of the eight hashes concatenated as the node's UUID-set hash, the hash of the burn-in report, the region, the accrediting inspector and an expiry 90 days after accreditation. The instruction also claims one marker account per GPU at `["gpu-uuid", uuid_hash]`. A marker belongs to one node until its expiry, so no GPU can sit in two active node records, and a node that shares seven of eight GPUs with another is rejected. The inspector must be a current committee member with the inspector role and a funded inspector bond (see §3 Participants and eligibility), and may not accredit its own seller.

`renew_node` records a new report and extends the expiry to 90 days from renewal, or keeps the current expiry if it is later; all eight markers must still belong to the node. Accreditation must cover the whole delivery week: reserving capacity, assigning the node and booking it as a replacement all require an expiry at or after delivery end.

Tier A is a recorded attribute of the node. An inspector who verifies NVIDIA confidential-computing attestation off-chain sets the `tier_a` flag in the record at accreditation or renewal. The contract does not require it, and a Tier A node delivers the same contract at the same price.

## 2.11 Contract parameters

| Parameter                 | Value               | Code name                                |
|---------------------------|---------------------|------------------------------------------|
| GPU-hours per node-week   | 1,344               | <code>GPU_HOURS_PER_NODE_WEEK</code>     |
| Price tick                | 10,000 base units   | <code>PRICE_TICK_USDC_MICROS</code>      |
| Healthy-hour reachability | 55 of 60 minutes    | <code>healthyHourReachableMinutes</code> |
| Delivery tolerance        | 9,900 bps           | <code>TOLERANCE_BPS</code>               |
| Damages floor             | 2,500 bps           | <code>DAMAGES_FLOOR_BPS</code>           |
| Collateral base           | 2,000 bps           | <code>COLLATERAL_BASE_BPS</code>         |
| Collateral per week       | 150 bps             | <code>COLLATERAL_PER_WEEK_BPS</code>     |
| Cure window               | 4 hours             | <code>CURE_HOURS</code>                  |
| Challenge window          | 24 hours            | <code>CHALLENGE_HOURS</code>             |
| Fault bond                | 500 bps of notional | <code>FAULT_BOND_BPS</code>              |
| Accreditation validity    | 90 days             | <code>ACCREDITATION_DAYS</code>          |
| Trade fee (buyer)         | 50 bps              | <code>TRADE_FEE_BPS</code>               |

| Parameter             | Value   | Code name          |
|-----------------------|---------|--------------------|
| Delivery fee (seller) | 250 bps | DELIVERY_FEE_BPS   |
| Node-weeks per series | 512     | MAX_SERIES_TICKETS |

The schedule parameters — listing window, auctions, notice, assignment and acceptance — are given in §4 Series and the calendar; the open-interest caps in §4 and Appendix A.

## 3 Participants and eligibility

Hourglass is a permissioned market. Every participant acts from a wallet, and every wallet that trades, delivers, inspects or lends has an on-chain eligibility record written by the operator's KYC authority. The market program and the claim transfer hook read that record at the moment of each action. This chapter defines the roles, what each role signs and must hold, which instructions it may call, and how a wallet becomes eligible and loses eligibility.

### 3.1 Roles

The program knows six roles, stored as bits of one byte in the wallet's `Eligibility` record. The operator is not a role bit: its two authorities are keys stored in `Config`.

| Role      | Bit | Must hold                              | Signs                                                 |
|-----------|-----|----------------------------------------|-------------------------------------------------------|
| Buyer     | 1   | USDC for notional, fee and fault bonds | bids, delivery requests, access key, faults           |
| Seller    | 2   | KYB, USDC collateral, accredited nodes | asks, assignment, cures, pledges                      |
| Trader    | 4   | USDC or claims                         | bids, claim sales, transfers                          |
| Inspector | 8   | committee seat, inspector bond         | accreditation, acceptance, probes, committee messages |
| Lender    | 16  | USDC for the advance and fee           | pledges                                               |
| Admin     | 32  | —                                      | label only; authority is <code>Config.admin</code>    |

A wallet may hold several roles. The program forbids one combination inside an auction round: a wallet cannot hold live orders on both economic sides of the same book, so a seller that also buys must do so in a different round or series.

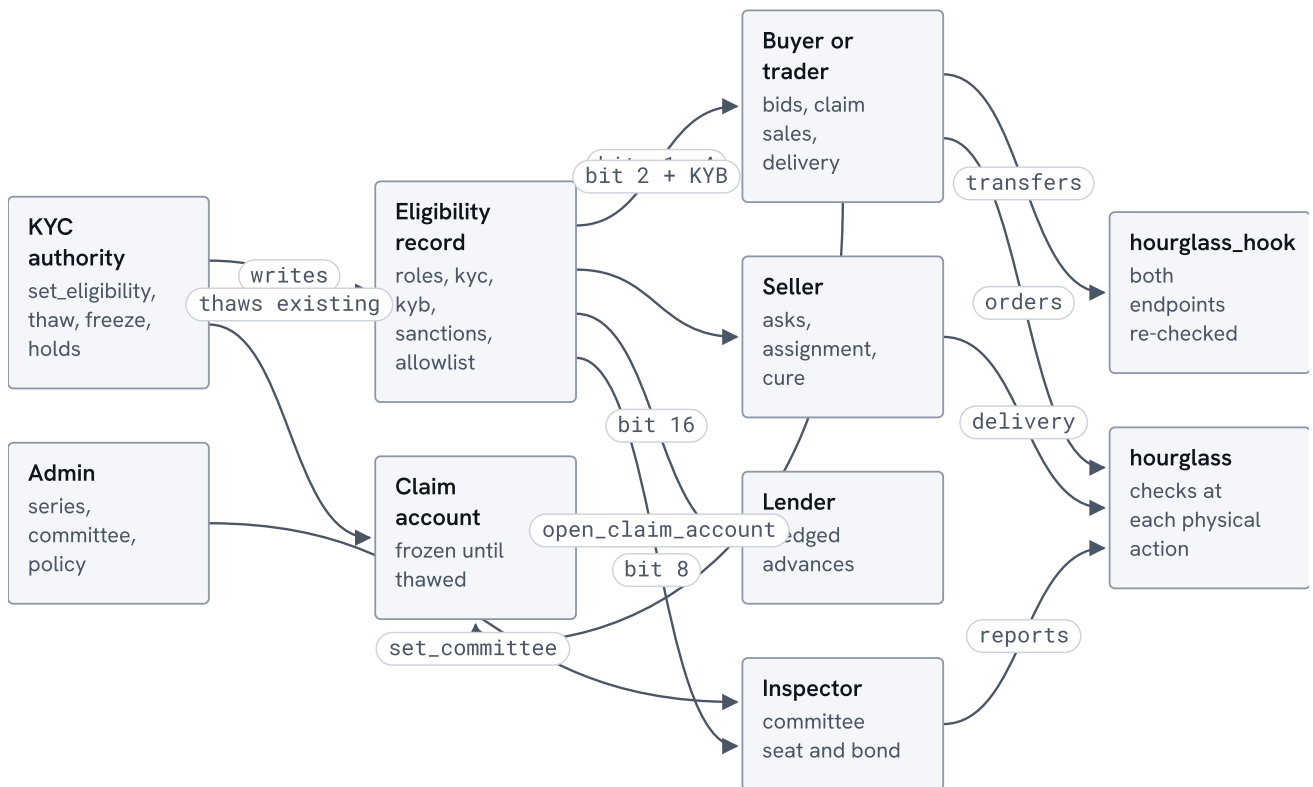


Figure 4 The KYC authority writes each wallet's eligibility record; the market program and the transfer hook read it on every gated action.

### 3.2 Buyer

A buyer acquires node-weeks and takes delivery. To bid, the buyer signs `place_order` with side 0; the program moves the full notional of the limit order plus the 0.5% trade fee from the buyer's USDC account into the series funding vault, and requires the buyer's canonical claim account for the series to exist and be thawed. When the account is missing or frozen, the transaction the service prepares opens or thaws it first with `open_claim_account`, and the buyer pays its rent. After the auction, the fill is materialized permissionlessly and the claims are delivered to that account with `collect_claims`.

Before notice the buyer converts claims with `request_delivery`, one ticket per transaction. On the ticket the buyer signs `set_access_hash` with the SHA-256 hash of the SSH public key, binds a Delivery Agent key with `bind_delivery_agent`, and later signs `confirm_access` or, within two hours of a passed acceptance, `object_access`. During delivery and the challenge window the buyer may sign `file_fault`. Each objection or fault escrows a bond of 5% of the ticket's notional, rounded up to one base unit: 147.84 USDC for a node-week at an example price of 2.20 USDC per GPU-hour.

### 3.3 Trader

A trader holds claims without intending to take delivery and exits before notice by selling into a later auction ( `place_order` with side 2, which escrows the claims in the series inventory) or by transferring the tokens to another eligible wallet. On chain the trader role carries exactly the long-side permissions of the buyer role: both may bid, sell claims, receive transfers and request delivery. Neither can open a short position. A fresh ask requires the seller role, KYB, collateral and accredited capacity; a claim sale can only deliver claims the wallet already holds.

### 3.4 Seller

A seller sells node-weeks forward and delivers them. Onboarding on chain starts with `register_seller`, which requires an approved record with the seller role and KYB and stores two values in the new `SellerProfile`: the hash of the KYB record and the seller's X25519 public key, to which buyers encrypt their SSH keys. The instruction also creates the seller's collateral vault at `["collateral", seller_profile]`.

The seller funds the vault with `deposit_collateral` and withdraws only free collateral with `withdraw_collateral`. Collateral moves through three balances in the profile: free, reserved by live asks, and locked behind filled obligations. For each accredited node and delivery week the seller signs `reserve_capacity`, which creates one `CapacitySlot` at `["capacity", node, delivery_start]` and adds one node-week to the seller's capacity in the series. A fresh ask ( `place_order` with side 1) needs that unobligated capacity and an explicit collateral budget at least equal to the collateral the full quantity would lock at its limit price at the time of placement.

By W - 48 hours the seller signs `assign_node` for each ticket allocated to it, binding the buyer's committed key hash and an accredited node; binds a Provider Agent key with `bind_provider_agent`; and, when a fault is filed, may sign `contest_fault` or `cure_fault`. A seller may also pledge an obligation to a lender, which both parties sign (see §13 Credit).

### 3.5 Lender

A lender advances USDC to a seller against the escrow of a filled obligation.

`pledge_obligation` in `hourglass_credit` requires two signatures in one transaction: the seller's, from an approved record with the seller role and KYB, and the lender's, from an approved record with the lender role. The advance may not exceed the configured percentage of the obligation's current escrow, and the lender pays the 0.5% pledge fee, rounded up, in addition to the principal. The lender is repaid from the seller's share of every escrow release, after the buyer's refund and the delivery fee; damages are paid from collateral and never reduce the lender's share.

### 3.6 Inspector

An inspector is a committee member that examines hardware. Its wallet needs an approved record with the inspector role, its key must be one of the three in `Config.committee`, and it must hold an inspector bond: `deposit_inspector_bond` moves USDC into a vault at `["inspector-bond-vault", inspector]` and records the balance in the `InspectorBond` account at `["inspector-bond", inspector]`. The bond must be at least `Config.inspector_bond_minimum`; the service configures 10,000 USDC.

With those three conditions met, the inspector signs `accredit_node` and `renew_node`, `record_acceptance` for a ticket's node, `accept_cure` for a replacement, and `resolve_fault` with a probe verdict. Independence is enforced per instruction: an inspector cannot accredit its own seller's nodes, cannot accept or probe a ticket on which it is the buyer or the seller, and cannot judge an objection to an acceptance it performed itself.

The bond stays in custody through the last challenge window that any of the inspector's signed work is exposed to, and the 72-hour arbitration handoff after it. Every accreditation, renewal, acceptance, cure acceptance and probe raises the bond's liability horizon, `InspectorBond.liability_until`, to the latest of its current value, the end of that work's challenge window plus the 72-hour arbitration handoff, and the current time plus 72 hours. For an accreditation or renewal the challenge window ends 24 hours after the node's accreditation expires; for an acceptance, cure acceptance or probe it ends at the series' delivery end plus its challenge hours.

To leave, the inspector signs `request_inspector_bond_withdrawal`. The bond is inactive from that instruction on: the inspector can no longer accredit, renew, accept, probe or accept a cure, and cannot add to the bond. The request records the release time as the later of the liability horizon and 72 hours after the request, and repeating it changes nothing. From the release time, `withdraw_inspector_bond`, signed by the same wallet, pays the full vault balance to a USDC account the inspector owns and closes the vault and the bond account, returning both rents to the inspector. Neither exit instruction reads eligibility or the committee, so an inspector whose role, seat or approval has been removed still recovers its bond.

### 3.7 Committee

The committee is the three inspector keys in `Config.committee`, with a threshold of two. It signs three kinds of message off-chain with Ed25519: the hourly epoch for each series, one certificate per ticket, and the record of an external arbitration award. The program accepts a message only when the native Ed25519 instruction preceding it verified two distinct current committee keys over the exact bytes; who submits the transaction does not matter, and the worker normally does. The administrator replaces the committee with `set_committee`, which requires three distinct nonzero keys.

### 3.8 Operator

The operator runs the service and holds two authorities. The **admin** ( `Config.admin` ) must be the program's upgrade authority when `init_config` runs. It creates the contract specification and series, sets the committee, pauses and resumes series, and sets auction and credit policy with `configure_auction` and `configure_credit` . Program upgrades are governed by a Squads multisig with a 48-hour timelock.

The **KYC authority** ( `Config.kyc_authority` ) writes eligibility records with `set_eligibility` , thaws existing claim accounts and freezes claim accounts with `thaw_claim_account` and `freeze_claim_account` , and records and releases compliance holds during delivery with `mark_compliance_hold` and `release_compliance_hold` . It signs nothing that moves USDC, and the service never uses it to pay for a participant's claim account.

### 3.9 The eligibility record

Each wallet has at most one record, at `["eligibility", wallet]` , 46 bytes:

```
pub struct Eligibility {
  pub wallet: Pubkey,
  pub roles: u8,
  pub kyc: bool,
  pub kyb: bool,
  pub sanctions_clear: bool,
  pub allowlisted: bool,
  pub bump: u8,
}
impl Eligibility {
  pub fn approved(&self) -> bool {
    self.kyc && self.sanctions_clear && self.allowlisted
  }
}
```

`set_eligibility(wallet, roles, kyc, kyb, sanctions_clear, allowlisted)` creates the record on first use, paid by the KYC authority, and overwrites every field on each later call. The roles byte must lie between 1 and 63. A wallet is approved when identity checks pass, sanctions screening is clear and the wallet is on the allowlist. Seller actions additionally require `kyb` .

The record is read at the instruction, not cached. The gated instructions and what each requires:

| Instruction                                | Signer | Eligibility required                                                  |
|--------------------------------------------|--------|-----------------------------------------------------------------------|
| <code>open_claim_account</code>            | payer  | payer and account owner approved; both buyer or trader; before notice |
| <code>place_order</code> bid or claim sale | owner  | approved; buyer or trader; thawed claim account                       |
| <code>place_order</code> fresh ask         | owner  | approved; seller; KYB                                                 |

| Instruction                                                                              | Signer                   | Eligibility required                          |
|------------------------------------------------------------------------------------------|--------------------------|-----------------------------------------------|
| <code>register_seller</code>                                                             | seller                   | approved; seller; KYB                         |
| <code>request_delivery</code> , <code>set_access_hash</code>                             | holder, buyer            | approved; buyer or trader                     |
| <code>assign_node</code> , <code>confirm_access</code> ,<br><code>finalize_access</code> | seller, buyer,<br>anyone | both parties qualify; no hold                 |
| <code>record_acceptance</code> , <code>accept_cure</code>                                | inspector                | inspector qualifies; both parties qualify     |
| <code>resolve_fault</code>                                                               | inspector                | inspector qualifies and is independent        |
| <code>collect_claims</code> to a wallet                                                  | anyone                   | owner approved; thawed account; before notice |
| Claim transfer (hook)                                                                    | holder                   | both ends approved buyer or trader            |
| <code>pledge_obligation</code>                                                           | seller and lender        | seller approved with KYB; lender approved     |

"Both parties qualify" means the ticket's buyer is approved with the buyer or trader role, its seller is approved with the seller role and KYB, and the ticket carries no compliance hold. An inspector qualifies when it is approved with the inspector role, sits on the committee and holds a sufficient bond.

### 3.10 Claim accounts are frozen by default

Every claim mint carries `DefaultAccountState(Frozen)` . A claim account created by anyone, for any wallet, starts frozen and can neither send nor receive. The program thaws a claim account in two instructions, and both act only on the owner's canonical associated token account for the series, only while the owner's record is approved and only before notice.

- `open_claim_account` is signed by the participant who pays for the account. The payer and the owner must both be approved with the buyer or trader role, and the series' mint must be initialized and the series not retiring. The instruction creates the canonical account if it does not exist, through the associated token account program's idempotent create at the payer's expense, and thaws it if it is frozen. The service places it before a bid, a claim sale or a claim collection that needs the account, paid by the signing participant, and before a claim transfer for either end, paid by the sender. The action `initialize_claim_account` opens one on its own, for the signer or for another approved buyer or trader.
- `thaw_claim_account` , signed by the KYC authority, thaws an existing canonical account other than the series inventory.

The Series account is the mint's freeze authority, so the program signs each thaw itself once its checks pass. Approval of a wallet creates no claim accounts; each one is opened by the first transaction on that series that needs it.

A thawed account still passes through the transfer hook on every transfer. `hourglass_hook` reads the current record of both the sender and the recipient and requires each to be approved with the buyer or trader role, both accounts to be canonical, the series to be

unpaused and the time to be before notice. The series inventory account is the only exception, because it is owned by the Series itself.

### 3.11 Revocation

The KYC authority revokes a wallet by calling `set_eligibility` again with a cleared flag or a removed role. The new record applies from the next instruction:

- New orders, claim collection to the wallet and transfers in either direction fail.
- `enforce_claim_freeze` , which anyone may call, freezes the wallet's canonical claim account on every series once its record is no longer approved; the worker calls it for every such account. The same instruction freezes every participant account at notice.
- Unrequested claims of a revoked holder cannot be converted. At notice they are waived like any other unconverted claim: their escrow goes to the seller and their collateral returns to the seller's free balance.
- On an existing ticket, assignment, acceptance and access confirmation fail. A buyer that never committed an access key is recorded in buyer default after W - 60 hours, which needs no approval.
- During the delivery week the KYC authority records a compliance hold on the ticket, with cause bit 1 for the buyer and 2 for the seller. Seconds held for a buyer-only cause are stored per delivery hour and excuse only hours that were actually unhealthy; time held for a seller cause, or for both, remains chargeable. Releasing the hold requires both parties to qualify again.

### 3.12 Credits that survive revocation

Eligibility gates physical delivery and claim movement. It never gates the return of a participant's own money. The following paths read no eligibility and stay open after revocation or pause:

- `cancel_order` on an open order, signed by its owner, and the permissionless `claim_fill` or `materialize_order` that settles a cleared order and refunds unused funding.
- `collect_cash` , `collect_obligation_cash` and `collect_ticket_cash` , which pay USDC credits recorded when a payee's canonical account could not receive them. Payment to the canonical account is permissionless; payment to another account of the owner requires the owner's signature.
- `withdraw_collateral` of free collateral, `settle_ticket` and `waive_delivery` , and `collect_lender_cash` for a lender's repayments.

An order displaced from the book by a better price releases its reservation in the same transaction: a bid's deposit is paid to the owner's canonical account or recorded as a credit, an ask's collateral and capacity return to the seller's free balances, and a sale's escrowed claims become a credit of the owner. The displacement needs no signature from the displaced owner, and the owner's revocation cannot block it.

### 3.13 Onboarding

Onboarding is an operator review followed by on-chain approval. A wallet signs in to the application by signing a single-use challenge; signing in grants no role. The signed-in wallet applies with `participant.apply`, naming the roles it wants, the company, a contact address and, for a seller, its region. The application waits for review.

The operator checks identity, KYB for sellers and sanctions off-platform, then approves with `operator.approve`, which carries the wallet, the granted roles and the four flags. Approval is an on-chain transaction signed by the KYC authority: `set_eligibility` for the wallet, followed by `thaw_claim_account` for each frozen claim account the wallet already holds on a series before its notice deadline. From that transaction on, the program treats the wallet as eligible. Approval opens no new claim accounts, and the worker's scheduler loop repeats the thaw for approved wallets whose existing accounts are still frozen. A revocation follows the same path with `operator.revoke`.

A seller continues on chain: it signs `register_seller` and `deposit_collateral`, submits each node's label, region, eight GPU UUIDs and health endpoint with `nodes.submit`, and reserves capacity once an inspector has accredited the node. An inspector deposits its bond. A lender needs no further setup before its first pledge.

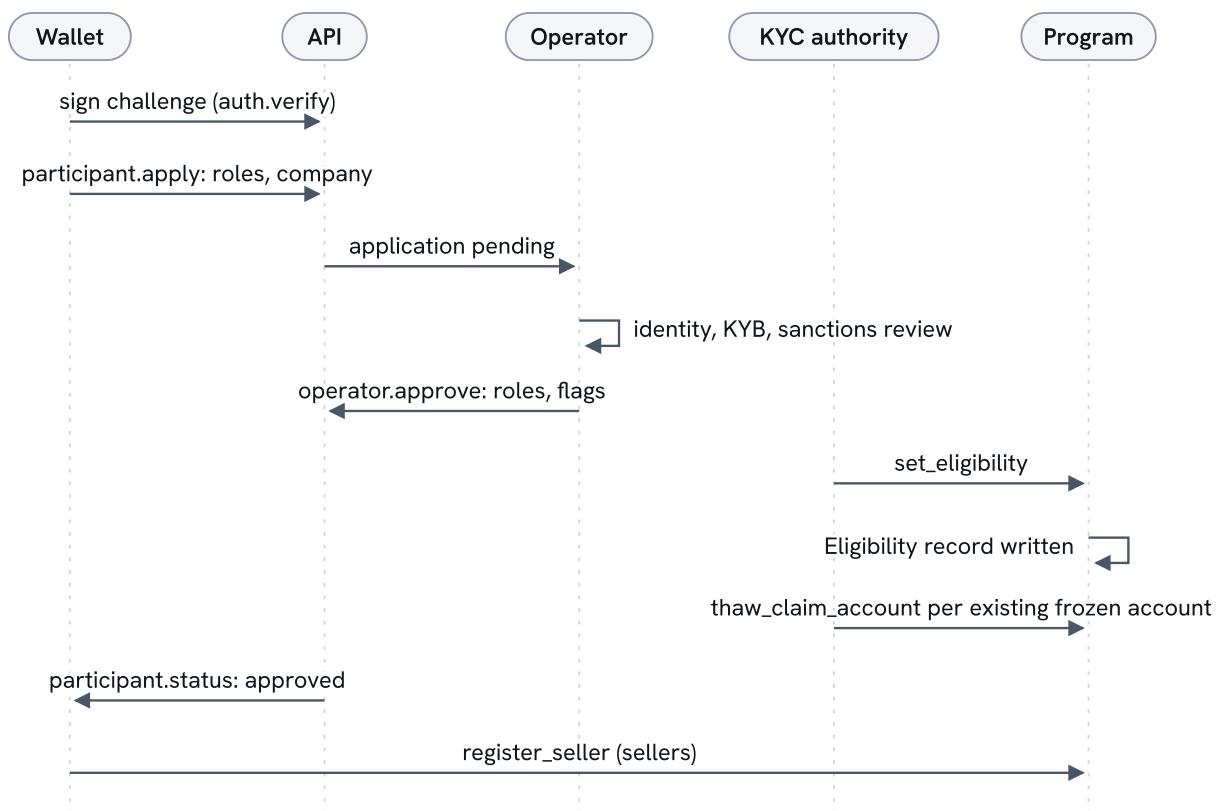


Figure 5 A wallet applies in the application, the operator reviews it off-platform, and the KYC authority makes the approval effective on chain.

### 3.14 Authority rotation

Both operator authorities rotate in two steps. The admin proposes a successor with `propose_admin` or `propose_kyc_authority` ; the proposal must be a nonzero key and is stored as `pending_admin` or `pending_kyc_authority` . The rotation completes only when the exact proposed key signs `accept_admin` or `accept_kyc_authority` ; the pending field is then cleared, so an acceptance cannot be replayed. The service exposes the pair as `propose_authority` and `accept_authority` .

The admin proposes the KYC authority's successor, so the loss of the current KYC key does not prevent its replacement. Until the successor accepts, the current KYC authority keeps signing eligibility writes, thaws, freezes and holds.

## 4 Series and the calendar

A series is one region and one delivery week. It is the unit in which Hourglass lists, trades, delivers and settles: each series has its own auction book, its own claim mint, its own escrow and its own evidence ledger, and every deadline that applies to it is a fixed offset from its delivery start `W`. This chapter defines the series, how it is listed, the complete calendar relative to `W`, the states a series passes through and the open-interest caps that bound it.

### 4.1 What a series is

A series is identified by the contract specification, the region and the delivery start. Its

`Series` account lives at the address derived from

```
["series", spec, region_u8, delivery_start_i64_le]
```

where `spec` is the `ContractSpec` address, the region is 0 for NA or 1 for EU, and `delivery_start` is `W` as Unix seconds in little-endian order. `W` must be a Monday at 00:00:00 UTC. The program derives every other instant from `W`: `delivery_end` is `W + 604,800` seconds and `notice_at` is `W - 259,200` seconds.

The 466-byte `Series` account holds the schedule ( `delivery_start` , `delivery_end` , `notice_at` , `next_auction_at` , `round` ), open interest and its cap, the addresses of the claim mint and the two USDC vaults, the counters of issued, requested, waived and settled node-weeks, and the flags `paused` , `mint_initialized` , `allocation_sealed` and `cap_released` . At creation it also snapshots the terms it will settle under: the specification hash, the collateral base and weekly rate, the tolerance, the damages floor, the trade and delivery fees, the cure window and the challenge window. A configuration change after creation never reaches an existing series. The last fields serve retirement: the keys that paid for the series' accounts, the `retiring` flag and the audit cursors (see Retirement below).

Each series owns a fixed set of accounts, all derived from the `Series` address.

|                                                                                                                 |                 |
|-----------------------------------------------------------------------------------------------------------------|-----------------|
| <b>Series</b><br>["series", spec, region, W]<br>schedule, caps, snapshots, counters                             | 466 B           |
| <b>Auction book</b><br>["book", series]<br>1,024 order slots of 64 B<br>→ Series series                         | 65,544 B        |
| <b>AuctionResult</b><br>["auction", series, round]<br>price, volume, scheduled cutoff<br>→ Series series, round | 124 B           |
| <b>Delivery registry</b><br>["delivery-alloc", series]<br>512 issued node-week units<br>→ Series series         | 32,776 B        |
| <b>Funding vault</b><br>["funding", series]<br>bid deposits, cash credits<br>→ Series authority                 | token account   |
| <b>Escrow vault</b><br>["escrow", series]<br>filled notional<br>→ Series authority                              | token account   |
| <b>Claim mint</b><br>["mint", series]<br>claims, 0 decimals<br>→ Series mint, freeze, pause authority           | Token-2022 mint |

|                                                                                                                |               |
|----------------------------------------------------------------------------------------------------------------|---------------|
| <b>Claim inventory</b><br><code>["claims", series]</code><br>claims for sale or pending<br>→ Series owner      | token account |
| <b>EpochLedger</b><br><code>["epoch", series]</code><br>168 roots, 512 counters<br>→ Series series             | 7,520 B       |
| <b>Health archive</b><br><code>["epoch-health", series]</code><br>168 hourly ticket bitmaps<br>→ Series series | 10,800 B      |

Figure 6 The accounts of one series and the series account they are derived from.

The claim mint carries its identity in its own metadata. The name is "Hourglass" followed by the region and W in Unix seconds, the symbol is `HG-H100x8`, the URI is `hourglass://series/` followed by the series address, and five additional fields record the series address, the specification hash, `delivery_start`, `delivery_end` and the region. For the example week used below the name is `Hourglass NA 1797206400`.

## 4.2 Listing

The administrator lists a series with `create_series(region, delivery_start, claim_mint, notional_cap)`. The program accepts it only when W is a Monday at 00:00 UTC, the region is NA or EU, W lies between two and eight weeks ahead of the current time inclusive, and `claim_mint` is the address derived from `["mint", series]`. The notional cap must be positive, no larger than the per-series cap in `Config`, and small enough that the sum of caps reserved by all unreleased series stays within the total cap. The instruction creates the series, its funding and escrow vaults, and the first 10,240 bytes of the auction book and the delivery registry, and records the admin as the series' rent payer.

A Solana instruction can grow an account by at most 10,240 bytes, so the remaining setup is a sequence of instructions. The two growth steps accept only the series' rent payer; the others are permissionless and record the key that paid for what they create:

1. `grow_book`, six times, until the book reaches 65,544 bytes.
2. `grow_delivery_registry`, three times, until the registry reaches 32,776 bytes.

3. `init_claim_mint` , which creates the Token-2022 mint with its extensions and metadata, creates and thaws the series inventory account, and registers the transfer hook's extra accounts.
4. `init_epoch_ledger` , then `init_epoch_health` and one `grow_epoch_health` call, which bring the hourly archive from 10,240 to 10,800 bytes.

The program enforces the order of these steps where it matters. `place_order` fails until the claim mint is initialized and the book is fully grown, a fresh fill cannot be materialized until the registry is complete, and `commit_epoch` needs both evidence accounts. The worker's series scheduler signs `create_series` , the growth steps and `init_claim_mint` with the admin key for each region and each upcoming Monday on this calendar. The committee aggregator opens the epoch ledger and the health archive with the worker key once a sealed series with requested tickets reaches the end of its first delivery hour; the service opens neither for a series without requested tickets.

### 4.3 The auction schedule

Auctions take place every Thursday of the eight weeks before delivery, at one UTC time of day. The time is `Config.auction_seconds` , fixed at `init_config` . The program requires

$$s + g_{\max} + c_{\min} \leq 86400$$

where  $s$  is the auction time in seconds after 00:00 UTC,  $g_{\max} = 14,400$  seconds is the longest permitted failed-auction grace and  $c_{\min} = 14,400$  seconds is the minimum time left for claim conversion before notice. The latest permitted auction time is therefore 16:00:00 UTC, and the service configures 16:00 UTC.

The scheduled auctions of a series are

$$t_k = -168 \times k + 72 + s / 3600, \quad k = 8, 7, \dots, 1$$

where  $t_k$  is the cutoff of the auction  $k$  weeks before delivery, in hours relative to  $W$ , and  $s$  is the auction time in seconds: the Thursday of the  $k$ -th week before delivery, three days after that week's Monday. At 16:00 UTC the first auction is at  $W - 1,256$  hours and the last at  $W - 80$  hours, eight hours before notice.

`create_series` sets `next_auction_at` to the first scheduled auction after the current time; a series listed two weeks ahead therefore has two auctions and one listed eight weeks ahead has eight. Orders are accepted while `now < next_auction_at` and `next_auction_at < notice_at` . `clear_auction` for the current round is accepted from `next_auction_at` until notice. It computes collateral from the scheduled cutoff, not from the time the crank arrives, so a late clear prices collateral exactly as a punctual one. A successful clear writes the round's `AuctionResult` , increments `round` and moves `next_auction_at` one week later. After the last clear, `next_auction_at` is  $W + 88$  hours, after notice, and order entry is closed.

Before the cutoff an owner may cancel an open order at any time. The failed-auction grace covers a round whose clear has not arrived. From the cutoff until the cutoff plus the grace, open orders of the round cannot be cancelled, because the round is waiting for its clear. After the grace an owner may cancel an open order that was not cleared, and from notice on cancellation is always available. The grace is snapshotted into the series when the round receives its first order, is at most 14,400 seconds, and is configured by the service at 3,600 seconds. A round that cleared with zero volume releases its orders immediately.

When trading ends, anyone may call `seal_delivery`. It succeeds once the final scheduled auction has cleared, or once the grace after the last scheduled auction has passed, and only when every fresh fill has been materialized and minted. Sealing sorts the registry of issued node-weeks into its canonical order and closes order entry and clearing for good. Delivery requests need a sealed series.

#### 4.4 Deadlines relative to W

Every deadline is a UTC instant at a fixed offset from W. Daylight-saving time plays no part.

| Event                  | Offset from W    | Rule                                                            |
|------------------------|------------------|-----------------------------------------------------------------|
| Listing opens          | -1,344 h         | <code>create_series</code> from W - 8 weeks                     |
| First auction          | -1,256 h         | Thursday of week W - 8 at 16:00                                 |
| Listing closes         | -336 h           | <code>create_series</code> until W - 2 weeks                    |
| Final auction          | -80 h            | Thursday of week W - 1 at 16:00                                 |
| Notice                 | -72 h            | conversion until, inclusive; waiver strictly after              |
| Access commitment      | -60 h            | <code>set_access_hash</code> until, inclusive                   |
| Assignment             | -48 h            | <code>assign_node</code> until, inclusive                       |
| Acceptance opens       | -24 h            | <code>record_acceptance</code> from                             |
| Access confirmation    | acceptance + 2 h | <code>finalize_access</code> from                               |
| Delivery               | 0 to +168 h      | hour h committed from W + h + 1                                 |
| Late acceptance closes | +4 h             | <code>record_acceptance</code> until, inclusive                 |
| Fault filing closes    | +192 h           | <code>file_fault</code> in [W, W + 192 h)                       |
| Certification opens    | +192 h           | <code>post_certificate</code> , then <code>settle_ticket</code> |
| Cure deadline          | filing + 4 h     | <code>cure_fault</code> , <code>accept_cure</code> until        |
| Arbitration            | filing + 72 h    | <code>advance_arbitration</code> from                           |
| Manual certificate     | +192 h + timeout | incomplete evidence ledger only                                 |

The notice instant belongs to conversion: `request_delivery` is accepted while `now ≤ notice_at`, and `waive_delivery` only while `now > notice_at`, so no instant permits both. Claim transfers, claim collection, claim-account opening and thaws stop at notice as well. With auctions at 16:00 UTC there are eight hours between the final cutoff and notice; even after the longest failed-auction grace, four remain.

The buyer commits the access-key hash by  $W - 60$  hours, twelve hours before the seller's assignment deadline at  $W - 48$  hours. Once  $W - 60$  hours has passed, anyone may mark a ticket with no committed hash, no node and no hold as a buyer default. Acceptance runs from  $W - 24$  hours through  $W$  plus the series' cure window of four hours. Every passed acceptance starts a two-hour window. Within it the buyer may object; the buyer may confirm access at any time; once the window has ended without an objection, anyone may finalize access as confirmed.

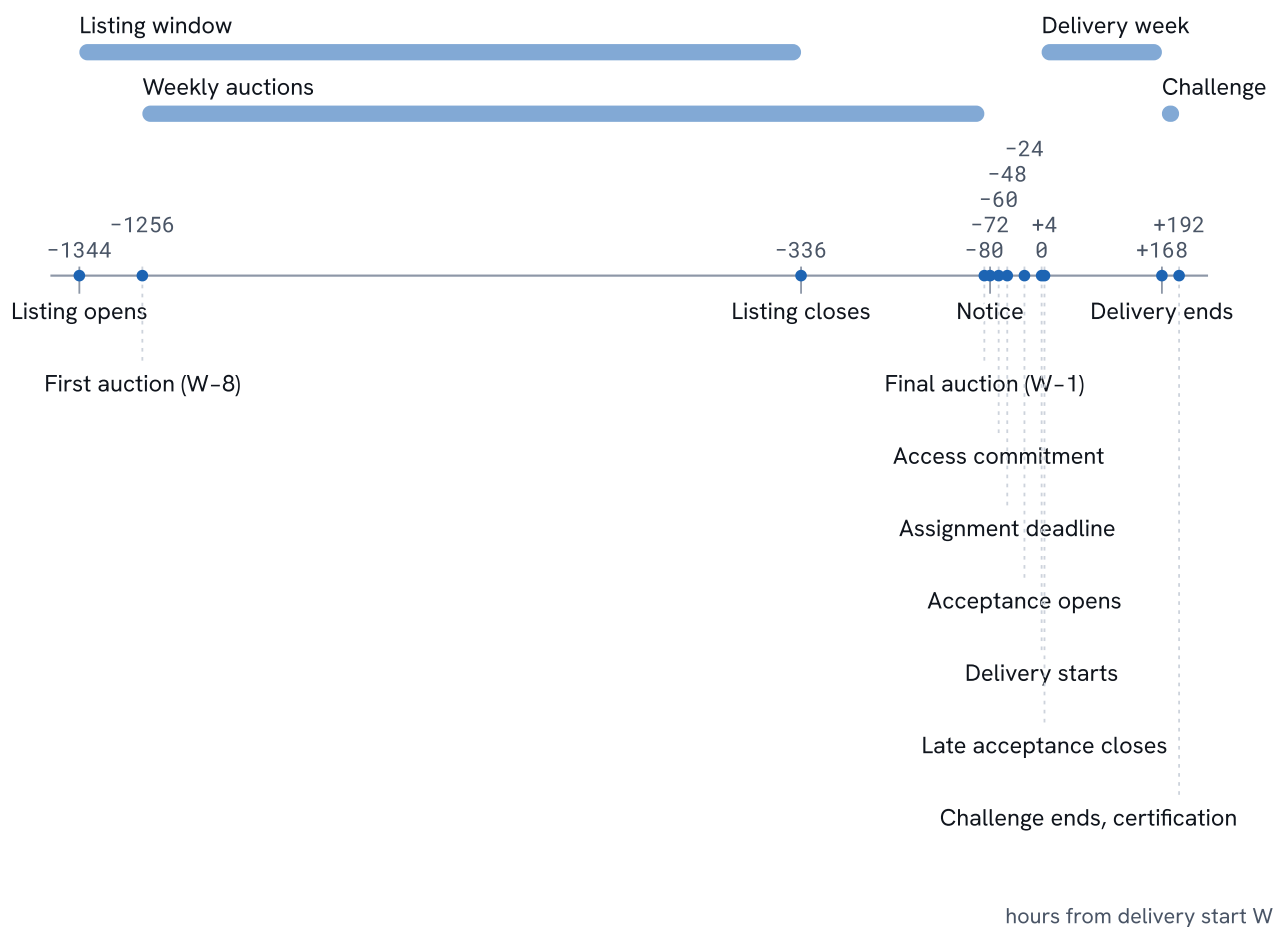


Figure 7 The life of one series from listing to the end of the challenge window, in hours from delivery start  $W$ , with auctions at 16:00 UTC.

## 4.5 Delivery, challenge and arbitration

The delivery week is the 168 hours from  $W$ . The committee commits hour  $h$ , for  $h$  from 0 to 167, with `commit_epoch` once the hour has closed, that is from  $W + (h + 1)$  hours, and strictly in order: an hour cannot be committed twice, and hour  $h + 1$  cannot precede hour  $h$ .

The challenge window is the 24 hours after delivery end. A buyer may file a fault from W until W + 192 hours, exclusive, so evidence from the last delivery hour can still be challenged. Each fault fixes two deadlines when it is filed: the cure deadline, four hours later, and the arbitration instant, 72 hours later. A fault still open at its arbitration instant moves to external arbitration with `advance_arbitration`, and its award is recorded with two committee signatures. A fault filed in the last moment of the challenge window reaches arbitration at W + 264 hours.

Certification opens at W + 192 hours. `post_certificate` writes one certificate per ticket and is refused while the ticket has an open dispute. The normal certificate is backed by the completed evidence ledger of 168 committed hours. When the ledger is incomplete, the committee may post a manually attested certificate once the challenge window and a further `Config.manual_certificate_timeout_seconds` have passed; the service configures 86,400 seconds, which places the manual fallback at W + 216 hours. Once certified, `settle_ticket` may be called by anyone, and repeating it changes nothing.

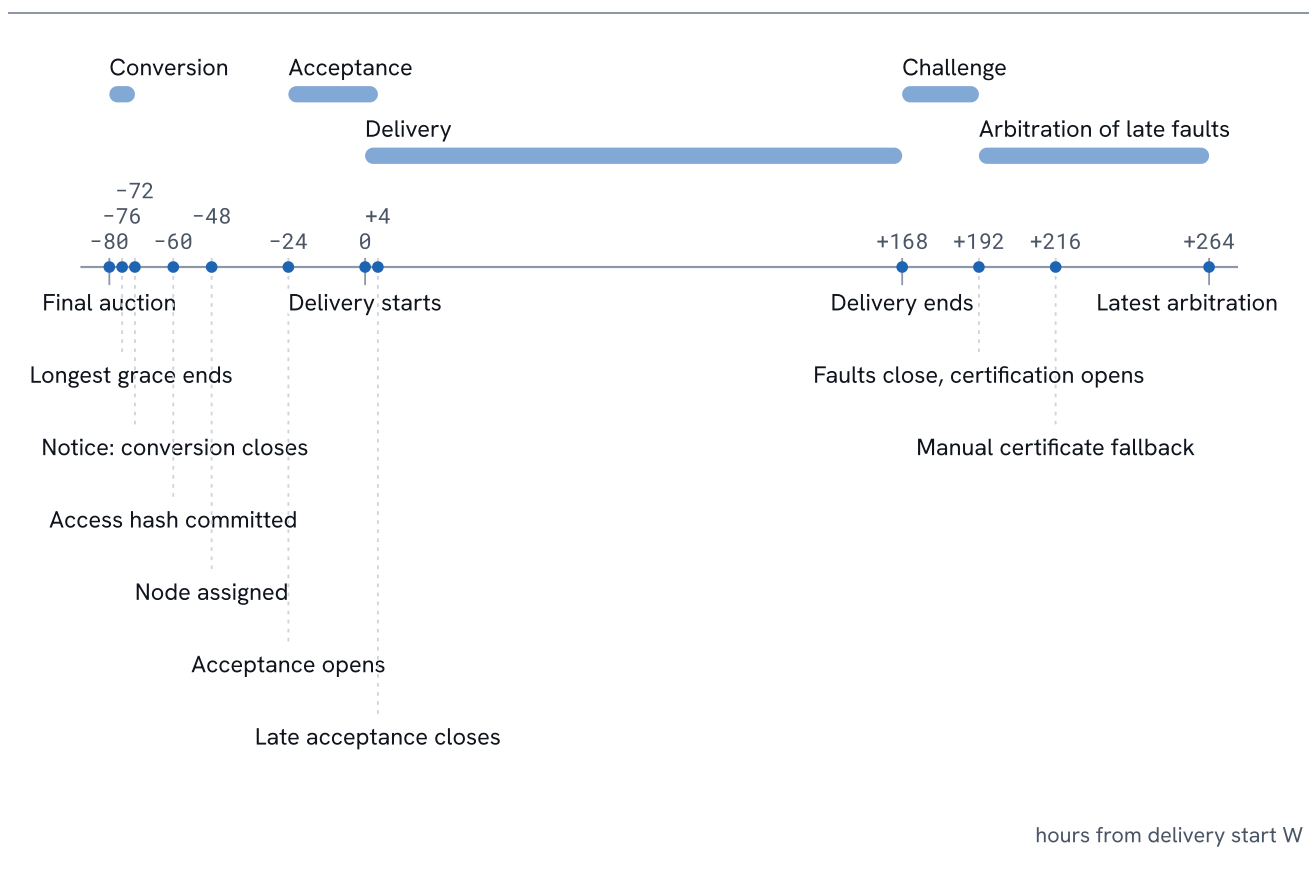


Figure 8 Deadlines from the final auction to the latest arbitration step, in hours from delivery start W.

## 4.6 Series states

The program stores no single phase field. The state of a series is the combination of its flags, its counters and the clock, and each transition is either a specific instruction or a fixed instant:

| State            | Entered by                                                    | What it permits                                                           |
|------------------|---------------------------------------------------------------|---------------------------------------------------------------------------|
| Created          | <code>create_series</code>                                    | setup steps                                                               |
| Ready            | <code>init_claim_mint</code> , book and registry grown        | orders                                                                    |
| Trading, round r | ready, before the final cutoff                                | orders until each cutoff; <code>clear_auction</code>                      |
| Paused           | <code>pause_series(true)</code>                               | no trading or transfers; cash and delivery paths continue                 |
| Sealed           | <code>seal_delivery</code>                                    | conversion until notice                                                   |
| After notice     | <code>notice_at</code> passes                                 | waivers, assignment, acceptance                                           |
| Delivering       | W                                                             | hourly <code>commit_epoch</code> , faults                                 |
| Challenge        | W + 168 h                                                     | faults until W + 192 h                                                    |
| Certifying       | W + 192 h                                                     | certificates, settlement                                                  |
| Closed           | last settlement or waiver, or <code>release_series_cap</code> | cash collection; cap returned                                             |
| Retiring         | <code>begin_retire_series</code> , closed and from W + 192 h  | receipt audits, then sweeps and account closures                          |
| Retired          | <code>close_retired_series</code>                             | the <code>Series</code> account no longer exists; workflow records remain |

Pausing is orthogonal to the schedule. While a series is paused, the program refuses orders, clears, capacity reservations, claim transfers and claim collection, and defers the minting of new claims; refunds, cancellations, cash collection and settlement continue, and a delivery request still burns its claim. A series is closed once it is sealed, its open interest, outstanding quantity and accounted escrow are all zero, and every issued node-week is settled or waived. `release_cap` then returns its reserved cap exactly once. The last `settle_ticket` or `waive_delivery` performs the release automatically; `release_series_cap` exists for any series that reaches the closed state otherwise.

The service reports the state of a series with the calendar phases of `getSeriesCalendar` , which depend on the clock alone: scheduled before W - 8 weeks, then trading until the final auction, notice until W - 72 hours, assignment until W - 48 hours, provisioning until W - 24 hours, acceptance until W, delivery until W + 168 hours, challenge until W + 192 hours, and settlement after that until the series is closed.

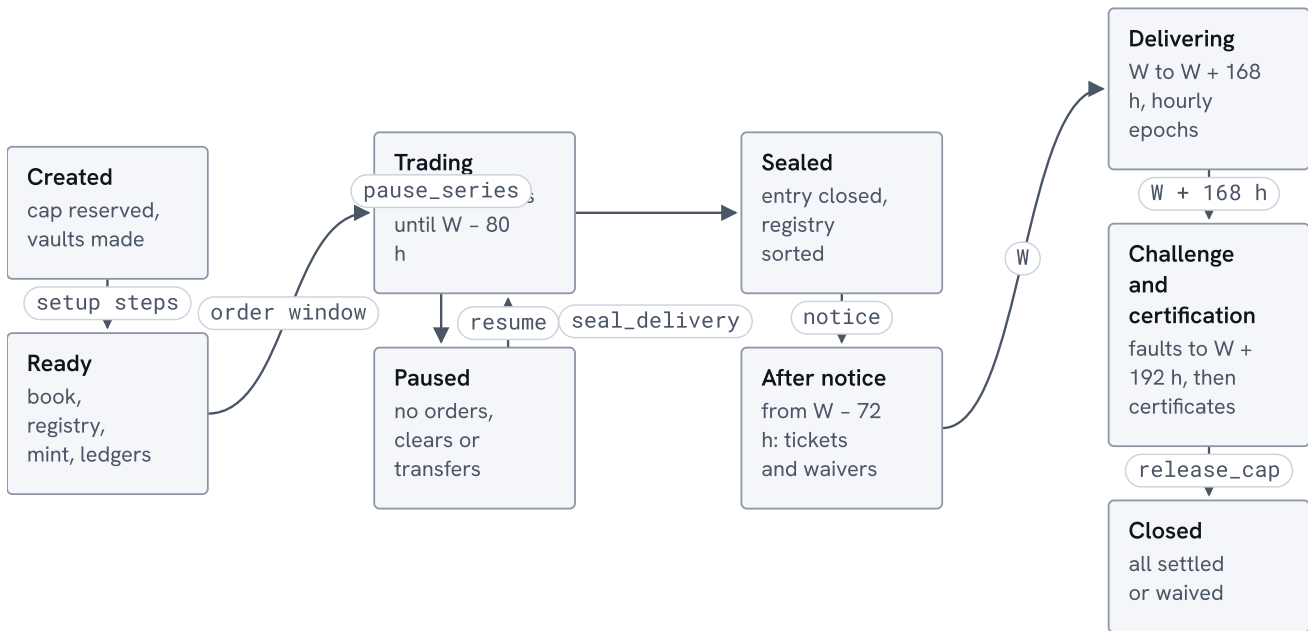


Figure 9 A series moves from creation through weekly rounds, sealing and delivery to closure; pausing suspends trading without stopping cash paths.

## 4.7 Retirement

A closed series still holds accounts that the operator and the crank paid for: the `Series` itself, its book, delivery registry, epoch ledger and health archive, its two USDC vaults, the claim mint with its inventory and the hook's account list, and one `AuctionResult` per round. Retirement closes them and returns each account's lamports to the key that funded it. Every retirement instruction is permissionless, and the worker sends them once a series qualifies (§16 Service infrastructure).

Each account's payer is recorded when it is funded. `create_series` records the admin as `Series.rent_payer`, which covers the series, its book, registry and vaults; book and registry growth accept only that key. `init_claim_mint` records its payer as `mint_rent_payer` for the mint, the inventory and the hook's list, `init_epoch_ledger` and `init_epoch_health` record `epoch_rent_payer` and `health_rent_payer`, and each `AuctionResult` records the payer of its clear. When a key other than the health archive's initializer grows the archive, the series records that key and its exact top-up, and retirement pays the top-up back to it before the rest of the archive's lamports go to the initializer.

Retirement runs in three stages.

1. **Begin.** `begin_retire_series` sets `Series.retiring`. It requires the closed state of the previous section (the series sealed, its cap released, open interest, open quantity and accounted escrow at zero, every issued unit settled or waived) and the clock at or after

delivery end plus the challenge window,  $W + 192$  hours. From then on `init_claim_mint`, `open_claim_account`, the evidence-account initializers and all growth steps refuse the series.

2. **Audit.** Three cursors on the `Series` walk every financial record of the series in order, one record per instruction. `audit_retirement_order` takes orders by arrival sequence and requires each to be settled with no pending cash and no pending mint; claim credits still on the receipt move into `Order.waived_claims`, because their units were waived at notice. `audit_retirement_ticket` takes tickets by index and requires each to be settled, with no open fault and no pending buyer or seller cash. `audit_retirement_unit` takes registry units by position and requires each to be requested or waived, its seller's `Obligation` to hold no escrow, locked collateral, short, reserved or assigned quantity, certified-unsettled quantity or pending cash, and the obligation's `Pledge`, if one exists, to be fully repaid with no pending lender cash. An uncollected cash credit holds its cursor until it is collected.
3. **Close.** Once the cursors reach `next_sequence`, `requested_count` and `allocation_count`, the closing instructions open. `sweep_retired_claims` burns the claims still held in the inventory or in wallet accounts through the mint's permanent delegate, thawing the account or resuming the mint first where needed and restoring that state afterwards, and emits `RetiredClaimsBurned` with the owner and quantity. `sweep_retired_surplus` moves any USDC left in the funding and escrow vaults to the fee vault, where it funds the buyback (§12 Fees). `close_retired_auction` closes the auction results in round order, each to its own payer. `close_retired_part` closes the book, registry, epoch ledger, health archive, both vaults, the inventory, the hook's list and the mint, each to its recorded payer; a token account closes only when empty, and the mint closes only at zero supply, after the inventory and the hook's list. `close_retired_series` closes the `Series` itself to its rent payer once all nine parts and every auction result are closed.

Every other account stays on chain, among them the series' orders, obligations, capacity reservations, tickets, node bookings, disputes, certificates, agent bindings and pledges, and the participants' eligibility, seller and node records and claim token accounts. Each closure returns the lamports the account holds, which is the rent its payer deposited under the cluster's rent schedule. The service's read model keeps the series, its auctions and its delivery history readable after the closure and marks the series archived.

## 4.8 Open-interest caps

Open interest is the original notional of node-weeks issued and not yet settled or waived. Two caps bound it. Each series has its own cap, chosen at listing and no larger than 250,000 USDC, and the caps of all unreleased series together may not exceed the total cap of 1,000,000 USDC. Reserving the cap at listing means that series never compete for headroom at clearing time: each clear checks only its own series cap and the total.

`clear_auction` passes the remaining notional headroom of the series and the remaining room under the limit of 512 node-weeks per series into the clearing computation, which trims new issuance to fit (see §5 The auction). It then requires the series total, the global total and the

series quantity all to remain within their limits. Claim resales move existing claims and consume no headroom. Open interest falls by the original notional of each node-week at settlement or waiver.

At an example price of 2.20 USDC per GPU-hour a full 250,000 USDC cap holds 84 node-weeks, 248,371.20 USDC of notional. The 512-node-week limit binds first only at prices of 0.36 USDC per GPU-hour or less. A series with open interest holds its reservation from listing until it closes, at the earliest  $W + 192$  hours. A region listed every week at the earliest moment therefore carries up to ten reservations at once: eight weeks in trading, one in delivery and one in its challenge or settlement. With both regions listed this way and equal caps, each series could be given at most  $1,000,000 / 20 = 50,000$  USDC.

#### 4.9 Worked example: the week of 14 December 2026

As an example, take the NA series for the delivery week starting Monday 14 December 2026, with auctions at 16:00 UTC.  $W$  is 1,797,206,400 in Unix seconds. The series may be listed from Monday 19 October 2026 00:00 UTC until Monday 30 November 2026 00:00 UTC. Listed at the earliest moment, it has eight auctions; the weeks remaining at each cutoff, rounded up to the micro-week, are the term the program uses for collateral.

| Auction | UTC                   | Hours before $W$ | Weeks remaining |
|---------|-----------------------|------------------|-----------------|
| $W - 8$ | Thu 22 Oct 2026 16:00 | 1,256            | 7.476191        |
| $W - 7$ | Thu 29 Oct 2026 16:00 | 1,088            | 6.476191        |
| $W - 6$ | Thu 5 Nov 2026 16:00  | 920              | 5.476191        |
| $W - 5$ | Thu 12 Nov 2026 16:00 | 752              | 4.476191        |
| $W - 4$ | Thu 19 Nov 2026 16:00 | 584              | 3.476191        |
| $W - 3$ | Thu 26 Nov 2026 16:00 | 416              | 2.476191        |
| $W - 2$ | Thu 3 Dec 2026 16:00  | 248              | 1.476191        |
| $W - 1$ | Thu 10 Dec 2026 16:00 | 80               | 0.476191        |

The deadlines that follow the final auction fall on these instants:

| Event                              | UTC                   |
|------------------------------------|-----------------------|
| Final auction grace ends (3,600 s) | Thu 10 Dec 2026 17:00 |
| Notice                             | Fri 11 Dec 2026 00:00 |
| Access commitment                  | Fri 11 Dec 2026 12:00 |
| Assignment deadline                | Sat 12 Dec 2026 00:00 |
| Acceptance opens                   | Sun 13 Dec 2026 00:00 |

| Event                                  | UTC                      |
|----------------------------------------|--------------------------|
| Delivery starts                        | Mon 14 Dec 2026 00:00    |
| Late acceptance closes                 | Mon 14 Dec 2026 04:00    |
| First epoch (hour 0) committable       | Mon 14 Dec 2026 01:00    |
| Delivery ends, last epoch committable  | Mon 21 Dec 2026 00:00    |
| Faults close, certification opens      | Tue 22 Dec 2026 00:00    |
| Manual certificate fallback (86,400 s) | Wed 23 Dec 2026 00:00    |
| Latest arbitration step                | Thu 24 Dec 2026 23:59:59 |

A buyer who holds a claim at 23:59 on Thursday 10 December can still convert it at 00:00:00 on Friday 11 December, the notice instant itself. From 00:00:01 the claim can no longer be converted, and anyone may waive the unconverted node-week, releasing its escrow to the seller. A fault filed at 23:59:59 on Monday 21 December has its cure deadline at 03:59:59 on Tuesday 22 December and reaches arbitration at 23:59:59 on Thursday 24 December; certification of that ticket waits until the dispute is closed.

## 5 The auction

Each series trades in a uniform-price call auction. Orders accumulate in the series' on-chain book during the week; nothing executes between cutoffs. At each scheduled Thursday cutoff the book for that round is closed, and one permissionless instruction, `clear_auction`, executes the whole book at a single price per GPU-hour. Every executed bid pays that price and every executed ask receives it. A series holds at most eight rounds, one per week from W-8 to W-1 (§4 Series and the calendar).

The call auction concentrates a week of orders into one print per round instead of a sequence of small trades. Fresh issuance by sellers and resale of existing claims meet in the same book, so a holder who no longer wants a node-week can sell it at the same price new supply clears at. The price is a deterministic function of the book: the program recomputes it from the order slots and rejects any other value, and any reader of the book can reproduce it with the shared preview `previewAuction`.

### 5.1 Order kinds

`place_order(nonce, side, price, quantity, collateral_budget)` creates an order. The side selects one of three kinds.

| Side | Kind       | Who may place it                                      | What placement locks                             |
|------|------------|-------------------------------------------------------|--------------------------------------------------|
| 0    | Bid        | Approved wallet with the buyer or trader role         | USDC deposit: limit notional plus trade fee      |
| 1    | Fresh ask  | Approved KYB seller with a <code>SellerProfile</code> | Collateral budget and unobligated capacity       |
| 2    | Resale ask | Approved wallet with the buyer or trader role         | The offered claims, moved into program inventory |

Prices are USDC base units (six decimals) per GPU-hour. A price must be positive and a multiple of 10,000 base units, the 0.01 USDC tick. Quantities are whole node-weeks from 1 to 512. The notional of an order is

$$N(p, q) = p \times 1344 \times q$$

where  $p$  is the price per GPU-hour in base units,  $q$  the quantity in node-weeks and 1,344 the GPU-hours in one node-week. The product is computed in checked 64-bit arithmetic; an overflow rejects the order.

Every order must also meet a minimum notional at its limit price,  $N(p, q) \geq M$ , where  $M$  is `Config.minimum_order_notional`. The program requires  $M$  to be positive; `init_config` sets it and `configure_auction` changes it. The service's default configuration sets  $M$  to 100,000,000 base units (100 USDC). The minimum gives every slot of the bounded book a capital cost.

### 5.1.1 Bids

A bid is prepaid in full at its limit price, including the trade fee:

$$F = N(p_b, q) + [N(p_b, q) \times f / 10,000]$$

where  $p_b$  is the bid's limit,  $q$  its quantity and  $f$  the trade fee in basis points from `Config.trade_fee_bps` at placement (50, or 0.5 %). The fee rounds up to one base unit. The deposit  $F$  moves from the bidder's USDC account into the series funding vault, the PDA `["funding", series]`. The order receipt stores  $f$  as `Order.trade_fee_bps`, so a later fee change never alters what an existing order pays. A bid for 3 node-weeks at an example limit of 2.40 USDC per GPU-hour deposits 9,676.80 USDC of notional plus 48.384 USDC of fee, 9,725.184 USDC in total.

A bid also requires the bidder's canonical claim token account for the series to exist and be thawed, so that the claims it buys can be delivered to it. When it does not, the transaction the service prepares opens or thaws it first with `open_claim_account`, paid by the bidder (§7 Claims and transfer).

### 5.1.2 Fresh asks

A fresh ask sells node-weeks that do not yet exist. If it fills, the program mints new claims and the seller's `Obligation` grows. The seller must hold the seller role with KYB, and the ask must fit inside the seller's unobligated capacity for the series (§6 Collateral):

$$S + R + q \leq K$$

where  $K$  is `Obligation.capacity` (reserved accredited node-weeks),  $S$  the node-weeks already short,  $R$  the quantity reserved by the seller's other open asks and  $q$  the new ask's quantity. The seller also names a collateral budget  $B$ , which must cover the requirement at the ask's limit price for its full quantity and must not exceed the seller's free collateral:

$$C(N(p_a, q), T_{\text{now}}) \leq B \leq \text{free collateral}$$

where  $p_a$  is the ask's limit and  $C(N, T)$  the collateral function of §6 evaluated with  $T$ , the weeks from placement to delivery start.  $B$  moves from free to reserved collateral and  $q$  is added to `Obligation.reserved_quantity`. A budget above the minimum keeps the ask eligible at clearing prices above its limit, because eligibility is tested at the clearing price.

### 5.1.3 Resale asks

A holder offers existing claims into a later round. Placement transfers  $q$  claims from the holder's canonical claim account to the series inventory account, the PDA `["claims", series]`, with a Token-2022 `TransferChecked`; the transfer hook checks both ends (§7). A

resale ask can never exceed what the wallet holds. It creates no obligation, consumes no issuance headroom and adds no open interest, because the escrow behind those claims was posted when they were first issued.

#### 5.1.4 Entry conditions

All three kinds share the same entry conditions. The series must be unpaused, its claim mint initialized and its delivery allocation not yet sealed, and the order window for the current round must be open:

$$t < t_A \quad \text{and} \quad t < t_N \quad \text{and} \quad t_A < t_N$$

where  $t$  is the current time,  $t_A$  the round's scheduled cutoff ( `Series.next_auction_at` ) and  $t_N$  the notice instant,  $W - 72 \text{ h}$ . Entry for a round therefore closes at its cutoff, and no orders enter after the last scheduled cutoff.

The wallet's `Eligibility` must be approved (KYC, sanctions clear, allowlisted). A wallet may have at most `Config.max_open_orders_per_wallet` open orders in the book; the program accepts values from 1 to 8 for this limit, one eighth of the admitted book, and the service's default configuration uses 8. A wallet with an open bid cannot place either kind of ask in the same book, and a wallet with an open ask cannot bid ( `SelfTrade` ).

Each accepted order takes the series' next arrival sequence, `Series.next_sequence` , a monotonic 32-bit counter, and receives a durable receipt `Order` at the PDA `["order", series, owner, nonce]` . The receipt records the price, deposit, collateral budget, quantity, sequence, slot, round, side and fee rate, and it outlives the book slot. Every check runs before any token moves, so a rejected order transfers nothing.

## 5.2 The book

The book is a zero-copy account, `AuctionBook` , at the PDA `["book", series]` . It holds an 8-byte tag, `HGB00K02` , followed by 1,024 slots of exactly 64 bytes: 65,544 bytes in total. It is created at 10,240 bytes with the series and grown by permissionless `grow_book` calls of at most 10,240 bytes each; placement and clearing require the full size.

```
#[repr(C)]
pub struct OrderSlot {
    pub owner: [u8; 32],
    pub price: u64,
    pub collateral_budget: u64,
    pub quantity: u32,
    pub filled: u32,
    pub sequence: u32,
    pub minted_new: u16,
    pub side: u8,
    pub state: u8,
}
```

### 5.2.1 Admitted slots

The program admits orders only into the first 64 slots, 32 per economic direction: bids in one direction, fresh and resale asks together in the other. Placement and clearing both reject the book if any of the remaining 960 slots holds an open order. The bound of 64 is compiled into the program; configuration cannot raise it.

| State | Meaning                       | Occupies its direction |
|-------|-------------------------------|------------------------|
| 0     | Empty                         | No                     |
| 1     | Open                          | Yes                    |
| 4     | Cleared, not yet materialized | Yes                    |
| 2     | Cancelled                     | No                     |
| 3     | Materialized                  | No                     |

### 5.2.2 Admission and displacement

If the incoming order's direction occupies fewer than 32 slots and a free slot exists, the order takes the lowest-numbered free slot. Otherwise it must displace the worst open order of its own direction. The worst bid is the one with the lowest price; the worst ask, fresh or resale, is the one with the highest price. Among equal prices the latest arrival is the worst. The newcomer must be strictly better: a bid must bid more than the worst bid, an ask must ask less than the worst ask. An equal price is rejected with `BookFull` before any funds move, so equal prices keep their arrival priority. Cleared orders awaiting materialization are never displaced.

Displacement happens atomically inside the newcomer's `place_order` and needs no signature from the displaced owner. The old receipt and the owner's canonical accounts, passed as the optional `evicted_*` accounts and checked against their PDAs, fix every beneficiary:

- A displaced bid's full deposit `F` returns to the owner's canonical USDC account in the same instruction.
- A displaced fresh ask's budget moves from reserved back to free collateral, and its quantity leaves `Obligation.reserved_quantity`.
- A displaced resale ask's escrowed claims become a claim credit, `Order.pending_claims`, on the old receipt.

The old receipt is marked cancelled, evicted and settled, and the program emits `OrderEvicted`. Two kinds of credit arise here and in every later payout. If the canonical USDC account is missing, frozen, closed or owned by someone else, the amount stays in the funding vault as a cash credit, `Order.pending_usdc`, collected later by `collect_cash`. Claims owed to a wallet sit in inventory as `Order.pending_claims` until `collect_claims` delivers them. A participant therefore cannot block book reuse by disabling its own accounts.

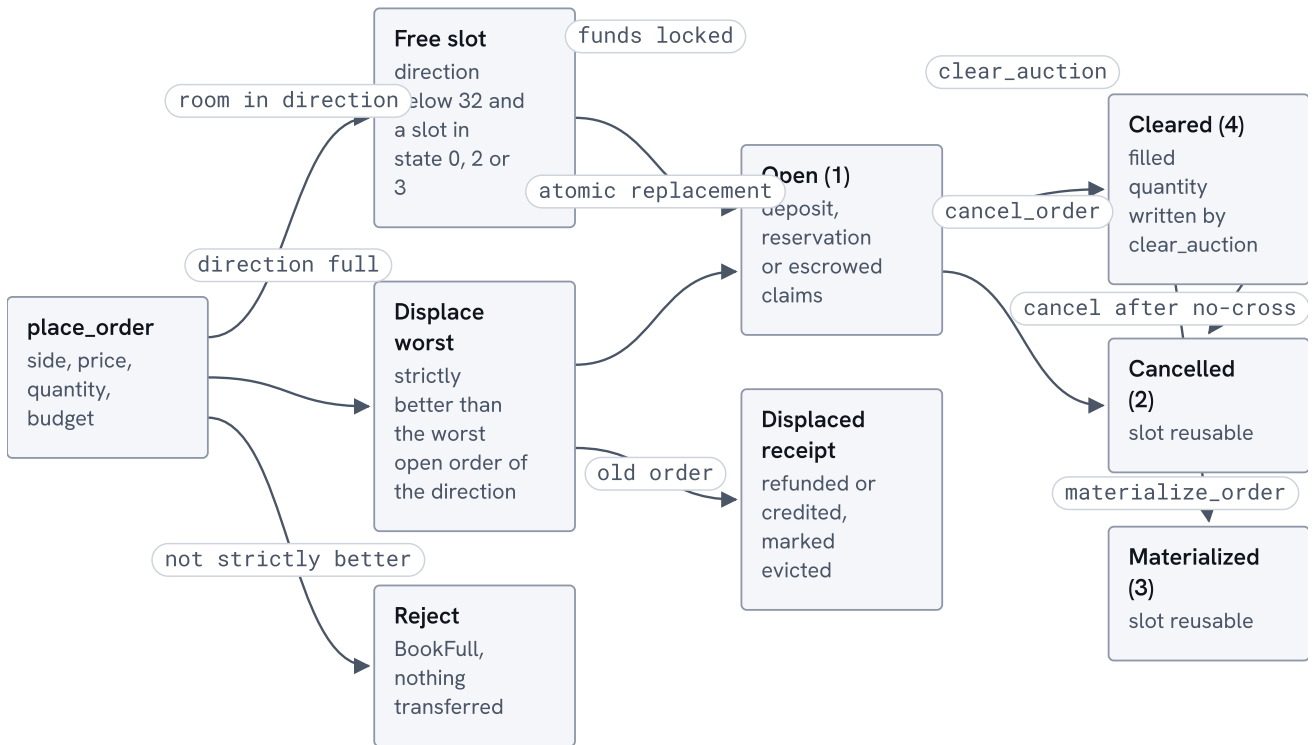


Figure 10 How a new order enters one of the 64 admitted slots and how a slot returns to the free pool.

### 5.2.3 Cancellation

`cancel_order(nonce)` requires the owner's signature. An open order of the current round can be cancelled when

$$t < t_A \quad \text{or} \quad t \geq t_A + \min(g, 14,400 \text{ s}) \quad \text{or} \quad t \geq t_N$$

where  $g$  is the round's grace period, `Series.round_grace_seconds`. The first order of a round copies  $g$  from `Config.failed_auction_grace_seconds`, which the program accepts from 1 to 14,400 seconds and the service configures at 3,600 seconds by default; a later configuration change cannot lengthen a running round's lock. Between the cutoff and the end of grace, open orders are locked so that the clear sees the book as it stood at the cutoff. At notice every open order can always exit.

A cleared order can also be cancelled when its round's `AuctionResult` records zero volume at price zero. Cancellation mirrors displacement: a bid gets its whole deposit back with no fee, a fresh ask's budget and capacity are released, and a resale ask's claims are credited back. The slot moves to state 2.

## 5.3 Clearing

### 5.3.1 The clear instruction

`clear_auction(round, price)` may be sent by anyone; the service's worker sends it at each cutoff. It requires an unpaused, unsealed series, `round` equal to `Series.round`, and  $t_A \leq t < t_N$ . It accepts exactly six accounts: payer, `Config`, `Series`, the book, the new `AuctionResult` at the PDA `["auction", series, round]` and the System Program; any additional account rejects before the solver runs.

The collateral term of the round is measured at the scheduled cutoff, not at the moment the transaction lands:

$$T_A = \lceil (W - t_A) \times 10^6 / 604,800 \rceil$$

where  $W$  is the delivery start,  $t_A$  the scheduled cutoff in Unix seconds and 604,800 the seconds in a week;  $T_A$  is in millionths of a week, rounded up. A late crank therefore cannot lower any seller's requirement.

The caller proposes a price. The program computes the canonical price itself and rejects any other proposal with `InvalidClearing` before it mutates a slot. A caller can delay a clear but cannot choose its price. A zero proposal is accepted only when the canonical result has no executable volume.

### 5.3.2 The clearing rule

For each positive tick  $p$  the program evaluates

$$\begin{aligned} D(p) &= \sum q_i \text{ over bids with } p_i \geq p \\ S_f(p) &= \sum q_j \text{ over fresh asks with } p_j \leq p \text{ and } C(N(p, q_j), T_A) \leq B_j \\ S_r(p) &= \sum q_k \text{ over resale asks with } p_k \leq p \\ I(p) &= \min(\lfloor H / 1344 \rfloor / p, 512 - Q) \\ V(p) &= \min(D(p), \min(S_f(p), I(p)) + S_r(p), 512) \\ p^* &= \min \{ p : V(p) = \max V > 0 \} \end{aligned}$$

$D$  is demand,  $S_f$  the eligible fresh supply,  $S_r$  the resale supply and  $V$  the executable volume, all in node-weeks.  $B_j$  is ask  $j$ 's collateral budget.  $H$  is the series' open-interest headroom, `Series.notional_cap` minus `Series.open_interest`, and  $Q$  is `Series.open_quantity`, the node-weeks issued so far;  $I(p)$  is the most fresh issuance the series can absorb at  $p$ . The clearing price  $p^*$  maximizes executable volume, and among prices with that volume it is the lowest positive tick. If the maximum volume is zero the round does not cross.

A fresh ask counts only when its whole posted quantity is collateral-backed at the candidate price. An ask whose budget backs only part of its quantity contributes nothing at that price, rather than a fraction. Resale supply is never limited by headroom. Rationing replaces vetoes: bids priced above  $p^*$  fill at  $p^*$  and the excess at the margin is rationed, so adding any order to a book never reduces its maximum executable volume.

### 5.3.3 Computing the price

The program does not scan ticks. It converts each open slot into at most two events: a bid becomes a demand decrease at its limit plus one tick; a resale ask becomes a supply start at its limit; a fresh ask becomes a supply start at its limit and, when its budget runs out first, an expiry one tick above the highest price its budget backs:

$$P_{\max} = \lfloor [B \times 10^{10} / \rho] / (1344 \times q) / 10,000 \rfloor \times 10,000$$

$$\rho = b \times 10^6 + w \times T_A$$

where  $b$  and  $w$  are the series' base and per-week collateral rates in basis points (§6). The events are sorted by price and swept upwards. Between two supply starts, demand, eligible fresh supply and issuance headroom can only fall, so the maximum volume occurs at a supply start and the first one that reaches it is the lowest such tick. The sweep evaluates  $V$  only at supply starts and stops when  $V$  reaches a precomputed upper bound or demand falls to the best volume found.

### 5.3.4 Allocation

Once  $p^*$  is fixed, every executable order enters one priority queue: bids at or above  $p^*$ , and eligible asks at or below it. Bids rank by price descending, asks by price ascending, and equal prices by arrival sequence. Slot position never affects priority, so reusing a low slot cannot move a later order ahead.

1. Bids fill in priority order until  $V$  is exhausted. The marginal bid may fill partially; bids after it receive nothing.
2. Fresh asks, in ask priority, draw on the issuance limit  $I(p^*)$ . Each takes the smaller of its quantity and the remaining limit; any excess is recorded as cap-trimmed.
3. All asks, fresh after trimming and resale in full, fill in priority order until  $V$  is exhausted.
4. The fresh quantity is attributed to bids in bid priority order as `minted_new`.

The program then requires that bid fills, ask fills and  $V$  are equal, and that the attributed fresh quantity equals the fresh fills; otherwise the clear fails with `InvalidClearing`. Each slot's `filled` and `minted_new` are written and every open slot, filled or not, moves to state 4.

### 5.3.5 Open interest and caps

Only fresh issuance adds open interest. After allocation the program adds  $N(p^*, Q_{\text{new}})$ , the notional of the round's fresh quantity, to `Series.open_interest` and to `Config.total_open_interest`, and adds  $Q_{\text{new}}$  to `Series.open_quantity`. It rejects the clear

with `CapReached` if the series would exceed its `notional_cap`, the global total would exceed `Config.total_cap`, or the series would exceed 512 node-weeks. Each series reserves its cap from the global total when it is created; a series cap is at most 250,000 USDC and the reserved caps together at most 1,000,000 USDC, so series never compete for global headroom during a clear.

The clear then increments `Series.round`, advances `Series.next_auction_at` by one week and releases the round's grace snapshot.

#### 5.3.6 Results, failed auctions and grace

Every clear, crossing or not, writes an `AuctionResult` recording the price, volume, fresh quantity, the scheduled cutoff, `T_A`, the round's grace, the clear time and a typed no-cross reason. It also carries `fresh_minted_quantity`, the count of the round's fresh claims minted so far, which materialization advances, and `rent_payer`, the key that paid for the account in its clear and receives its rent back when the series retires (§4 Series and the calendar).

| Code | Reason                        | Condition                                      |
|------|-------------------------------|------------------------------------------------|
| 0    | None                          | The round crossed                              |
| 1    | <code>EmptySide</code>        | No bids, or no asks of either kind             |
| 2    | <code>LimitsDoNotCross</code> | Highest bid below lowest ask                   |
| 3    | <code>NoFeasiblePrice</code>  | Limits cross but no tick has executable volume |

A failed auction is a round whose clear has not landed by  $t_A + g$ . Owners may then cancel, and the clear remains callable until notice for whatever is still open. For the final scheduled round, sealing the delivery allocation with `seal_delivery` (§8 Delivery) becomes possible at the eighth cutoff plus grace even without a clear; sealing closes entry and clearing for good, and open orders keep their cancellation path. The program bounds the cutoff time of day so that the latest cutoff plus the maximum grace of 4 hours still leaves at least 4 hours before notice on Friday 00:00 UTC; the latest permitted cutoff is therefore 16:00 UTC on Thursday.

### 5.4 Materialization

A clear only writes fills into slots. Money and claims move afterwards, one order per transaction, through `materialize_order(nonce)`; `claim_fill(nonce)` is the same instruction under its original name. Anyone may send it, because the receipt fixes every payee. It requires the slot in state 4 and the `AuctionResult` of the receipt's round, and it is idempotent: a settled receipt returns without effect. After materialization the slot is in state 3 and free for reuse.

#### 5.4.1 Bids

For a bid that filled  $x$  node-weeks at  $p^*$ :

```
spent = N(p*, x) + [N(p*, x) × f / 10,000]
refund = F - spent
```

The fee moves to the protocol fee vault. The refund returns the unfilled quantity and the price improvement, the difference between the limit and  $p^*$  on the filled quantity together with the fee on that difference; it goes to the canonical USDC account or becomes a cash credit. The filled notional stays in the funding vault, and the bid receives  $x$  claims as

```
Order.pending_claims .
```

#### 5.4.2 Fresh asks

For a fresh ask that filled  $x$  node-weeks the program locks  $L = C(N(p^*, x), T_A)$ , returns  $B - L$  to free collateral, removes the whole posted quantity from `reserved_quantity` and adds  $x$  to `short_quantity`. It moves  $N(p^*, x)$  from the funding vault to the series escrow vault, `["escrow", series]`, and mints  $x$  claims into program inventory. While the series is paused the mint is deferred as `Order.pending_mint`. The fill also appends  $x$  original issuance units, each with its price and collateral share, to the delivery allocation registry (§8 Delivery).

#### 5.4.3 Resale asks

A resale ask that filled  $x$  of its  $q$  node-weeks receives  $N(p^*, x)$  from the funding vault. The unsold  $q - x$  claims become a claim credit on its receipt.

#### 5.4.4 Collecting

`collect_claims(nonce)` first mints any deferred `pending_mint`, resuming and re-pausing the mint atomically if the series is paused; this step runs even after notice. It then transfers `pending_claims` from inventory to the owner's canonical claim account. That outward transfer requires an unpaused series,  $t < t_N$ , an approved owner and a thawed account; for a bid it also requires the whole round to be backed, `fresh_minted_quantity` equal to the round's fresh quantity, so no buyer collects inventory ahead of its minting. `collect_cash(nonce)` pays a cash credit to the canonical USDC account without the owner's signature, or to another account of the owner with it. Both are idempotent.

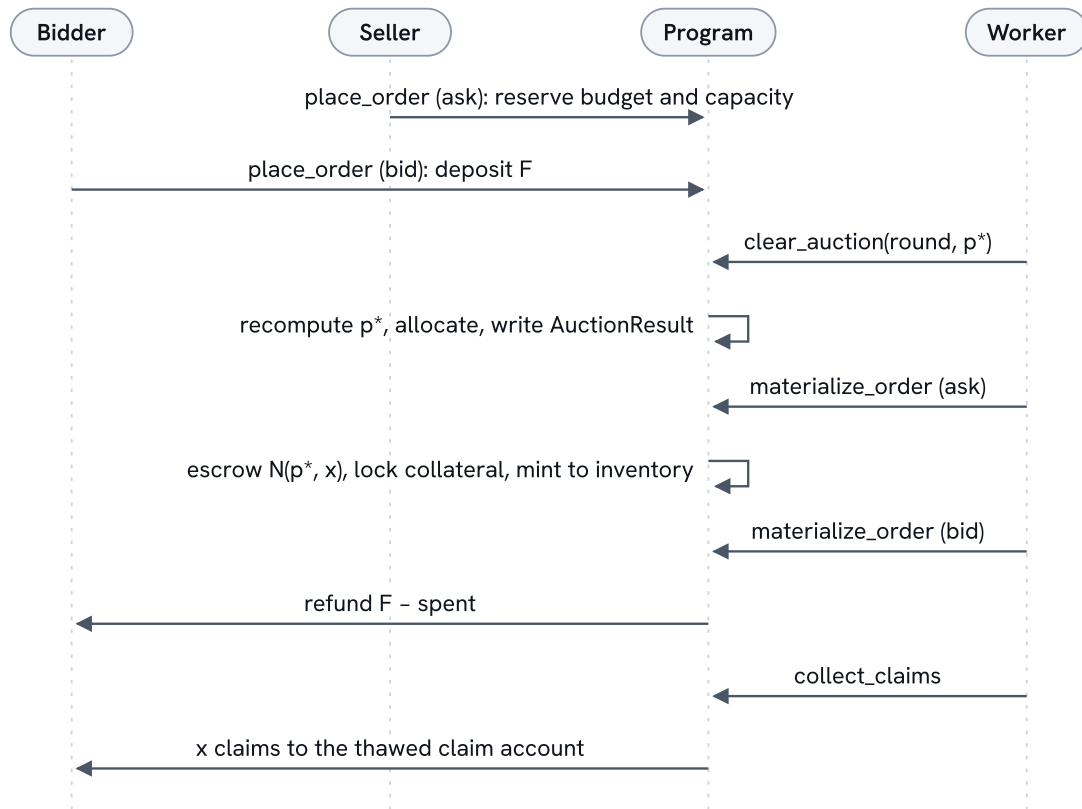


Figure 11 The life of one fresh ask and one bid through placement, clearing, materialization and collection.

## 5.5 Worked example

The book below is an example. It is the W-4 round of a series with a 16:00 UTC cutoff, so  $W - t_A$  is 584 hours and  $T_A$  is 3,476,191 millionths of a week. Nothing has been issued yet, the series headroom is 250,000 USDC, and the series uses the standard schedule of 2,000 and 150 basis points.

| Order | Kind       | Limit (USDC/GPU-h) | Node-weeks | Sequence |
|-------|------------|--------------------|------------|----------|
| B1    | Bid        | 2.40               | 3          | 1        |
| A1    | Fresh ask  | 2.00               | 3          | 2        |
| B2    | Bid        | 2.30               | 4          | 3        |
| A2    | Fresh ask  | 2.15               | 4          | 4        |
| R1    | Resale ask | 2.20               | 2          | 5        |
| B3    | Bid        | 2.20               | 4          | 6        |
| A3    | Fresh ask  | 2.20               | 3          | 7        |
| B4    | Bid        | 2.10               | 2          | 8        |
| A4    | Fresh ask  | 2.10               | 2          | 9        |

| Order | Kind | Limit (USDC/GPU-h) | Node-weeks | Sequence |
|-------|------|--------------------|------------|----------|
| B5    | Bid  | 2.20               | 2          | 10       |

A1, A2 and A3 posted budgets that back their full quantity up to 2.60 USDC per GPU-hour: 2,643.264083, 3,524.352110 and 2,643.264083 USDC. A4 posted 1,457.184046 USDC, which backs its 2 node-weeks only up to 2.15. Each budget exceeds its placement minimum, computed 24 hours before the cutoff at the ask's limit.

| Tick | D  | S_f | S_r | V  |
|------|----|-----|-----|----|
| 2.10 | 15 | 5   | 0   | 5  |
| 2.15 | 13 | 9   | 0   | 9  |
| 2.20 | 13 | 10  | 2   | 12 |
| 2.25 | 7  | 10  | 2   | 7  |

Volume peaks at 12 node-weeks at 2.20, the clearing price. A4 drops out of fresh supply at 2.20, because its full quantity needs 1,491.072047 USDC of collateral there; it is reported as ineligible. At 2.20, 13 node-weeks are bid and 12 offered, so the bids at 2.20 are rationed by sequence: B3 (sequence 6) fills all 4 and B5 (sequence 10) fills 1 of 2. Every ask at or below 2.20 fills. Fresh issuance is 10 node-weeks with 29,568 USDC of new open interest, and 2 node-weeks are resold.

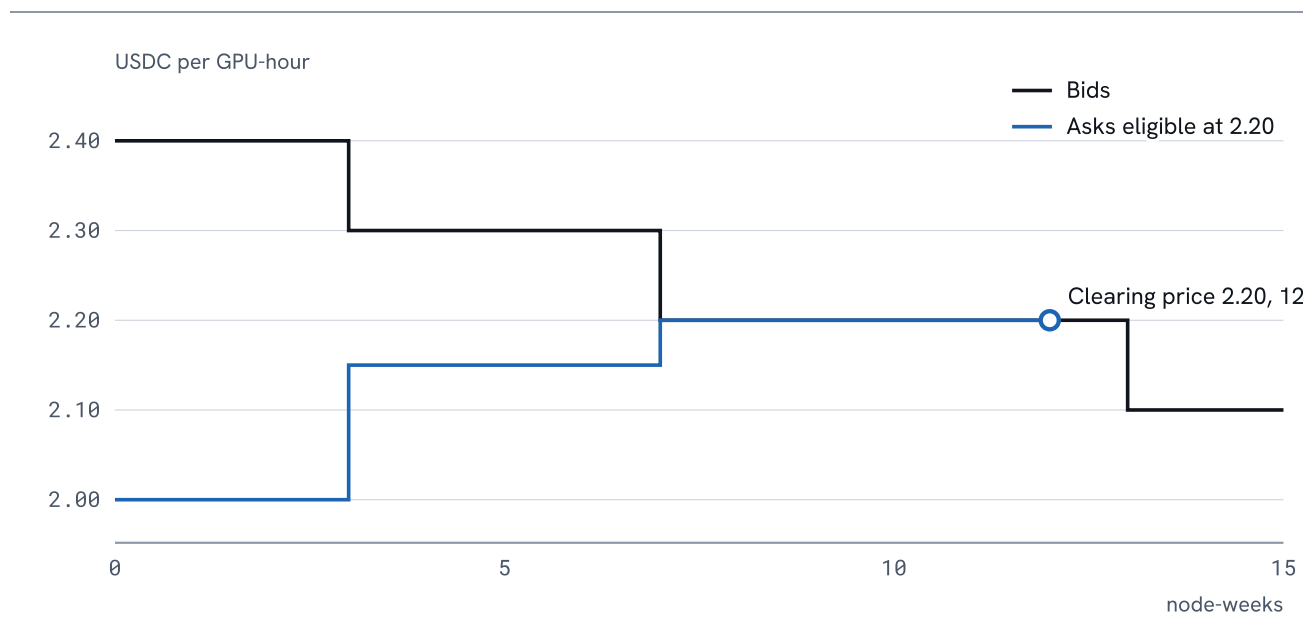


Figure 12 Demand and eligible supply in the example book; A4 is excluded because its budget does not back its full quantity at 2.20.

Materialization settles each receipt as follows; amounts are USDC.

| Order | Filled | Deposit    | Spent incl. fee | Refund    |
|-------|--------|------------|-----------------|-----------|
| B1    | 3      | 9,725.184  | 8,914.752       | 810.432   |
| B2    | 4      | 12,426.624 | 11,886.336      | 540.288   |
| B3    | 4      | 11,886.336 | 11,886.336      | 0         |
| B4    | 0      | 5,673.024  | 0               | 5,673.024 |
| B5    | 1      | 5,943.168  | 2,971.584       | 2,971.584 |

| Order | Filled | To escrow or seller | Collateral locked | Budget returned |
|-------|--------|---------------------|-------------------|-----------------|
| A1    | 3      | 8,870.40 to escrow  | 2,236.608070      | 406.656013      |
| A2    | 4      | 11,827.20 to escrow | 2,982.144093      | 542.208017      |
| A3    | 3      | 8,870.40 to escrow  | 2,236.608070      | 406.656013      |
| A4    | 0      | none                | 0                 | 1,457.184046    |
| R1    | 2      | 5,913.60 to R1      | none              | none            |

The money balances: the four filled bids pay 35,481.60 USDC of notional, of which 29,568.00 moves to escrow behind the new claims and 5,913.60 pays the reseller, plus 177.408 USDC of trade fees. B1's refund of 810.432 USDC is the price improvement of 0.20 USDC per GPU-hour on 3 node-weeks, 806.40 USDC, plus the 4.032 USDC of fee on it.

|                          |                                                                                      |                |
|--------------------------|--------------------------------------------------------------------------------------|----------------|
| Deposit                  |  | 9,725.18 USDC  |
| Filled notional          |  | -8,870.40 USDC |
| Trade fee                |   | -44.35 USDC    |
| Price improvement refund |   | -806.40 USDC   |
| Fee refund               |   | -4.03 USDC     |

Figure 13 Where B1's prepaid deposit goes when it fills 3 node-weeks at 2.20 against a 2.40 limit.

If the same book met a series with only 25,000 USDC of headroom left, the issuance limit at 2.20 would be  $\lfloor [25,000,000,000 / 1344] / 2,200,000 \rfloor = 8$  node-weeks. The price stays 2.20 and volume falls to 10: A1 and A2 fill in full, A3 fills 1 and has 2 node-weeks cap-trimmed, R1 still sells 2, and the bids ration to B1 3, B2 4 and B3 3.

## 5.6 Compute budget

`clear_auction` does bounded work regardless of prices or quantities. It reads exactly the 64 admitted slots and checks the 960 tail slots once. It sorts with fixed bitonic sorting networks, whose comparison schedule does not depend on the data. Each event packs into 64 bits (41 bits of price ticks, then quantity and kind) and each priority entry into 128 bits (price key, sequence, slot), so sorting moves small words. Collateral eligibility uses a precomputed reciprocal of the reduced collateral rate, which yields the exact quotient or one less, corrected by a single comparison. Clearing makes no token transfer, mint or evidence check; its only cross-program call creates the fixed-size `AuctionResult`.

| Work for one clear           | Bound                   |
|------------------------------|-------------------------|
| Tail checks                  | 960 slots               |
| Solver input                 | 64 orders               |
| Events                       | 128 rows                |
| Event sorting network        | 1,792 compare-exchanges |
| Priority rows                | 64 rows                 |
| Priority sorting network     | 672 compare-exchanges   |
| Fill and issuance allocation | 64 entries each         |

The clear transaction is 401 bytes. It requests 1,400,000 compute units and a 256 KiB heap; the program installs an ascending bump allocator over the whole 256 KiB frame, because the default allocator uses only 32 KiB of it. The design's budget for a canonical clear is a reviewed ceiling of 1,370,000 compute units inside that request.

| Component                                                         | Compute units |
|-------------------------------------------------------------------|---------------|
| Two canonical PDA searches, each at most 256 attempts at 1,500 CU | 768,000       |
| Auction math, including allocation and arithmetic                 | 460,000       |
| Entrypoint, account validation, serialization and handler         | 130,000       |
| Other runtime charges and transaction setup                       | 12,000        |
| Reviewed ceiling                                                  | 1,370,000     |
| Requested limit                                                   | 1,400,000     |

The ceiling leaves 30,000 compute units, 2.14 % of the request, as margin. Executed against the compiled program, the most expensive of the adversarial 64-order book shapes uses 180,314 compute units. Raising the admitted bound of 64 requires a new compiled program and a new review of this budget.

## 6 Collateral

Seller collateral is USDC that a seller deposits into a vault controlled by the market program. A fresh ask reserves part of it, each node-week that fills locks a share of it, and settlement, waiver or buyer default releases it. Buyers post no collateral: they prepay their bids in full (§5 The auction).

Collateral is the fund from which liquidated damages are paid when a seller under-delivers, up to the full amount allocated to each delivery ticket. Its size grows with the time remaining before delivery, because the cost of replacing capacity can move further before a distant window than before a near one. The amount is fixed when a fill locks it; there is no variation margin.

### 6.1 The seller's vault

`register_seller(kyb_hash, encryption_key)` creates the seller's `SellerProfile` at the PDA `["seller", wallet]` and its collateral vault at `["collateral", seller_profile]`, a USDC token account whose authority is the `SellerProfile` PDA. Registration requires an approved `Eligibility` with KYB and the seller role, and nonzero KYB hash and X25519 encryption key. Only program instructions signed by the profile's PDA can move USDC out of the vault.

The profile splits the vault into three balances. Every instruction that changes one of them changes it in the same transaction as the token transfer it accounts for, if any.

| Balance  | Field                            | Backs                           | Leaves through                                   |
|----------|----------------------------------|---------------------------------|--------------------------------------------------|
| Free     | <code>free_collateral</code>     | Nothing; withdrawable           | <code>withdraw_collateral</code> , ask placement |
| Reserved | <code>reserved_collateral</code> | Budgets of open fresh asks      | Fill, cancellation, displacement                 |
| Locked   | <code>locked_collateral</code>   | Filled node-weeks, unit by unit | Settlement, waiver                               |

#### 6.1.1 Deposits and withdrawals

`deposit_collateral(amount)` requires a positive amount and the seller's signature. It transfers the amount from the seller's USDC account into the vault and adds it to free collateral.

`withdraw_collateral(amount)` requires a positive amount no larger than free collateral. The vault pays the seller's account under the profile PDA's signature, and free collateral falls by the amount. A withdrawal can never touch reserved or locked collateral: those balances back open asks and filled node-weeks, and they return to free only through the program events described below.

### 6.2 The requirement

The required collateral for a notional  $N$  with  $T$  millionths of a week remaining before delivery start is

$$T = [(W - t) \times 10^6 / 604,800]$$

$$\rho = b \times 10^6 + w \times T$$

$$C(N, T) = [N \times \rho / 10^{10}]$$

where  $W$  is the delivery start and  $t$  the instant of measurement, both in Unix seconds, and 604,800 the seconds in a week.  $b$  is the base rate, 2,000 basis points, and  $w$  the add-on, 150 basis points per week.  $N = p \times 1344 \times q$  is the notional in USDC base units at price  $p$  for  $q$  node-weeks. The 1.5 percentage-point add-on therefore multiplies the fractional number of weeks from the measurement instant until delivery starts, kept to six decimals and rounded up, and applies to the whole notional of the order or fill. The requirement rounds up to one USDC base unit.

The rate runs from 20 % with no time left to 32 % at the eight-week listing horizon.

`create_spec` writes  $b$  and  $w$  into the `ContractSpec`, and `create_series` copies them onto the `Series`, so an existing series keeps its schedule for life.

The program applies the function at three points, with different inputs.

| Point               | Instruction                    | Price          | Quantity        | Measured at      |
|---------------------|--------------------------------|----------------|-----------------|------------------|
| Reservation minimum | <code>place_order</code>       | Ask's limit    | Posted          | Placement        |
| Eligibility         | <code>clear_auction</code>     | Clearing price | Posted, in full | Scheduled cutoff |
| Lock                | <code>materialize_order</code> | Clearing price | Filled          | Scheduled cutoff |

The clearing price is at least the ask's limit, while the scheduled cutoff is later than placement, so the lock can be larger or smaller than the reservation minimum. The seller's explicit budget absorbs the difference: an ask whose budget does not back its full quantity at the clearing price is ineligible in that round (§5).

The schedule for one node-week at an example price of 2.20 USDC per GPU-hour, a notional of 2,956.80 USDC, measured at each of the eight auctions with a 16:00 UTC cutoff:

| Auction | Hours to W | T (weeks) | Rate         | Collateral (USDC) |
|---------|------------|-----------|--------------|-------------------|
| W-8     | 1,256      | 7.476191  | 31.2142865 % | 922.944024        |
| W-7     | 1,088      | 6.476191  | 29.7142865 % | 878.592024        |
| W-6     | 920        | 5.476191  | 28.2142865 % | 834.240024        |
| W-5     | 752        | 4.476191  | 26.7142865 % | 789.888024        |
| W-4     | 584        | 3.476191  | 25.2142865 % | 745.536024        |
| W-3     | 416        | 2.476191  | 23.7142865 % | 701.184024        |
| W-2     | 248        | 1.476191  | 22.2142865 % | 656.832024        |

| Auction | Hours to W | T (weeks) | Rate         | Collateral (USDC) |
|---------|------------|-----------|--------------|-------------------|
| W-1     | 80         | 0.476191  | 20.7142865 % | 612.480024        |

The final digits show both roundings: 584 hours is 3.4761904... weeks, rounded up to 3.476191, and the product rounds up to the next base unit.

## 6.3 Capacity

Collateral alone never makes an ask valid; the seller must also have unobligated accredited capacity in the series' region. An inspector who sits on the committee, holds the inspector role and is not the seller accredits each node with `accredit_node` for 90 days, binding its region and the hashes of its eight GPU UUIDs (§3 Participants and eligibility).

`reserve_capacity` commits one accredited node to one series. The series must be unpaused and before notice, the node's region must equal the series' region, and its accreditation must last until the end of the delivery week. The instruction creates a `CapacitySlot` at the PDA `["capacity", node, W]`, adds one node-week to `Obligation.capacity` and extends the node's `booked_until`. Because the slot's seeds are the node and the delivery week, one node backs at most one obligation per week.

The obligation, at the PDA `["obl", series, seller]`, then bounds what the seller can offer:

$$S + R \leq K$$

where `K` is `capacity`, `S` is `short_quantity`, the node-weeks already filled, and `R` is `reserved_quantity`, the node-weeks of open fresh asks. Placement adds the ask's quantity to `R` only if the bound still holds. A fill removes the ask's whole posted quantity from `R` and adds the filled quantity to `S`, and the bound is checked again.

## 6.4 Reservation, lock and release

### 6.4.1 Reservation

At placement the seller names a budget `B` with  $C(N(p_a, q), T_{\text{now}}) \leq B \leq \text{free collateral}$ , where `p_a` is the ask's limit and `q` its quantity. `B` moves from free to reserved. If the ask is cancelled, displaced by a better ask, or clears with nothing filled, `B` returns to free in full and its quantity leaves `R`.

### 6.4.2 Lock

When a fill of `x` node-weeks at clearing price `p*` materializes, the program computes

$$L = C(p^* \times 1344 \times x, T_A)$$

with  $T_A$  measured at the round's scheduled cutoff. It requires  $L \leq B$ , removes  $B$  from reserved, adds  $L$  to locked and returns  $B - L$  to free. The obligation records  $L$  in `collateral_locked` and the filled notional in `escrow_notional`. The lock then divides into one share per filled node-week:

$$c_u = \lfloor L / x \rfloor + [u < L \bmod x]$$

where  $u = 0, 1, \dots, x - 1$  is the unit's ordinal within the fill and  $[\cdot]$  is 1 when the condition holds and 0 otherwise; the remainder goes one base unit at a time to the first ordinals. Each share is stored with its original issuance unit in the delivery allocation registry and travels to the `DeliveryTicket` that eventually draws that unit (§8 Delivery), as `collateral_allocated`.

In the example auction of §5, the fresh ask A2 posted 4 node-weeks at a limit of 2.15 with a budget of 3,524.352110 USDC. Placed 24 hours before the W-4 cutoff, its minimum was 2,939.136067 USDC. It filled 4 node-weeks at 2.20, so  $L$  is 2,982.144093 USDC, 542.208017 USDC returns to free, and the four units carry 745.536024, 745.536023, 745.536023 and 745.536023 USDC. A budget at the bare placement minimum would have been 43.008026 USDC short of  $L$ , and A2 would have been ineligible at 2.20.

#### 6.4.3 Release

Locked collateral returns to free only through the delivery outcome of each unit:

- `settle_ticket` releases the ticket's share less the damages it pays (§11 Settlement).
- `waive_delivery` releases the full share of a unit that no holder requested by notice (§7 Claims and transfer).
- A ticket certified as buyer default settles with zero damages and releases its full share.

These rules keep three identities after every instruction. A seller's reserved collateral equals the sum of the budgets of its fresh asks whose receipts are not yet settled, across all series. An obligation's `collateral_locked` equals the sum of the shares of its units that are neither settled nor waived, and the seller's locked collateral is the sum over its obligations. The vault's token balance is at least free plus reserved plus locked: USDC reaches the balances only through `deposit_collateral`, and leaves the vault only through `withdraw_collateral` and damages, so tokens sent to the vault by a plain transfer are credited to none of them.

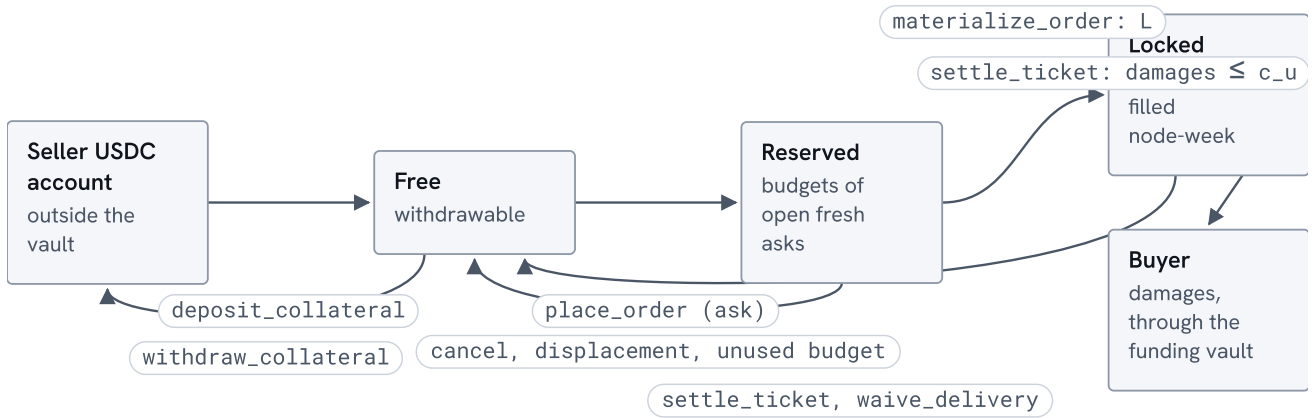


Figure 14 The three collateral balances of a seller and the instructions that move USDC between them.

## 6.5 Backing damages

When a ticket's certified delivery falls below tolerance, the buyer is owed liquidated damages for each undelivered node-hour beyond the 1.68-hour allowance, at the larger of 25 % of the contract price and the reference spot price minus the contract price, for each of the 8 GPUs (§11 Settlement). The damages paid are capped by the ticket's share:

$$D_{\text{paid}} = \min(D, c_u)$$

where  $D$  is the uncapped damages, rounded up to one base unit, and  $c_u$  the ticket's `collateral_allocated`. `settle_ticket` transfers  $D_{\text{paid}}$  from the seller's vault to the series funding vault, where it is paid to the buyer together with the pro-rata refund, reduces locked collateral by  $c_u$  and adds  $c_u - D_{\text{paid}}$  to free. Apart from the seller's own withdrawals of free collateral, no other instruction debits a seller's vault, so a ticket's exposure is bounded by its own share and never reaches the seller's other collateral.

For one of A2's units, with  $c_u = 745.536024$  USDC, a contract price of 2.20 and an example reference spot price of 2.80 USDC per GPU-hour, the damages rate is 0.60 USDC per GPU-hour, since  $2.80 - 2.20$  exceeds 25 % of 2.20:

| Healthy node-hours | Hours beyond allowance | Damages (USDC) | Paid from collateral | Released   |
|--------------------|------------------------|----------------|----------------------|------------|
| 168                | 0                      | 0              | 0                    | 745.536024 |
| 160                | 6.32                   | 30.336         | 30.336               | 715.200024 |
| 100                | 66.32                  | 318.336        | 318.336              | 427.200024 |
| 0                  | 166.32                 | 798.336        | 745.536024           | 0          |

In the last row the full share is paid and nothing is released; the buyer also receives the whole escrowed notional back as refund.



Figure 15 Collateral of the example ask A2 from reservation to settlement, with one of its four tickets delivering 160 healthy node-hours.

## 6.6 Bonds held apart from collateral

Two other USDC deposits sit in their own vaults and never mix with seller collateral. A buyer who files a fault or objects to access posts a bond of 5 % of the ticket's escrowed notional, rounded up, into a vault at `["fault-bond", dispute]`; the bond goes to whichever side the dispute resolves for (§10 Faults, disputes and arbitration). Inspectors post a bond into a vault at `["inspector-bond-vault", inspector]`, recorded in `["inspector-bond", inspector]`; accreditation, acceptance and probing require it to hold at least `Config.inspector_bond_minimum` with no withdrawal requested, and it returns to the inspector through a delayed exit (§3 Participants and eligibility). Neither bond can pay damages, and seller collateral pays nothing but damages.

## 7 Claims and transfer

A claim is the long side of a filled node-week. Each series has one Token-2022 mint with zero decimals, and one token is the right to request delivery of one node-week of that series.

Buyers and traders hold claims, move them between eligible wallets and sell them back into later auctions; before notice, a holder burns a claim to open a delivery ticket.

Claims let a long position change hands without the seller's involvement and without new issuance, while every holder stays inside the eligibility perimeter. The token is a position token for node-weeks: holding it carries no fee, moving it adds no open interest, and it stops transferring at notice. The seller's short side is not a token; it is an `Obligation` account bound to the seller.

### 7.1 The claim mint

`init_claim_mint` creates the mint at the PDA `["mint", series]`, after `create_series` has fixed that address. The `Series` PDA is the mint authority, the freeze authority, the permanent delegate and the authority of every extension, so only market instructions signed by the series can mint, freeze, thaw, pause, update metadata or close the mint.

| Extension                        | Setting                             | Effect                                                         |
|----------------------------------|-------------------------------------|----------------------------------------------------------------|
| <code>TransferHook</code>        | Program <code>hourglass_hook</code> | Every transfer runs the compliance checks below                |
| <code>DefaultAccountState</code> | Frozen                              | Every new token account starts frozen                          |
| <code>MetadataPointer</code>     | The mint itself                     | Metadata is stored in the mint account                         |
| <code>TokenMetadata</code>       | Name, symbol, URI, five fields      | Identifies the series, its specification and window            |
| <code>Pausable</code>            | Series PDA                          | A paused series halts transfers                                |
| <code>MintCloseAuthority</code>  | Series PDA                          | Only the series can close the mint                             |
| <code>PermanentDelegate</code>   | Series PDA                          | Retirement burns the claims left after their units were waived |

`pause_series(paused)`, signed by the admin, pauses or resumes the mint through the series PDA and sets `Series.paused`. While a series is paused, claims do not transfer, `collect_claims` delivers nothing outward, `place_order` and `clear_auction` reject, and fresh mints from materialization are deferred as `pending_mint` until `collect_claims` catches them up.

The metadata fields are written once at initialization:

|                |                                                      |
|----------------|------------------------------------------------------|
| name           | "Hourglass " + region + " " + delivery_start         |
| symbol         | "HG-H100x8"                                          |
| uri            | "hourglass://series/" + series address               |
| series         | series address                                       |
| spec_hash      | SHA-256 of the specification text, 64 hex characters |
| delivery_start | Unix seconds                                         |
| delivery_end   | Unix seconds                                         |
| region         | "NA" or "EU"                                         |

The same instruction creates the series inventory, a Token-2022 account at the PDA ["claims", series] owned by the series, with the `ImmutableOwner`, `TransferHookAccount` and `PausableAccount` extensions, and thaws it. Inventory holds freshly minted claims, claims escrowed by resale asks, and claims credited to receipts but not yet collected. Finally it calls `hourglass_hook` to write the mint's extra-account list, with the series PDA signing.

Claims enter circulation only through fills. When a fresh ask's fill materializes, the program mints its node-weeks into inventory (§5 The auction). A buyer's receipt then carries them as a claim credit, and `collect_claims` moves them to the buyer's own account once the whole round's issuance is minted. Once every fresh fill of the series is materialized and minted, which sealing requires, the supply equals the fresh node-weeks issued minus the claims burned.

## 7.2 Holding claims

A wallet holds claims of a series in its canonical associated token account for that mint. Because the mint's default state is frozen, the account starts frozen, and it can receive or send claims only after the program thaws it. Both thawing instructions require the wallet's `Eligibility` to be approved (KYC, sanctions clear, allowlisted), the current time to be before notice, and the account to be that wallet's canonical account, never the inventory.

`open_claim_account` is how a claim account normally comes into use. It is signed by the participant who pays for the account, and both that payer and the account's owner must be approved buyers or traders. It creates the canonical account through the associated token account program's idempotent `create` if it does not exist, then thaws it under the series PDA if it is frozen. The service places it at the front of the transaction that first needs the account: a bid, a claim sale or a collection, paid by the participant who signs it, and a transfer, where the sender pays to open a missing recipient account. Opening an existing, already thawed account changes nothing, and the instruction refuses a series that is retiring. `thaw_claim_account`, signed by the KYC authority, thaws an account that already exists, and the service sends it for an approved wallet's frozen accounts.

Freezing runs through two instructions. The KYC authority may freeze a participant's account at any time with `freeze_claim_account`. `enforce_claim_freeze` is permissionless and freezes a canonical claim account whenever its owner is no longer approved, and any canonical claim account from notice onward; it returns without effect if the account is already frozen. The service's worker sends it for every account that meets either condition.

Two roles may hold and move claims: buyers and traders. Both may bid, receive transfers, sell into later auctions and request delivery. A trader cannot go short. Selling a node-week that does not exist yet is a fresh ask, which requires the seller role with KYB, a `SellerProfile`, reserved accredited capacity and collateral (§6 Collateral). A resale ask by a buyer or trader moves the offered claims into inventory at placement, so it can never sell more than the wallet holds.

## 7.3 Transfer

### 7.3.1 Moving claims between wallets

A transfer of claims is a plain Token-2022 `TransferChecked` with zero decimals, from the sender's canonical claim account to the recipient's, signed by the sender, carrying the extra accounts that the mint's hook list declares. The market program has no transfer instruction of its own; the hook enforces every rule. The service's `transfer_claims` action builds this transaction with the extra accounts resolved, after checking that notice has not passed, that sender and recipient are approved buyers or traders and that the sender holds enough claims. When the sender's account is frozen, or the recipient's is missing or frozen, the transaction first thaws or opens it with `open_claim_account`, paid by the sender.

### 7.3.2 The transfer hook

`hourglass_hook` is a separate program, so Token-2022's call into the hook never re-enters the market program. Its extra-account list, at the PDA `["extra-account-metas", mint]`, names five accounts: the series, the market `Config`, the market program, and the `Eligibility` PDAs of the source and destination owners. Each `Eligibility` address is derived from the literal `eligibility` and the 32-byte owner field of the source or destination token account, so the accounts checked are always the owners' own.

Token-2022 first applies its own rules: a frozen account cannot send or receive, and a paused mint moves nothing. It then invokes the hook, which checks, in order:

1. The call is the transfer-hook `Execute` instruction with at least ten accounts.
2. The mint belongs to Token-2022, the extra-account list is the canonical one for this mint and is owned by the hook, and `Config` is the market's `["config"]` PDA.
3. `Series` and `Config` are owned by the market program and carry its current state layout version, and the series' `claim_mint` is this mint.
4. Source and destination are Token-2022 accounts of this mint, and the source's `transferring` flag is set, which proves the hook is running inside a live transfer.
5. The series is not paused and the market clock is before notice; otherwise `TransferClosed`.
6. Each end is either the series inventory, whose owner must be the series PDA, or the canonical account of its owner, whose `Eligibility` must be approved and hold the buyer or trader role; otherwise `NotEligible`.

`Eligibility` is read live on every transfer. A wallet whose approval lapses can neither receive nor send claims, even before its account is frozen.

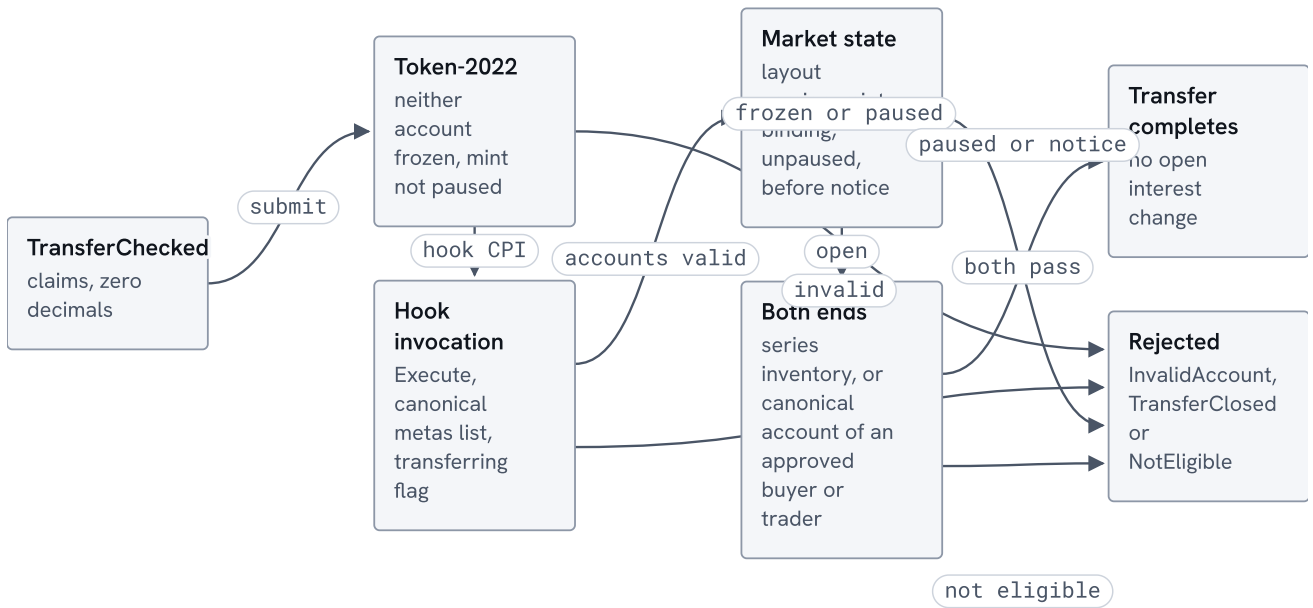


Figure 16 The checks a claim transfer passes, first in Token-2022 and then in the hourglass\_hook program.

### 7.3.3 Resale into later auctions

A holder can sell claims back into any later round of the same series with a resale ask (§5). The claims move into inventory at placement, through the hook. If the ask fills  $x$  of  $q$  node-weeks at the clearing price  $p^*$ , the seller of the claims receives  $p^* \times 1344 \times x$  from the buyers' deposits, and the remaining  $q - x$  claims are credited back to its receipt. Resale creates no obligation and adds no open interest: the escrow behind those node-weeks stays with the obligation that issued them, at the original issuance price. Buyers who take resale supply collect it from inventory with `collect_claims` like any other fill.

`collect_claims` delivers claims outward only while the series is unpaused and before notice, to an approved owner's thawed account. A holder does not have to collect a credit before converting it: `request_delivery` can burn it directly from inventory, as described next.

## 7.4 Notice

Notice is the instant  $W - 72$  h. It ends the life of a claim as a tradable token. Transfers stop at notice ( $t < t_N$  in the hook), outward collection stops at notice, and delivery requests are accepted through notice inclusive ( $t \leq t_N$ ). From the first instant after notice, unrequested units can be waived.

```

transfer and collect:   $t < t_N$ 

request_delivery:       $t_{\text{seal}} \leq t \leq t_N$ 

waive_delivery:         $t > t_N$ 

```

where `t_N` is the notice instant and `t_seal` the moment `seal_delivery` sorts the series' issuance units. Sealing is permissionless; it is possible once the final scheduled auction has cleared, or once its cutoff plus grace has passed, and every fresh fill has been materialized and minted (§8 Delivery). With the latest permitted cutoff of 16:00 UTC on Thursday and the maximum grace of 4 hours, at least 4 hours remain for requests before notice on Friday 00:00 UTC.

#### 7.4.1 Requesting delivery

`request_delivery(ticket_index)` converts one claim into one `DeliveryTicket` per transaction. The requester must be approved and hold the buyer or trader role, the allocation must be sealed, and `ticket_index` must equal the series' next ticket counter. The claim burned is either one from the requester's canonical claim account, with the requester signing, or one from inventory against the requester's own settled receipt with a positive `pending_claims`, which then falls by one. If the account is frozen or the series paused, the program thaws or resumes, burns and restores the previous state in the same instruction; a burn does not invoke the transfer hook.

The program, not the requester, chooses which seller's issuance unit the ticket draws, by the proportional-deficit rule of §8, and the ticket takes that unit's original price, its escrowed notional at that price and its collateral share. The ticket is created at the PDA `["ticket", series, ticket_index]`.

#### 7.4.2 Claims not requested

After notice, every issuance unit that no claim was converted into is waived.

`waive_delivery(unit_index)` is permissionless and acts on one unit:

- It burns one claim from inventory, if inventory holds any.
- It releases the unit's original notional from escrow, repaying an active lender first and paying the remainder to the seller, or crediting it to `Obligation.pending_usdc` if the seller's account cannot receive it. No delivery fee is charged.
- It returns the unit's collateral share from locked to free.
- It lowers the obligation's short quantity, the series' open interest and quantity, and the global open interest, and releases the series' cap reservation when the series is empty.

A claim still in a wallet after notice can no longer transfer or be requested, and `enforce_claim_freeze` freezes its account. The prepaid notional of every unrequested unit goes to its seller through waiver, and when the series retires, `sweep_retired_claims` burns the claim from the wallet's account (§4 Series and the calendar). Holding a claim past notice therefore yields neither delivery nor cash, and holders who do not want a node exit before notice, by transfer or by resale. A requested ticket can still end in buyer default when its holder commits no access key by  $W - 60$  h (§8).

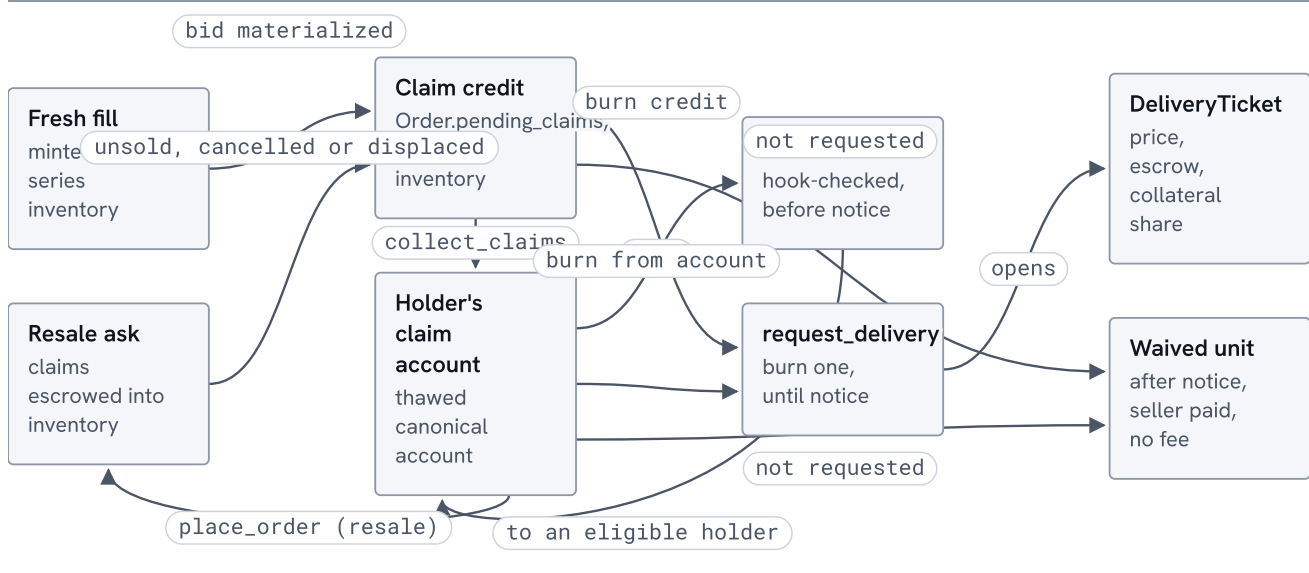


Figure 17 Where a claim can be from minting to delivery request or waiver.

## 7.5 The short side

A seller's position in a series is an **Obligation** at the PDA `["obl", series, seller]`. It records the accredited capacity reserved for the series, the node-weeks reserved by open asks, short, assigned to tickets, waived and settled, the escrowed notional, the locked collateral, any cash credit and the pointer to a lender's pledge.

The obligation is not a token, and no instruction moves it to another wallet: its address is derived from the seller's wallet, and every check on it binds that wallet. A seller's exposure ends unit by unit, through delivery and **settle\_ticket** for requested units and through **waive\_delivery** for the rest. A pledge to a lender (§13 Credit) redirects escrow releases to the lender first without moving the obligation or its collateral. Delivery duties therefore always rest with the seller whose capacity and collateral backed the fill.

## 8 Delivery

Delivery is the part of the contract that turns a claim token into a specific node-week on a specific machine. A claim (§7 Claims and transfer) is a fungible right to one node-week of the series; a delivery ticket is that right after redemption, bound to one seller's obligation, one accredited node, one buyer SSH key and one inspector's acceptance report. Everything that happens to the ticket afterwards — hourly evidence, faults, the certificate and settlement — refers to this account.

The device exists so that the physical side of the forward can be enforced without trusting either counterparty's word. The buyer cannot pick a favourable seller or price, the seller cannot assign a node that is double-booked or out of accreditation, the access key is committed before the node is chosen, and an inspector who is neither party decides whether the node met the grade before the clock starts. This chapter follows a ticket from the redemption registry to confirmed access, and lists every ticket state the program defines.

### 8.1 The delivery registry

Each series owns one raw allocation registry, a program-owned account at the PDA `["delivery-alloc", series]`. It begins with the 8-byte tag `HGDEL006` and holds up to 512 fixed-size issuance units of 64 bytes each, one per node-week that the series has ever issued. The full account is therefore  $8 + 512 \times 64 = 32,776$  bytes. It is created with the series at 10,240 bytes and grown by the permissionless `grow_delivery_registry`, which adds at most 10,240 bytes per call and tops up rent from the caller; three calls bring it to its final size. Every read and write of the registry requires the exact final length and tag.

A unit records where the node-week came from. When a filled ask is materialized after an auction (§5 The auction), `record_issuance` writes one unit per filled node-week, in the seller's order sequence and with ordinals 0, 1, 2, ... within the lot.

```
#[repr(C)]
pub struct AllocationUnit {           // 64 bytes
    pub seller: [u8; 32],             // seller wallet
    pub price: u64,                   // lot clearing price, USDC base units per GPU-hour
    pub collateral: u64,               // this unit's exact share of locked collateral
    pub sequence: u32,                // originating order sequence
    pub ticket_index: u16,            // ticket created from this unit
    pub unit_ordinal: u16,            // position within the lot
    pub state: u8,                    // 0 available, 1 requested, 2 waived
    pub padding: [u8; 7],
}
```

The collateral locked for a lot of  $q$  node-weeks is split across its units so that the shares sum exactly to the locked amount and the remainder is tied to the lowest ordinals:

$$c_k = \lfloor C / q \rfloor + [k < C \bmod q]$$

Here  $C$  is the collateral locked for the whole lot in USDC base units,  $q$  is the lot quantity,  $k$  is the unit ordinal ( $0 \leq k < q$ ), and  $[\cdot]$  is 1 when the condition holds and 0 otherwise.

For example, take a lot of three node-weeks filled at an example clearing price of 2.20 USDC per GPU-hour in a final auction held 80 hours before delivery. The term is 476,191 micro-weeks ( $80 \text{ h} \div 168 \text{ h}$ , rounded up), the notional is  $2,200,000 \times 1,344 \times 3 = 8,870,400,000$  base units, and the locked collateral is 1,837,440,070 base units, or 1,837.440070 USDC (\$6 Collateral). Dividing by three leaves a remainder of 1, so unit 0 carries 612,480,024 base units and units 1 and 2 carry 612,480,023 each.

### 8.1.1 Sealing

Requests draw from a fixed, canonical registry. `seal_delivery` may run once the series has no further auction (`next_auction_at ≥ notice_at`) or once the final auction plus its snapshotted failure grace has passed; that grace is bounded by 4 hours. Sealing also requires every fresh short to be materialized and fully minted: `materialized_fresh_quantity`, `minted_fresh_quantity` and `allocation_count` must all equal `open_quantity`. The instruction then sorts the units by seller key bytes, then source sequence, then unit ordinal, and sets `allocation_sealed`. A sealed series rejects `place_order` and `clear_auction`. A repeated seal is a no-op.

## 8.2 Requesting delivery

A claim owner redeems one node-week with `request_delivery(ticket_index)`. The signer must hold current approval with a buyer or trader role, the series must be sealed, and `ticket_index` must equal the series' `requested_count`, so tickets are numbered 0, 1, 2, ... in request order. The instruction burns one claim token from the owner's canonical Token-2022 account, or one unit of the owner's own settled pending receipt (`Order.pending_claims`) held in the series claim vault. If the claim account is frozen or the mint is paused, the program thaws or resumes it for the burn and restores the previous state in the same instruction. Only one ticket is created per transaction.

The request window closes at the notice instant,  $W - 72$  hours, inclusive: a request at exactly `notice_at` succeeds and one second later fails. Waiver begins strictly after that instant, so no moment allows both a burn and a waiver of the same right.

The ticket is a new account at `["ticket", series, ticket_index]` (index as little-endian u16). It copies the selected unit's seller, price, collateral share, source sequence and ordinal, and records the buyer and the seller's obligation. Its escrow notional is the unit's own price times 1,344 GPU-hours, which is exactly the escrow that unit added when it was issued. The program marks the unit requested, stores the ticket index in it, and increments `assigned_quantity` on the obligation and `requested_count` on the series.

### 8.2.1 The allocation rule

The buyer does not choose a seller. `select_next_unit` assigns each request to the seller whose share of requests so far lags furthest behind its share of issued units. For every seller  $s$  that still has an available unit, the program computes a deficit:

$$D_s = n_s \times (r + 1) - a_s \times N$$

Here  $n_s$  is the number of units seller  $s$  has in the registry,  $a_s$  is the number of those already requested,  $r$  is the number of requests made so far ( `requested_count` ), and  $N$  is the total number of units in the registry. The seller with the greatest  $D_s$  wins; on a tie, the seller that comes first in key-byte order wins, because the comparison is strict and sellers are scanned in sorted order. Within the winning seller, the first available unit in (sequence, ordinal) order is taken.

The rule is integer arithmetic on at most  $513 \times 512$ , so it is exact, and the browser runs the same function ( `selectDeliveryAllocation` ) against the same cross-language vectors. Before selecting, the program re-verifies that the registry is in canonical order, that every state is 0, 1 or 2, and that the number of requested units equals `requested_count` . Every unit is drawn exactly once, and the sequence of draws interleaves sellers in proportion to their issued units.

As an example, suppose a sealed registry holds six units: seller A has three (key bytes lowest), seller B two and seller C one. The table shows the deficits before each request and the seller drawn.

| Request $r$ | $D_A$ | $D_B$ | $D_C$ | Drawn                    |
|-------------|-------|-------|-------|--------------------------|
| 0           | 3     | 2     | 1     | A, unit 0                |
| 1           | 0     | 4     | 2     | B, unit 0                |
| 2           | 3     | 0     | 3     | A, unit 1 (tie, A first) |
| 3           | 0     | 2     | 4     | C, unit 0                |
| 4           | 3     | 4     | —     | B, unit 1                |
| 5           | 6     | —     | —     | A, unit 2                |

Because each ticket inherits the original unit's price and collateral share, the price a ticket settles at is the price at which that node-week was issued, regardless of what the buyer paid for the claim in a later round or on resale. Settlement (§11 Settlement) works on that original basis.

### 8.2.2 Waiver of unrequested units

Units still available after notice are waived. `waive_delivery(unit_index)` is permissionless, runs strictly after `notice_at` , cannot touch a requested unit, and is a no-op for a unit already waived. It pays the unit's original notional out of escrow without a delivery fee — to an active

lender first, then to the seller — frees the unit's collateral and sets the unit's state to 2. §7 Claims and transfer covers what this means for unconverted holders, and §11 Settlement and §13 Credit cover the payment order.

### 8.3 Node assignment

The seller named on the ticket binds a machine to it with `assign_node(expected_access_hash)` no later than  $W - 48$  hours. The node must be the seller's own accredited `NodeRecord`, in the series region, with accreditation that does not expire before delivery end, and with a `CapacitySlot` reserved for this series (§3 Participants and eligibility). The program creates or reuses the booking account `["booking", node, week]` and rejects the call if that booking already names another ticket, so one node serves at most one ticket per delivery week.

Assignment also requires that both parties currently qualify for physical delivery — an approved buyer with a buyer or trader role, and an approved, KYB-verified seller with the seller role — and that no compliance hold is active. The ticket must still be in state 0 with a committed access hash, and `expected_access_hash` must equal that commitment. A transaction prepared against an older key therefore fails instead of assigning a node to a key the seller has not read. On success the ticket records the node and `assigned_at`, moves to state 1, and the node's `booked_until` advances to delivery end. Repeating the same assignment is a no-op.

### 8.4 The access handoff

The buyer reaches the node over SSH, so the seller needs the buyer's SSH public key. The handoff moves that key from the buyer's browser to the seller without placing it on chain and without letting the service read it. Only a hash of the key is committed on chain, by the buyer's wallet.

The seller's encryption key is part of its on-chain profile: `register_seller` stores a nonzero 32-byte X25519 public key in `SellerProfile.encryption_key`, and the ticket view exposes it to the buyer. In the browser, the web app takes the exact bytes of the buyer's OpenSSH public-key line, computes their SHA-256 as the key hash, generates a fresh ephemeral X25519 key pair and a random 24-byte nonce, and seals the key with NaCl box (X25519-XSalsa20-Poly1305) to the seller's public key. The envelope it sends to the API contains only public or encrypted material:

```
{
  ticketId,           // the indexed ticket address
  ciphertextBase64,   // 16 to 8,192 bytes
  keyHash,            // SHA-256 of the SSH public-key bytes, hex
  encryption: 'x25519-xsalsa20-poly1305',
  senderPublicKey,    // ephemeral X25519 public key, 32 bytes
  nonceBase64,        // 24 bytes
}
```

The API's access relay ( `access.handoff` ) validates the envelope, requires an idempotency key of 8 to 128 characters, and checks that the ticket named is the exact indexed ticket and that the caller is its buyer. It serializes writes per ticket and per buyer and idempotency key with database advisory locks. A retry with identical content returns the stored receipt; different content under the same ticket and hash, or under the same idempotency key, is rejected. New material is accepted only from a currently approved buyer and only until the access-commitment deadline; once the ticket is assigned, only the committed key hash is accepted. Each row is scoped to the cluster and its genesis hash and expires at delivery end. The response carries the `set_access_hash` action for the buyer's wallet to sign.

`set_access_hash(access_hash)` writes the commitment to the ticket. The buyer must sign, hold current buyer approval, and have no compliance hold on the ticket; the hash must be nonzero. Replaying the current hash is a no-op. Changing it — a first commitment or a rotation — requires that the ticket is still in state 0 (not yet assigned) and that the time is no later than  $W - 60$  hours, the access-commitment deadline. That deadline sits twelve hours before the assignment deadline, so the seller always has time to fetch and check the final key. The ticket records `access_committed_at` . After assignment the hash is immutable.

On the seller's side, the Provider Agent retrieves the envelope with `access.get` . The relay returns ciphertext only to the ticket's buyer or seller wallet, and only the envelope whose key hash equals the ticket's current on-chain `access_hash` ; inspectors and other wallets receive nothing. The agent opens the box with the seller's X25519 secret key and the envelope's ephemeral public key and nonce, checks that the SHA-256 of the plaintext equals the committed hash, signs `assign_node` with that hash as `expected_access_hash` , and installs the buyer's public key on the assigned node. The secret key never leaves the seller.

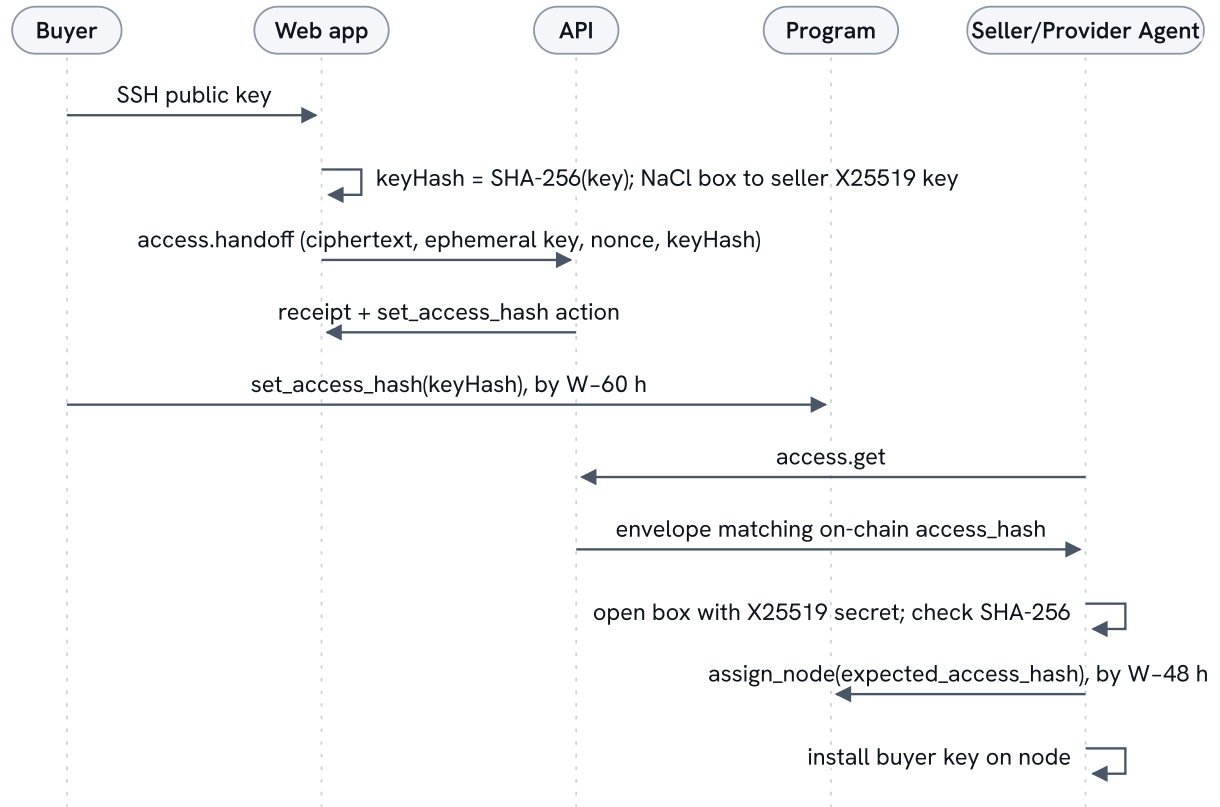


Figure 18 The buyer's SSH public key travels encrypted through the API while only its SHA-256 is committed on chain and pinned by the seller's assignment.

## 8.5 Acceptance

Once the node is provisioned, an inspector tests it for about 30 minutes: the GPU UUIDs reported by the node must match the accredited `NodeRecord`, a short DCGM diagnostic must pass, and NVLink bandwidth must be in range. The inspector records the outcome with `record_acceptance(report_hash, passed)`, where `report_hash` is the 32-byte hash of the signed acceptance report, which is kept off chain.

The acceptance window opens at  $W - 24$  hours and closes at  $W$  plus the series' stored cure window of 4 hours. The signer must be an approved inspector (role bit 8), a current committee member, and neither the buyer nor the seller. Both parties must still qualify for physical delivery, no compliance hold may be active, the report hash must be nonzero, and the ticket may have no open dispute. The ticket must be in state 1 (assigned) or state 4 (objected), with a node and a committed access hash.

Every report increments the ticket's `acceptance_revision` and records the hash, inspector and time. A failed report leaves the ticket where it was, so the seller can fix the node and ask for a retest while the window is open. A passing report sets `accepted_at`, clears any active access objection, and moves the ticket to state 2. The window's extension past  $W$  lets a provider fix a node after a buyer's objection near delivery start and still be reaccepted.

## 8.6 Access confirmation

A passing acceptance starts a two-hour confirmation window. The buyer can confirm with `confirm_access` ; if the buyer does nothing, anyone can call `finalize_access` once `accepted_at + 7,200` seconds has passed, and the program records the confirmation as `automatic ( access_auto_confirmed )`. Both instructions require state 2, a passed acceptance, a committed hash, no active objection, both parties still qualified and no compliance hold. Both move the ticket to state 3 and set `buyer_confirmed_at` ; a repeat after confirmation is a no-op.

If the buyer cannot reach the node, it objects instead with `object_access(index, evidence_hash)` before the two hours expire. An objection is a bonded dispute of cause 4 with the same 5% bond as any fault, it is allowed once per acceptance revision, and it moves the ticket to state 4, where neither confirmation instruction will run. It ends in one of three ways: an independent probe that favours the seller restores confirmed access; a passing reacceptance or an accepted replacement node returns the ticket to state 2 with a fresh two-hour window; or the dispute goes to arbitration. §10 Faults, disputes and arbitration sets out the dispute mechanics.

## 8.7 Replacement nodes

During a dispute the seller can move the ticket to a different machine: `cure_fault` books a replacement node of its own and `accept_cure` records an independent inspector's acceptance of it, both within 4 hours of the dispute's filing. The path is open for every dispute cause: missing heartbeat (0), unreachable (1), GPU health (2), performance (3) and access objection (4). §10 Faults, disputes and arbitration gives the conditions.

An accepted replacement becomes the ticket's node and its acceptance report. For an access objection it also renews acceptance: the revision increments, `accepted_at` restarts, any earlier confirmation is cleared and the ticket returns to state 2, or stays in state 5 if a compliance hold is active. For the other causes the confirmed access is kept. A ticket without a committed key, or one already in buyer default, cannot gain a replacement.

## 8.8 Compliance holds

Eligibility can change during delivery. The KYC authority — and only that key — can record a hold with `mark_compliance_hold` during the delivery window  $[W, W + 168 \text{ h})$  when at least one party no longer qualifies. The hold carries a cause: value 1 when the buyer lacks approval or a buyer or trader role, value 2 when the seller lacks approval, KYB or the seller role, and 3 when both apply. The ticket moves to state 5, and every physical-access instruction — key commitment, assignment, acceptance, confirmation and replacement acceptance — rejects it while the hold lasts.

While a buyer-only hold (cause exactly 1) is active, the program accumulates its seconds per delivery hour into `buyer_hold_seconds` , 168 counters of at most 3,600 seconds each. A change of cause first flushes the interval so far. `release_compliance_hold` requires the KYC authority and both parties qualified again; it flushes the interval up to the earlier of now and delivery end, clears the hold, records `compliance_released_at` and restores the ticket's state from its

fields: buyer default, objected, confirmed, accepted, assigned or requested, in that order of precedence. A ticket that had passed acceptance but was not yet confirmed receives a new acceptance revision and a fresh two-hour window. Holds are recorded only for unsettled, uncertified tickets. How held seconds excuse missing hours is described in §10.

## 8.9 Buyer default

A buyer that redeems a claim but never commits an access key cannot receive a node. After the access-commitment deadline, anyone can record that outcome with `mark_buyer_default`: the ticket must be in state 0 with a zero access hash, no node and no compliance hold. The ticket moves to state 6 and stores `buyer_default_at`. The instruction does not check current buyer eligibility, so a buyer whose approval lapses still reaches this financial exit. A defaulted ticket is certified with disposition 1 and zero healthy hours, and settles by paying the original escrow to the seller with no delivery fee or damages (§11 Settlement).

### 8.10 Ticket states

The `status` byte on `DeliveryTicket` takes nine values. The table lists each with the instructions that enter and leave it.

| Code | State            | Entered by                                                                            | Left by                                                                                                     |
|------|------------------|---------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| 0    | Requested        | <code>request_delivery</code>                                                         | <code>assign_node</code> ; <code>mark_buyer_default</code> ; hold; certificate                              |
| 1    | Assigned         | <code>assign_node</code>                                                              | passing <code>record_acceptance</code> ; hold; certificate                                                  |
| 2    | Access pending   | passing acceptance; accepted replacement                                              | <code>confirm_access</code> , <code>finalize_access</code> ; <code>object_access</code> ; hold; certificate |
| 3    | Access confirmed | <code>confirm_access</code> , <code>finalize_access</code> ; seller-winning objection | hold; <code>post_certificate</code>                                                                         |
| 4    | Access objected  | <code>object_access</code>                                                            | passing acceptance; accepted replacement; seller win; hold; certificate                                     |
| 5    | Compliance hold  | <code>mark_compliance_hold</code>                                                     | <code>release_compliance_hold</code> ; certificate                                                          |
| 6    | Buyer default    | <code>mark_buyer_default</code>                                                       | hold; <code>post_certificate</code> (disposition 1)                                                         |
| 7    | Certified        | <code>post_certificate</code>                                                         | <code>settle_ticket</code>                                                                                  |
| 8    | Settled          | <code>settle_ticket</code>                                                            | terminal                                                                                                    |

A hold release returns the ticket to whichever of states 0, 1, 2, 3, 4 or 6 its fields describe. A certificate can be posted for a ticket in any of states 0 to 6 once delivery and the challenge window are over and no dispute is open. Positive healthy hours require confirmed access, so a ticket that never reached state 3 is certified at zero healthy hours; §10 lists every terminal route.

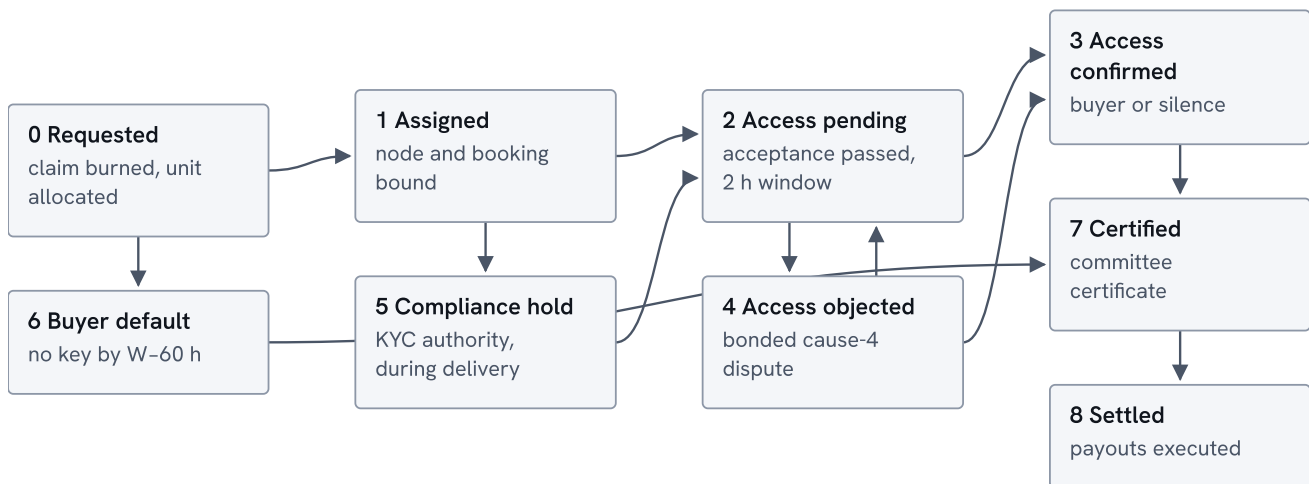


Figure 19 Delivery ticket states 0 to 8 and the transitions between them; the table above names the instruction for each.

## 8.11 Deadlines

All delivery deadlines are fixed offsets from the delivery start  $W$ , a Monday at 00:00 UTC, and are checked against the cluster clock.

| Deadline          | Time                           | Rule                                                                                |
|-------------------|--------------------------------|-------------------------------------------------------------------------------------|
| Notice            | $W - 72 \text{ h}$             | last <code>request_delivery</code> (inclusive); waiver after                        |
| Access commitment | $W - 60 \text{ h}$             | last first commitment or rotation of <code>access_hash</code> ; buyer default after |
| Assignment        | $W - 48 \text{ h}$             | last <code>assign_node</code>                                                       |
| Acceptance opens  | $W - 24 \text{ h}$             | first <code>record_acceptance</code>                                                |
| Confirmation      | <code>accepted_at</code> + 2 h | objection before; <code>finalize_access</code> from                                 |
| Acceptance closes | $W + 4 \text{ h}$              | last <code>record_acceptance</code>                                                 |
| Holds             | $[W, W + 168 \text{ h})$       | <code>mark_compliance_hold</code> allowed                                           |

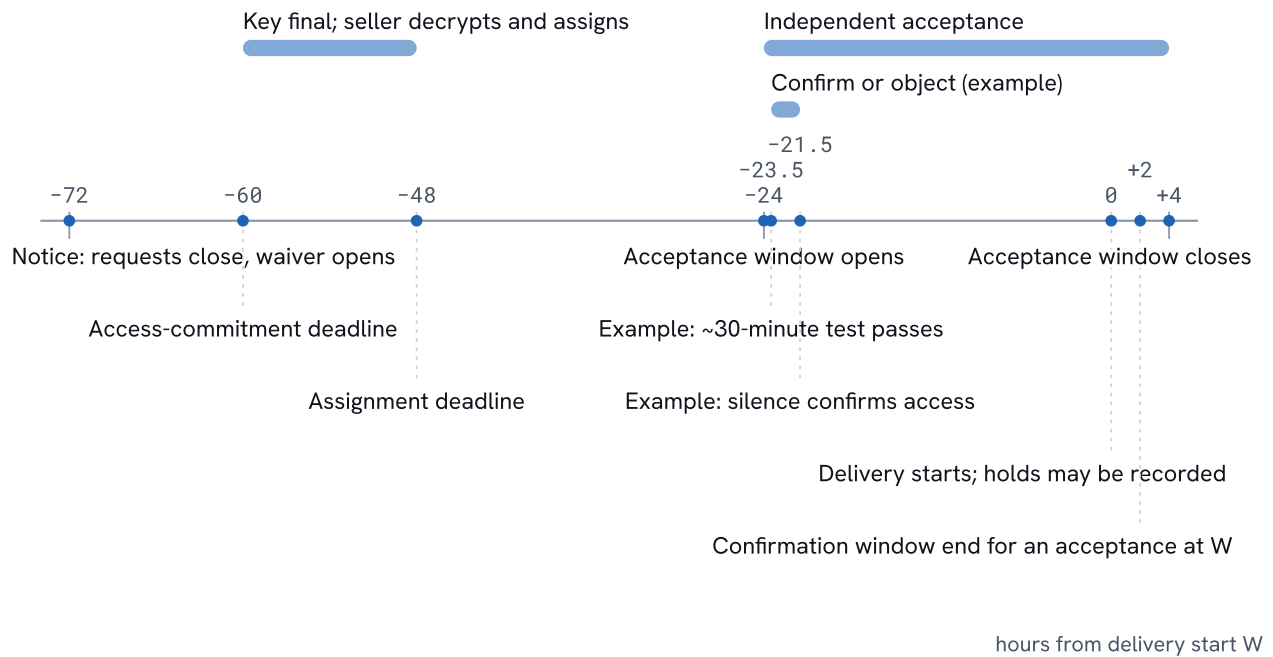


Figure 20 Delivery deadlines from notice to the close of the acceptance window, with an example acceptance at W-24 h.

## 8.12 Service automation

The permissionless transitions do not wait for a party to act. On every snapshot the service worker seals series that are ready, waives units still available after notice, records buyer default for requested tickets without a key after  $W - 60$  hours, finalizes access for eligible tickets after two silent hours, advances unresolved disputes to arbitration, settles certified tickets and collects deferred cash to valid canonical accounts. It submits at most 16 such transactions per tick and retries the rest on the next snapshot; each instruction is idempotent, so a repeated submission changes nothing. Compliance holds are submitted only when the worker holds the KYC authority key.

## 9 Evidence and the committee

A delivery ticket is paid by the hour. During the 168 hours of the delivery window, the system has to establish for every ticket and every hour whether the node was healthy, and it has to do so in a form that both parties can check and that the program can accept without seeing the raw data. This chapter describes the chain of evidence that does this: two signing agents that observe the node every minute, an hourly aggregation that turns their records into one proposal per ticket, a Merkle commitment, a 2-of-3 committee that signs each hour onto the chain, and a final delivery certificate per ticket that settlement consumes.

The design separates observation from judgment. Agents sign what they see and nothing else. The aggregation is a deterministic, replicable function of signed records. The committee reviews and signs; the program checks the signatures, the binding of every signed byte to this deployment, and the arithmetic of the counters. No absence of evidence moves money by itself: a missing heartbeat can only lead to an unhealthy hour through a committee-signed bitmap, and a certificate can only be posted once every dispute on the ticket is closed (§10 Faults, disputes and arbitration).

### 9.1 Healthy node-hour

The unit of evidence is the healthy node-hour. An hour is healthy for a ticket when all eight GPUs are visible, their UUIDs match the accredited node record, no gating XID event occurred, and the node was reachable for at least 55 of the 60 minutes. Gating XIDs are NVIDIA error codes that disqualify an hour; the service configures the set ( `HEALTH_GATING_XIDS` ), by default 48, 64, 74, 79, 95, 119 and 120, and an agent refuses to start without an explicit list.

### 9.2 The agents

#### 9.2.1 The Delivery Agent

The Delivery Agent runs on the delivered node under the buyer's control. Once per delivery minute it reads the node's GPU state and signs one record. A delivery minute is  $\lfloor (t - W) / 60 \rfloor$  for Unix time  $t$  and delivery start  $W$ , and must lie in 0 to 10,079. The agent caches the record it emitted for each minute and returns the same record if asked again, so it never signs two different statements for one minute.

The record carries the ticket, the minute, a hash of the GPU UUID set, the count of gating XID events, a recent Solana slot and two flags. The UUID set hash is the SHA-256 of the eight 32-byte UUID hashes sorted in byte order and concatenated, which is the same value the program stores as `NodeRecord.uuid_hash` at accreditation. `allGpusVisible` is true only when the adapter reports all GPUs visible and exactly eight UUIDs. `reachable` is the agent's own reachability flag. The provider-only fields are false in a buyer record.

The Delivery Agent also exposes the one interface that inspectors use to test a node during a dispute: `GET /.well-known/hourglass/health?nonce=<64 hex characters>`. Any other method or path returns 404, a malformed nonce returns 400, and a failed hardware read returns 503;

responses are never cached. A valid request causes a fresh hardware sample, independent of the minute cache, and returns a signed probe over a fixed 293-byte encoding:

```
"HOURLASS_HEALTH_PROBE_V5" || program || Config || domain || series
|| ticket || nonce (32) || observed_at (u64) || uuid_set_hash (32)
|| gating_xid_count (u16) || all_gpus_visible (u8) || reachable (u8)
|| signer_public_key (32)
```

The probe reads only these health fields. It never reads or returns workload files, SSH keys or application data. Buyers accept this endpoint as a condition of the contract; §10 describes how inspectors use it.

### 9.2.2 The Provider Agent

The Provider Agent runs on the seller's management plane, outside the buyer's operating system. Each minute it reads the baseboard management controller (BMC) for power and network-link state and DCGM for GPU health, and signs one record of kind `provider`. It carries the same GPU identity fields as the buyer's record — UUID set hash, gating XID count, visibility and reachability — plus `powerOn`, `linkUp` and `dcgmClean`. Like the Delivery Agent it emits at most one record per minute.

### 9.2.3 Agent keys

Each agent signs with its own Ed25519 key, not with a wallet. The ticket's parties bind those keys on chain in an `AgentBinding` account at `["agent-binding", ticket]`: the buyer's wallet signs `bind_delivery_agent` and the seller's wallet signs `bind_provider_agent`. A key must be nonzero and the ticket unsettled; once set, a key can be replaced only before delivery start, and rebinding the same key is always accepted. The aggregator takes these bound keys as each ticket's fixed party keys: a record counts only if it is signed by the key bound for its kind.

## 9.3 The minute record

Both agents sign the same canonical encoding. Every record binds the deployment (program, Config account and signature domain, described below), the series, the ticket and the delivery start, so a record cannot be replayed into another deployment, series or ticket.

| offset | bytes | field                                                                |
|--------|-------|----------------------------------------------------------------------|
| 0      | 19    | "HOURGLASS_MINUTE_V5"                                                |
| 19     | 32    | program id                                                           |
| 51     | 32    | Config PDA                                                           |
| 83     | 32    | signature domain                                                     |
| 115    | 32    | series                                                               |
| 147    | 32    | ticket                                                               |
| 179    | 8     | delivery start W (u64 LE, Unix seconds)                              |
| 187    | 1     | kind (0 delivery, 1 provider)                                        |
| 188    | 2     | minute (u16 LE, 0–10,079)                                            |
| 190    | 8     | recent Solana slot (u64 LE)                                          |
| 198    | 32    | GPU UUID set hash                                                    |
| 230    | 2     | gating XID count (u16 LE)                                            |
| 232    | 5     | all_gpus_visible, reachable, power_on, link_up, dcgm_clean (u8 each) |
| 237    | 32    | signer public key                                                    |
| 269    |       | end of signed bytes; a 64-byte Ed25519 signature follows             |

The recent slot ties each record to a point on the chain, which bounds how early a record can have been produced. The minute index and delivery start fix its place in the week.

## 9.4 Collection

`runMinuteAgent` drives each agent. It aligns sampling to minute boundaries after delivery start, signs one record per new minute with a fresh recent slot, stores the record by the SHA-256 of its bytes, hands it to a delivery callback, and stops when the week ends at minute 10,080 or when its caller cancels it. Storage is content-addressed: an object can be written once, a second write must carry identical bytes, and stored bytes are checked against their address.

Both agents post their signed records to the service's evidence endpoint, `evidence.submit` ( `POST /v1/evidence` ). The endpoint needs no wallet session, since each record authenticates itself by its signature and bound key. It returns a receipt with an id, the payload hash, the acceptance time and an idempotent-replay flag, so an agent can resend a record after a network failure without creating a duplicate. Parties keep their own copies; any of them can recompute the hourly commitment from the records.

## 9.5 Hourly aggregation

`aggregateHour` turns one hour of signed records for one series into a proposal. Its input is the deployment context, the delivery start, the hour (0 to 167), the series' tickets with their bitmap index, fixed buyer and provider keys, expected UUID set hash and whether a fault is open, and the records. It accepts at most 512 tickets and  $512 \times 120 = 61,440$  records.

Validation is strict and runs before any proposal:

- every record's Ed25519 signature verifies over its canonical bytes;
- its context equals the hour's deployment context and delivery start, and its ticket is one of the series' tickets;

- its minute lies inside the hour ( $\lfloor \text{minute} / 60 \rfloor = \text{hour}$ );
- its signer is the fixed buyer key for a delivery record or the fixed provider key for a provider record;
- ticket identities and bitmap indices are unique.

A second record for the same ticket, kind and minute is accepted only if it is byte-for-byte identical, signature included; then it is a harmless duplicate. Two different signed records for one party and minute reject the whole hour with an evidence-conflict error for review.

For each ticket the function then classifies the hour. A record has valid hardware when all GPUs are visible, its UUID set hash equals the ticket's expected hash and its gating XID count is zero. Let  $B$  be the ticket's buyer records,  $b$  the number of them with valid hardware that are reachable, and  $p$  the number of provider records with valid hardware, reachable, powered on, link up and DCGM clean.

| Source            | Condition                                                                                                                  | Healthy |
|-------------------|----------------------------------------------------------------------------------------------------------------------------|---------|
| Buyer heartbeats  | no buyer record with invalid hardware, and $b \geq 55$                                                                     | yes     |
| Provider fallback | no open fault; no invalid buyer record; $b =  B $ ; $p \geq 55$ ; no provider record with invalid hardware or unclean DCGM | yes     |
| Fault process     | any other case                                                                                                             | no      |

The fallback covers buyer silence: if the buyer's agent sent fewer than 55 records but every record it did send was clean, and no fault is open on the ticket, clean provider telemetry carries the hour. A buyer record reporting wrong GPUs, a gating XID or an unreachable node blocks the fallback, as does an open fault. A fault-process result is a proposal that the hour is not healthy and that the buyer may pursue it through a bonded fault; it does not open a dispute, move funds or forfeit a bond. Each proposal also reports the buyer and provider record counts and the number of missing buyer minutes,  $60 - |B|$ .

The healthy proposals form a 64-byte bitmap in which bit  $i$  (byte  $\lfloor i / 8 \rfloor$ , bit  $i \bmod 8$ ) is the ticket with index  $i$ . The function writes a fixed-format summary — version, context, delivery start, hour, bitmap and proposals — and a manifest consisting of the summary plus all deduplicated records in canonical order. The evidence hash is the SHA-256 of the manifest, and the manifest is stored under that hash. The committee reviews the proposals against the manifest before it signs anything.

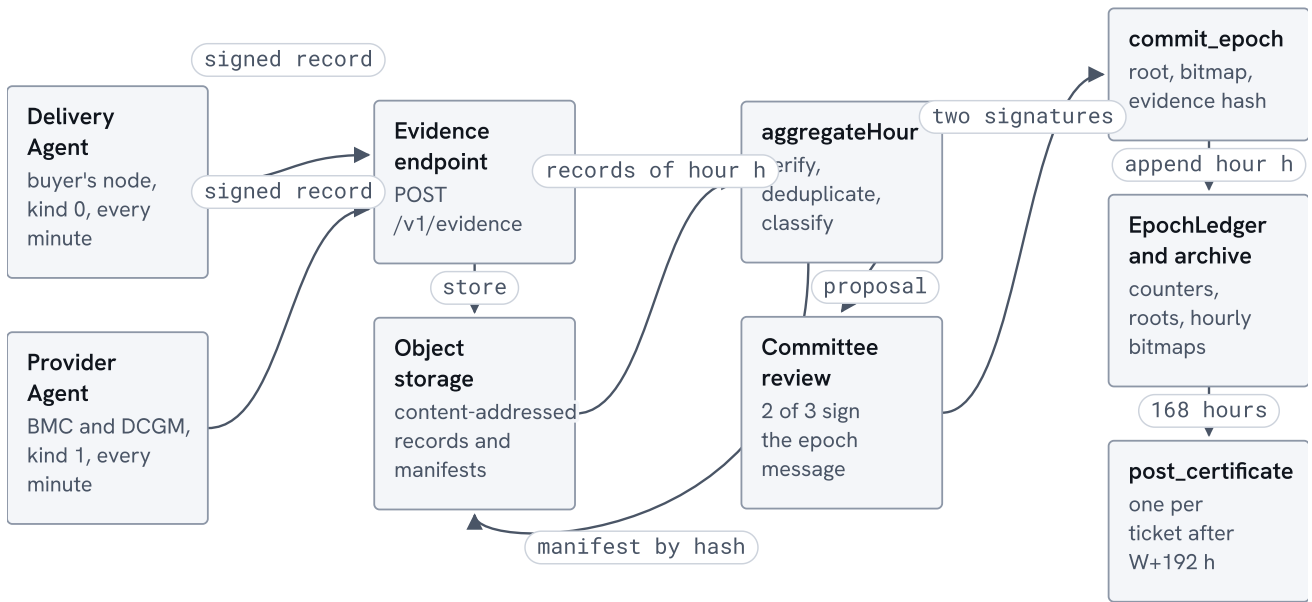


Figure 21 Signed minute records flow from the two agents through hourly aggregation and committee review into the epoch ledger and, after the week, the certificate.

## 9.6 The Merkle commitment

The hour's root commits to the summary and to every record. Leaves and internal nodes are domain-separated SHA-256 hashes:

```

leaf(x) = SHA-256(0x00 || x)
node(l, r) = SHA-256(0x01 || l || r)

```

Here  $x$  is the leaf's bytes,  $l$  and  $r$  are the left and right child hashes, and  $\parallel$  is concatenation. The first leaf is the summary; each following leaf is one record's 269 canonical bytes followed by its 64-byte signature. Records are sorted by ticket, then kind, then minute, so the root does not depend on arrival order. At a level with an odd number of nodes, the last node is paired with itself.

A proof for leaf  $i$  lists one sibling per level. The verifier walks up from the leaf, hashing with the sibling on the left or right according to the index bit, and requires that a missing right sibling at an odd level be the node itself. A party holding its own copy of a record can therefore prove that record's inclusion in the committed hour without the rest of the manifest.

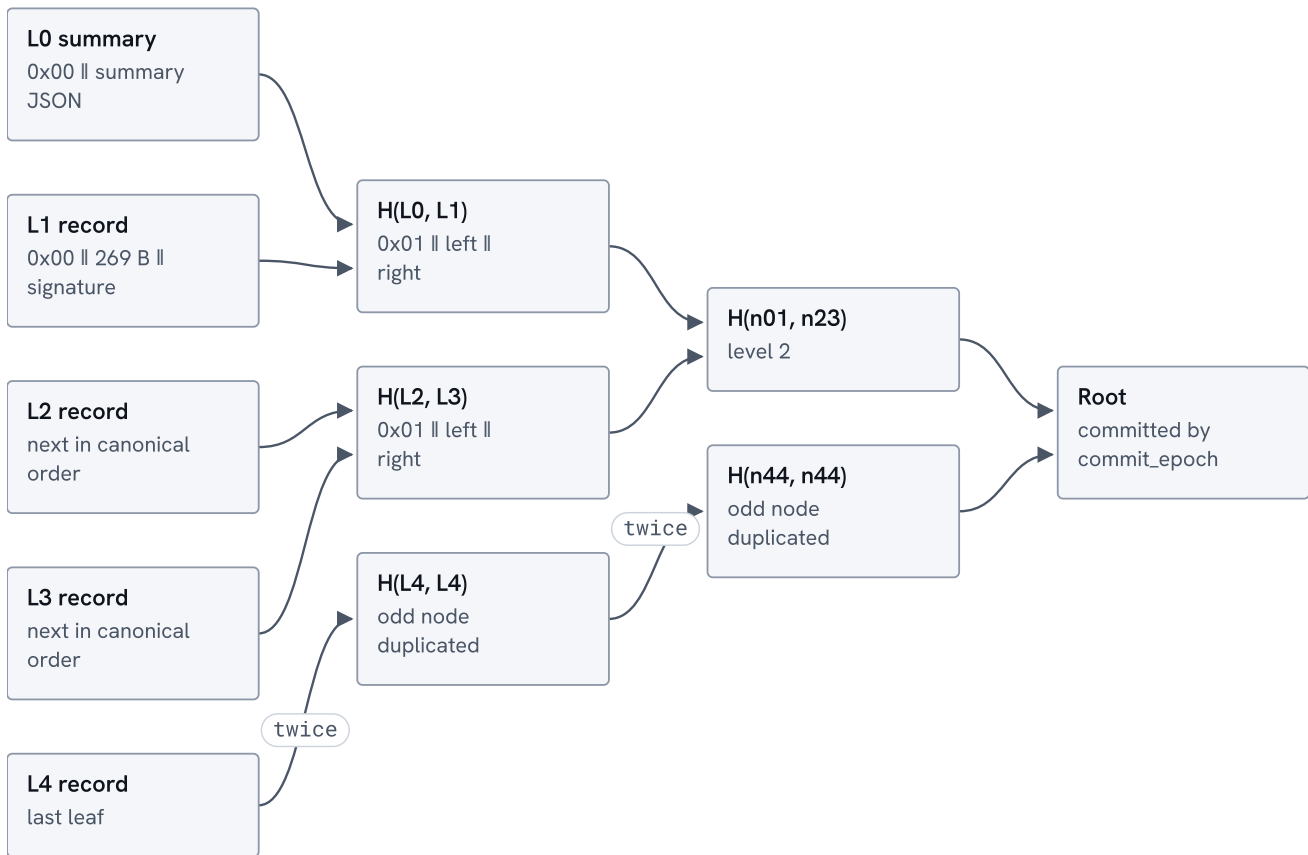


Figure 22 A five-leaf hour tree: the summary and four records, with the odd node duplicated at each level.

## 9.7 Committing an epoch

`commit_epoch(hour, root, healthy_bitmap, evidence_hash)` records one closed hour for one series, once. The program requires a sealed series, an hour below 168 and a current time no earlier than  $W + (\text{hour} + 1) \times 3,600$  seconds, so an hour can be committed only after it has ended. Hours commit strictly in sequence: the hour must equal the ledger's `next_hour`, a lower hour fails as already committed, and a higher one fails as out of order. The root and evidence hash must be nonzero, and no bit may be set at or above the series' `requested_count`.

The committee signs the exact 276-byte epoch message:

```
"HOURLASS_EPOCH_V7" (18) || program (32) || Config (32) || domain (32)
|| series (32) || hour (u16 LE) || root (32) || healthy_bitmap (64)
|| evidence_hash (32)
```

Two committee signatures over that message travel in one native Ed25519 instruction placed immediately before the Hourglass instruction. The two signers share a single copy of the message, which keeps the transaction within Solana's 1,232-byte packet:

| offset | bytes | content                                                          |
|--------|-------|------------------------------------------------------------------|
| 0      | 1     | signature count = 2                                              |
| 1      | 1     | padding = 0                                                      |
| 2      | 14    | offsets, signer 0: signature 62, key 30, message 222, length m   |
| 16     | 14    | offsets, signer 1: signature 158, key 126, message 222, length m |
| 30     | 32    | public key 0                                                     |
| 62     | 64    | signature 0                                                      |
| 126    | 32    | public key 1                                                     |
| 158    | 64    | signature 1                                                      |
| 222    | m     | message                                                          |

Each offset entry also carries the instruction index 0xFFFF for signature, key and message, meaning "this instruction". For an epoch the instruction data is 222 + 276 = 498 bytes.

The Ed25519 precompile verifies the signatures cryptographically as part of the transaction; it cannot be called from a program. Hourglass then checks what was verified, through the instructions sysvar ( `verify_committee_signatures` ). It loads the instruction immediately before itself and requires the Ed25519 program id, no accounts, one or two signatures, offsets that point inside that same instruction, a message length and content exactly equal to the expected message, and public keys that are two distinct members of the current three-key committee. Two single-signature Ed25519 instructions immediately preceding the Hourglass instruction are also accepted. The committee itself must be three distinct nonzero keys.

On success the program adds 1,000,000 micro-hours to the counter of every ticket whose bit is set, stores the root, marks the hour committed, advances `next_hour`, extends the evidence chain, appends the bitmap to the hourly archive and emits an `EpochCommitted` event with the series, hour, root, evidence hash and bitmap. The evidence chain makes the whole week's sequence of signed messages one value:

$$E_h = \text{SHA-256}(\text{"HOURLASS}_{\text{EPOCH-CHAIN}_{v7}}" \parallel E_{-(h-1)} \parallel M_h)$$

Here  $M_h$  is the full epoch message for hour  $h$ ,  $E_h$  is the chain value after hour  $h$ , and  $E_{-1}$  is 32 zero bytes. Every precondition is checked before any field changes, and a replayed commit cannot increment a counter twice.

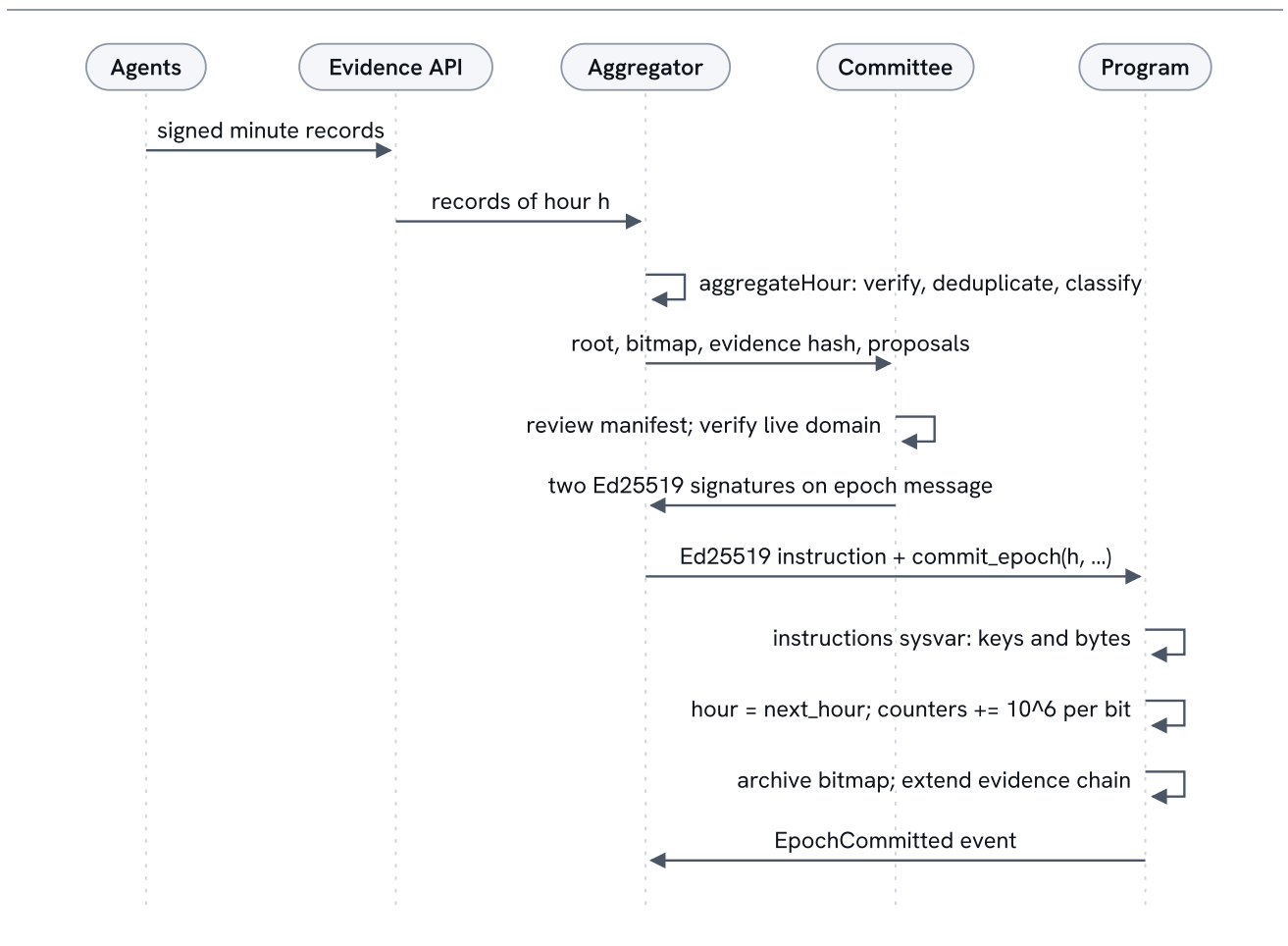


Figure 23 One hour of evidence becomes an on-chain epoch through aggregation, two committee signatures and the `commit_epoch` checks.

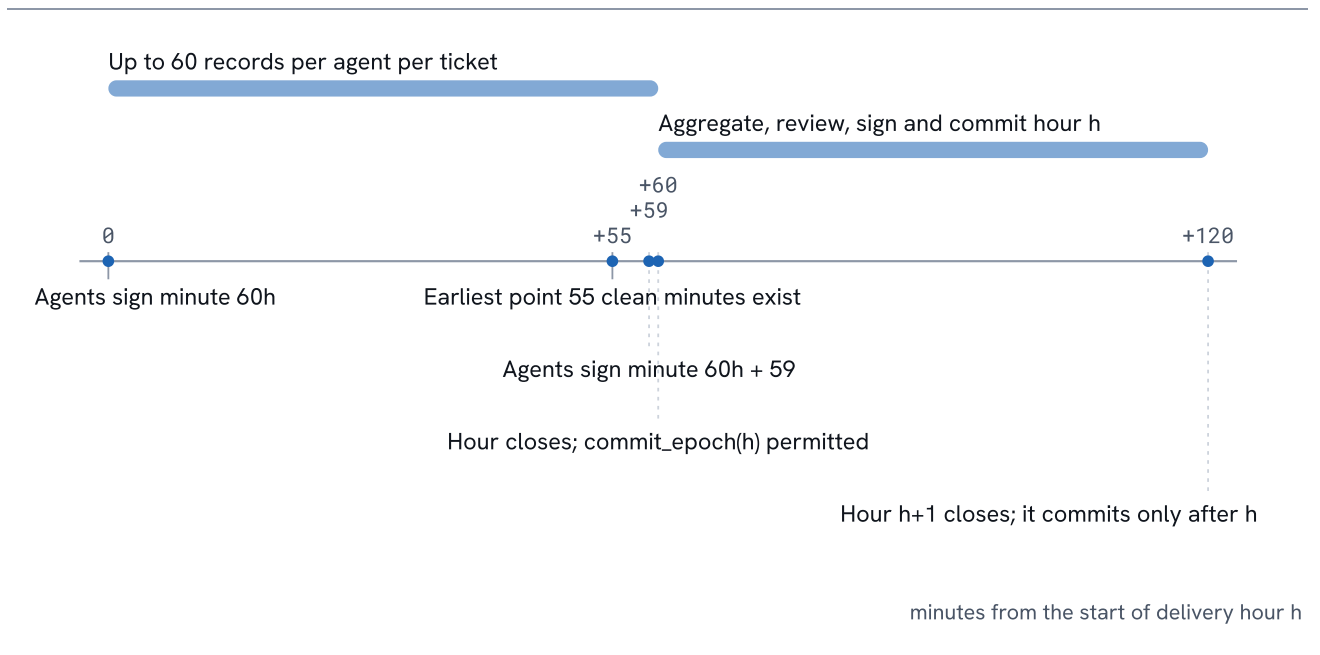


Figure 24 One delivery hour from the first minute record to the point at which its epoch may be committed.

## 9.8 The epoch ledger and the hourly archive

Each series has one zero-copy `EpochLedger` at `["epoch", series]`. It holds the running healthy counter per ticket and the root per hour, sized for the series maximum of 512 tickets and 168 hours. A counter can reach at most 168,000,000 micro-hours, which fits in a `u32`.

| Field                            | Type                         | Bytes |
|----------------------------------|------------------------------|-------|
| Discriminator                    | —                            | 8     |
| <code>series</code>              | Pubkey                       | 32    |
| <code>healthy_hour_micros</code> | <code>[u32; 512]</code>      | 2,048 |
| <code>roots</code>               | <code>[[u8; 32]; 168]</code> | 5,376 |
| <code>evidence_chain_hash</code> | <code>[u8; 32]</code>        | 32    |
| <code>next_hour</code>           | <code>u16</code>             | 2     |
| <code>committed</code>           | <code>[u8; 21]</code>        | 21    |
| <code>bump</code>                | <code>u8</code>              | 1     |
| Total                            |                              | 7,520 |

The counters say how many healthy hours a ticket had, but not which hours. Settlement needs the hours themselves to apply hour-scoped relief, so every commit also appends its bitmap to an immutable archive at `["epoch-health", series]`. The archive is a raw program-owned account: a 48-byte header (tag `HGHEAL06`, series key, next hour as `u16` and bump) followed by 168 bitmaps of 64 bytes, 10,800 bytes in total. It is created at 10,240 bytes by `init_epoch_health` and grown to 10,800 bytes by one permissionless `grow_epoch_health` call. A commit requires the full-size archive and writes only the slot of the hour being committed; earlier bitmaps can never be rewritten. Keeping the bitmaps out of the ledger holds the ledger at its 7,520-byte budget.

The service's epoch indexer rebuilds the per-hour view from the `EpochCommitted` events. It reads the series' transaction history in pages of 1,000 signatures, up to four pages, eight transactions at a time; it keeps only events emitted by invocation frames that succeeded, and only events whose root equals the root the ledger holds for that hour at the indexed slot. Malformed or conflicting events are dropped and the hour stays pending; the indexer never infers health.

## 9.9 The committee

The committee is three `Ed25519` keys stored in `Config.committee`, with a threshold of two. Every committee act — an epoch commit, a delivery certificate and an arbitration award — needs signatures from two distinct current members. The admin sets the committee with `set_committee`, which, like `init_config`, rejects a zero key or a repeated key. The same keys are the inspectors: only a current committee member holding the inspector role (role bit 8) and

current approval can accredit a node, record acceptance, accept a cure or resolve a fault, and never for its own wallet as buyer or seller. Inspectors post a USDC bond into their own program vault with `deposit_inspector_bond` ; Config stores the required minimum, `inspector_bond_minimum` .

The seats are held by Hourglass, as operator, and by independent inspection firms. The service's worker holds the operator's committee keys and runs the aggregator: it builds each hour's proposal, collects the members' signatures and submits the commit.

Members do not sign arbitrary bytes. The API's `committee.payload` returns, for a requested action, the exact bytes to sign, their hash, the encoding ( `raw` for epochs and arbitration, `sha256` for certificates), the full canonical report for certificates, the required signature count of two and the current committee keys. The signing helper accepts only messages that begin with one of the three V7 prefixes, re-reads the deployment context from the chain on every call, and refuses to sign unless the 128 context bytes after the prefix equal that fresh context. For a certificate it also requires the canonical report and checks that the 32 bytes to be signed are its SHA-256. The two returned attestations are verified against the message before the service encodes the Ed25519 instruction.

## 9.10 The signing domain

Every committee message and every minute record carries a 32-byte signature domain that ties it to one deployment on one cluster:

```
domain = SHA-256("HOURGLASSGENESIS-DOMAINV7" || G || P || C)
```

Here G is the cluster's 32-byte genesis hash, P is the Hourglass program id and C is the Config PDA `[ "config" ] . init_config` takes the genesis hash and the domain as arguments, recomputes the domain on chain and stores both, and rejects a mismatch. Because a program cannot read the cluster's genesis hash, the service checks it: on every payload generation, transaction preparation and signing call it fetches the genesis hash from its RPC node, requires the program id to match its manifest, reads the live Config account, and requires that the stored genesis hash and domain equal the values derived from the live genesis. The result is never cached.

## 9.11 The delivery certificate

After the week, each ticket receives one immutable `DeliveryCertificate` at `[ "certificate", ticket ]` . It states the ticket's healthy hours, the reference spot price and its source, the disposition, the evidence mode, any excused unhealthy hours and any signed health adjustment. §11 Settlement turns it into payouts.

`post_certificate` may run only at or after delivery end plus the 24-hour challenge window, `W + 192` hours, and only when the ticket is unsettled and has no open dispute. Its arguments are the nonce (which must be 0), the healthy micro-hours (at most 168,000,000), the reference spot price, the evidence root, the reference source hash, the disposition, the evidence mode, the

excused micro-hours, the health adjustment and the adjustment reason hash. The evidence root and source hash must be nonzero, and the reference spot must not exceed the positive maximum stored in Config, `reference_spot_max`.

The committee signs the SHA-256 of a 318-byte canonical report, not the report itself. Signing the 32-byte digest keeps the Ed25519 instruction at  $222 + 32 = 254$  bytes, which leaves room in the packet for the certificate's larger account list. The program rebuilds the report from the instruction's arguments, hashes it, verifies the two signatures over the digest and stores the digest as `certificate_hash`.

```
"HOURGLASS_CERTIFICATE_V7" (24) || program || Config || domain || series (4 × 32)
|| ticket (32) || nonce (u32) || evidence_root (32) || healthy_hour_micros (u64)
|| reference_spot (u64) || reference_source_hash (32) || disposition (u8)
|| evidence_mode (u8) || excused_unhealthy_hour_micros (u64)
|| health_adjustment (i64) || adjustment_reason_hash (32)
```

| Field         | Values                                                 |
|---------------|--------------------------------------------------------|
| Disposition   | 0 ordinary; 1 buyer default; 2 buyer-caused non-access |
| Evidence mode | 0 epoch-backed; 1 committee-attested                   |

The disposition must agree with the ticket. A ticket in buyer default takes disposition 1 with zero healthy hours and no excusal; any other ticket cannot use disposition 1. Positive healthy hours require a passed acceptance, a confirmed access, an assigned node and no active objection. A ticket must also have a committed access key unless it is in buyer default or under a compliance hold. A repeated post with the same digest is a no-op; a different digest for a certified ticket fails.

#### 9.11.1 Epoch-backed certificates

In the normal mode the ledger must be complete (`next_hour` = 168), the evidence root must equal the ledger's final evidence chain value, and the archive must be complete. The certified healthy hours  $H$  must equal the ticket's counter  $K$  after all 168 hours, unless the committee signs an explicit adjustment:

$$H = K + \Delta$$

Here  $\Delta$  is the signed `health_adjustment` in micro-hours. When  $\Delta = 0$  the reason hash must be zero and  $H = K$  exactly; when  $\Delta \neq 0$  the reason hash must be nonzero and the sum must hold exactly. An adjustment therefore never happens silently, and its reason is part of the signed report.

Excused hours are computed, not asserted. From the archive, the program takes every hour in which the ticket's bit was unhealthy and counts, for that hour, 3,600 seconds if an arbitration award covered it and otherwise the buyer-only compliance-hold seconds recorded for it. The

authorized relief is:

$$X = \lfloor (\sum_{h \text{ unhealthy}} s_h) \times 1,000,000 / 3,600 \rfloor$$

Here  $s_h$  is 3,600 for an arbitration-covered hour and the buyer-held seconds of hour  $h$  otherwise, and  $X$  is in micro-hours, rounded down. Healthy hours are never excused, and a buyer in default is authorized zero. Without an adjustment, the certificate's excused micro-hours must equal  $X$  exactly, and any nonzero excusal requires disposition 2. With an adjustment, excusal must be zero and the authorized relief must be zero, so an adjustment can never discard relief that the archive authorizes. Excused micro-hours can never exceed  $168,000,000 - H$ . §10 gives a worked example.

#### 9.11.2 Committee-attested certificates

If the ledger never completes, the ticket still reaches a certificate. Mode 1 requires that the ledger exists and is incomplete (`next_hour` < 168) and that the current time is at least delivery end plus the challenge window plus the Config's liveness timeout, `manual_certificate_timeout_seconds`, which the service configures to 86,400 seconds by default. A committee-attested certificate allows no excusal, no health adjustment and no reason hash, and it cannot override a completed ledger, because the program loads the canonical ledger and checks its state itself.

In both modes, posting increments the obligation's `certified_unsettled_quantity`, which may not exceed its outstanding short quantity, stores the two actual signers and the post time, and moves the ticket to state 7. `settle_ticket` (§11 Settlement) is then permissionless.

## 10 Faults, disputes and arbitration

A fault is the buyer's bonded statement that a delivered node failed during a stated interval. It opens a `Dispute`, an on-chain process with fixed deadlines that ends in one of three ways: the seller replaces the node, an independent inspector probes it and records a verdict, or the committee records an external arbitration award. Faults exist so that disagreements about delivery are settled by evidence and by people accountable for it, and never by the bare absence of a signal. A missing heartbeat can open a fault; it cannot by itself refund escrow, slash collateral or forfeit a bond.

The same machinery handles the buyer's objection to access at the start of delivery, which is a dispute of its own cause. This chapter describes the dispute account, filing, cure, contest and probe, arbitration, the relief that compliance holds and awards can grant, and the rule that every ticket, disputed or not, reaches a certificate and settlement.

### 10.1 Fault kinds

The dispute's `cause` byte names the kind. Causes 0 to 3 are filed with `file_fault` during delivery and the challenge window; cause 4 is reserved for `object_access`.

| Cause | Kind                           | Instruction                | Alleges                                  |
|-------|--------------------------------|----------------------------|------------------------------------------|
| 0     | <code>missing_heartbeat</code> | <code>file_fault</code>    | the buyer's heartbeats were missing      |
| 1     | <code>unreachable</code>       | <code>file_fault</code>    | the node could not be reached            |
| 2     | <code>gpu_health</code>        | <code>file_fault</code>    | GPU visibility, identity or XID failure  |
| 3     | <code>performance</code>       | <code>file_fault</code>    | the node ran below grade                 |
| 4     | <code>access_objection</code>  | <code>object_access</code> | access failed after a passing acceptance |

### 10.2 The dispute account

A dispute lives at `["dispute", ticket, index]`, with the index as a little-endian u16 equal to the ticket's running `fault_count`, so a ticket's disputes are numbered 0, 1, 2, ... in order. Its bond is held in a USDC token account at `["fault-bond", dispute]` whose authority is the dispute PDA itself. The account stores the parties, the node at filing, the evidence, contest, probe and award hashes, the filing time, the cure and arbitration deadlines, the bond, the cause, status and verdict, the acceptance revision at filing, the alleged hour interval and, after arbitration, the covered-hour bitmap.

| Status | Meaning                     | Reached by                                           |
|--------|-----------------------------|------------------------------------------------------|
| 0      | Open (optionally contested) | <code>file_fault</code> , <code>object_access</code> |

| Status | Meaning               | Reached by                                  |
|--------|-----------------------|---------------------------------------------|
| 1      | Cured                 | <code>accept_cure</code>                    |
| 2      | Provider fault upheld | <code>resolve_fault</code> , verdict 0      |
| 3      | Seller upheld         | <code>resolve_fault</code> , verdict 1 or 2 |
| 4      | In arbitration        | <code>advance_arbitration</code>            |
| 5      | Arbitrated            | <code>resolve_arbitration</code>            |

### 10.3 Filing a fault

The ticket's buyer files with `file_fault(index, evidence_hash, cause, first_affected_hour, end_affected_hour)` . The filing window runs from delivery start  $W$  through delivery end plus the 24-hour challenge window, exclusive: a fault can be filed at  $W + 191\text{ h } 59\text{ min } 59\text{ s}$  but not at  $W + 192\text{ h}$ . The ticket must be unsettled and uncertified, may have no other open dispute, and `index` must equal its `fault_count` ; the evidence hash must be nonzero. The buyer's evidence itself — records, logs, benchmark output — stays off chain and is committed by that hash.

The buyer posts a bond of 5% of the ticket's notional, transferred from its USDC account into the bond vault and rounded up to one base unit:

$$\text{bond} = \lceil N \times 500 / 10,000 \rceil$$

Here  $N$  is the ticket's escrow notional in USDC base units and 500 is `FAULT_BOND_BPS` . At an example contract price of 2.20 USDC per GPU-hour,  $N = 2,200,000 \times 1,344 = 2,956,800,000$  base units and the bond is 147,840,000 base units, or 147.84 USDC. Because every contract price is a multiple of the 10,000-base-unit tick, every ticket notional is a multiple of 13,440,000 base units and the bond is exactly 672,000 base units (0.672 USDC) per tick of price; the upward rounding never changes a ticket's bond.

Filing fixes two deadlines from the filing time  $t$ . The cure deadline is  $t$  plus the series' stored cure window of 4 hours; the arbitration time is  $t$  plus 72 hours.

$$\text{cure}_{\text{deadline}} = t + 4 \times 3,600$$

$$\text{arbitration}_{\text{at}} = t + 72 \times 3,600$$

#### 10.3.1 The alleged interval

Every fault names the delivery hours it concerns as a half-open interval  $[a, b)$  of hour indices, where hour  $h$  is  $[W + 3,600\text{ h}, W + 3,600(h + 1))$ . The interval must be nonempty and may reach only hours that have begun:

$$a < b \leq L, \quad L = \min(\lceil \max(0, t - W) / 3,600 \rceil + 1, 168)$$

Here `a` and `b` are `first_affected_hour` and `end_affected_hour`, and `L` is one past the hour in progress at filing. Evidence may describe any past hour of the week, but it cannot pre-authorize relief for hours that have not started. The interval is written once and never changes. When the buyer's request through the API omits it, the service files for the current delivery hour alone.

For example, a buyer filing at `W + 10 h 30 min` has `L = 11`. It may allege `[8, 11)` for an outage that began in hour 8, but not `[10, 12)`, because hour 11 has not begun. A filing in the last second of the challenge window has `L = 168` and may allege any interval of the week.

An access objection derives its interval instead of declaring it: from the hour in which it is filed through the hour containing its cure deadline, both clamped to the delivery window.

```
a = [(clamp(t) - W) / 3,600],    b = [(clamp(t + 4 * 3,600) - W) / 3,600]
```

Here `clamp(x)` limits `x` to `[W, W + 168 h]`. An objection filed one hour before delivery start covers `[0, 3)`; one filed and curable entirely before `W` covers no hours.

## 10.4 Cure

The seller can end a fault by moving the ticket to another node within the cure window.

`cure_fault` is signed by the seller and names a replacement: its own accredited node, different from the node under dispute, in the series region, accredited through delivery end, with a capacity slot for the series and a booking that is free or already this ticket's. The ticket must have a committed access key and no buyer default, the dispute must be open and the ticket uncertified, and the time must be no later than the cure deadline. The program books the replacement and records it on the dispute.

`accept_cure(report_hash)` completes the cure. An independent inspector — approved, holding the inspector role, a current committee member, neither buyer nor seller, and for an objection not the inspector of the acceptance under objection — tests the replacement and records a nonzero report hash, again no later than the cure deadline. Both parties must still qualify and no compliance hold may be active. The replacement becomes the ticket's node and its acceptance report; for an objection the acceptance is renewed, as described in §8 Delivery. The dispute moves to status 1 and the bond returns to the buyer.

A cure changes the node from the time it is accepted; it does not rewrite the past. The hours already missed remain unhealthy in the committed epochs, and certified health stays authoritative for settlement.

## 10.5 Contest and probe

A seller that disputes the fault signs `contest_fault(evidence_hash)` with its own nonzero evidence hash while the dispute is open and before the arbitration time. A contest is what allows an inspector to find for the seller.

The inspector tests the node through the Delivery Agent's fixed health endpoint (§9 Evidence and the committee). It sends `GET /.well-known/hourglass/health?nonce=` with a fresh random 32-byte nonce to an explicit HTTP(S) origin, refuses redirects, waits at most the configured timeout (no more than 30 seconds) and reads at most 8,192 bytes. It accepts the response only if the version, nonce, ticket and deployment context match, the signature verifies under the buyer's agent key, and the observation is no older than the configured maximum age (no more than 3,600 seconds). The probe then passes if the node is reachable, all GPUs are visible, no gating XID is reported and the UUID set hash matches the accredited set; otherwise it fails.

An endpoint that cannot be reached, or a response that fails verification, gives no result: the probe reports the health as unknown and does not infer who caused the failure. The inspector then attributes it from independent telemetry, chiefly the Provider Agent's BMC records. Every probe produces an evidence hash over its full outcome — context, ticket, nonce, attempt time, result, transport state, reason and signed report — which becomes the report hash of the verdict.

| Finding                                                  | Verdict           | Dispute status | Bond              |
|----------------------------------------------------------|-------------------|----------------|-------------------|
| Probe fails, or node unreachable while BMC shows it down | 0, provider fault | 2              | returned to buyer |
| Probe passes                                             | 1                 | 3              | paid to seller    |
| Buyer blocks the probe                                   | 2                 | 3              | paid to seller    |
| Endpoint unreachable, no attribution                     | none              | stays 0        | to arbitration    |

`resolve_fault(verdict, report_hash)` records the finding. The signer must be an independent inspector on the same terms as for a cure, the dispute must be open and the time before the arbitration time, and the verdict must be 0, 1 or 2 with a nonzero report hash. Verdicts 1 and 2 require a real contest with a contest hash. Verdict 0 is allowed only after the cure deadline has passed, so the seller always keeps its full cure opportunity. A single inspector's verdict moves only the bond and, for an objection, the access state; it never grants damages relief for any hour.

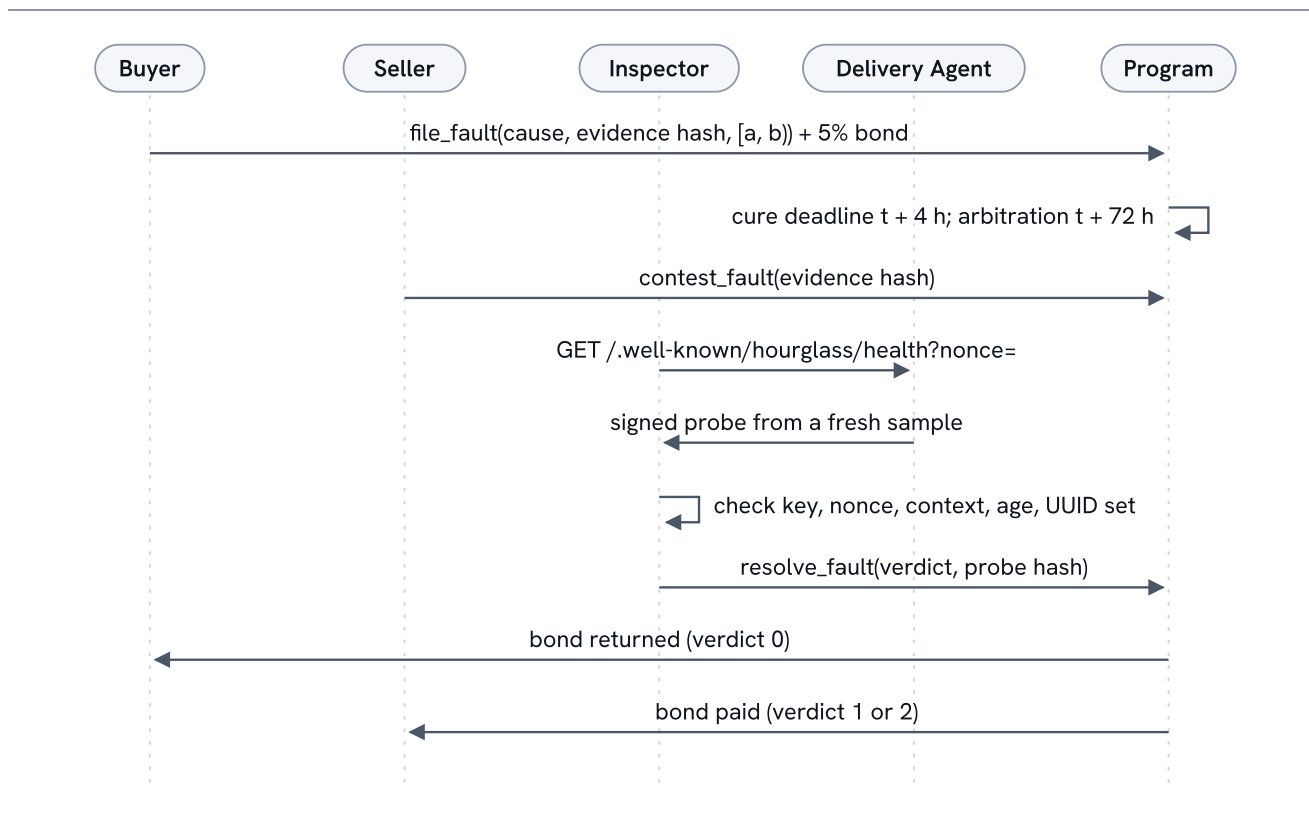


Figure 25 A contested fault resolved by an independent inspector's probe of the Delivery Agent's fixed health endpoint.

## 10.6 Bond payment

Every resolution pays the bond in one step. The bond vault, signing as the dispute PDA, moves the bond into the series funding vault, and the program pays it from there to the winner's canonical USDC token account: the buyer for a cure or verdict 0, the seller for verdicts 1 and 2. If that account does not exist or cannot receive, the amount is credited to the ticket instead ( `pending_buyer_usdc` or `pending_seller_usdc` ), where `collect_ticket_cash` later pays it out; an invalid payee never blocks a resolution. The dispute's bond is then zero and the ticket's open-dispute count returns to zero.

## 10.7 Arbitration

A dispute that is still open at its arbitration time, 72 hours after filing, goes to arbitration under the participant agreement. Anyone can call `advance_arbitration` once that time has passed; it moves the dispute to status 4 and is a no-op if repeated. From then on neither a cure, a contest nor a probe verdict is possible.

The award reaches the chain through the committee. `resolve_arbitration(verdict, award_hash, covered_hours)` requires status 4, a verdict of 0, 1 or 2, a nonzero award hash, and two distinct current committee signatures over the 270-byte arbitration message, checked through the instructions sysvar as for epochs (§9):

```
"HOURLASS_ARBITRATION_V7" (24) || program || Config || domain (3 × 32)
|| series || ticket || dispute (3 × 32) || verdict (u8) || award_hash (32)
|| covered_hours (21)
```

Any account can submit the transaction; the signatures carry the authority. The verdicts mean what they mean for a probe: 0 is a provider fault and returns the bond to the buyer; 1 and 2 find for the seller and pay it the bond, 2 recording that the buyer prevented delivery or inspection. The dispute moves to status 5.

#### 10.7.1 Hour coverage

`covered_hours` is a 21-byte bitmap with one bit per delivery hour: hour  $h$  is bit  $h \bmod 8$  of byte  $\lfloor h / 8 \rfloor$ . It is the only way a dispute can excuse hours in settlement, and only a verdict of 2 may set any bit. Every covered hour must lie inside the fault's immutable interval, must have ended by the time of the award, and must lie within the delivery week:

$$a \leq h, \quad W + 3,600 \times (h + 1) \leq \min(W + 3,600 \times b, t_{\text{award}}, W + 168 \times 3,600)$$

Here  $[a, b)$  is the dispute's alleged interval and  $t_{\text{award}}$  is the time `resolve_arbitration` executes. However late the award arrives, it cannot reach outside  $[a, b)$ : a one-hour fault cannot become a whole-week excuse. The program ORs the bitmap into the ticket's `arbitration_excused_hours`, which the certificate later intersects with the hours that were actually unhealthy (§9).

An award acts only through the bond and the covered hours. The ticket's own settlement stays bounded by its escrow and by the collateral allocated to it, so a dispute's financial exposure never exceeds the ticket's escrow plus that collateral (§11 Settlement).

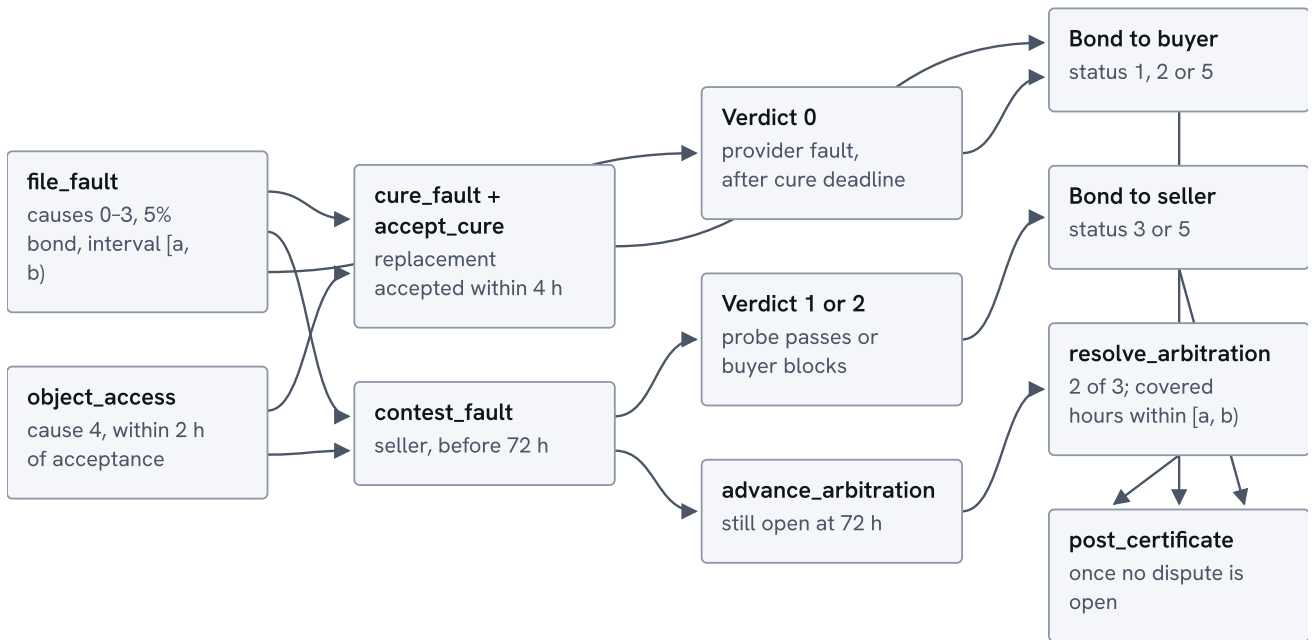


Figure 26 A dispute from filing to closure: cure, contest and probe, or arbitration after 72 hours, before the ticket can be certified.

## 10.8 Access objections

An objection (§8 Delivery) is a dispute of cause 4 with the same bond, the same deadlines and the same cure, probe and arbitration paths, plus three rules of its own. It can be filed only while the ticket is in state 2, unconfirmed, within two hours of the passing acceptance, and only once per acceptance revision. The inspector who resolves it or accepts its cure must differ from the inspector whose acceptance is under objection. And its resolution sets the access state:

- A verdict for the seller, from a probe or from arbitration, clears the objection and confirms access at the resolution time, moving the ticket to state 3, or leaves it in state 5 if a compliance hold is active. It applies only if the acceptance revision has not changed since filing.
- An accepted replacement renews the acceptance: a new revision, a fresh two-hour window and state 2, or state 5 under a hold. A new revision also permits a new objection.
- A verdict for the buyer returns its bond and leaves the ticket in state 4. The seller can still obtain a passing reacceptance until  $W + 4$  hours; otherwise the ticket can be certified only with zero healthy hours, since positive hours require no active objection.

## 10.9 Compliance holds and excused seconds

A compliance hold (§8) suspends physical access when a party stops qualifying. Whether the missing service is excused depends on whose qualification lapsed. Only a buyer-only hold, cause 1, records excusable time, as seconds per delivery hour in the ticket's

`buyer_hold_seconds` ; a hold caused by the seller, or by both parties, records none, and those hours count like any other missing hours.

Relief is the union of buyer-held seconds and arbitration-covered hours, restricted to hours that were actually unhealthy in the archive, with each hour counted once (§9 gives the formula). An excusal requires certificate disposition 2. In settlement, excused hours are still refunded to the buyer pro rata, but they are removed from the hours that attract liquidated damages.

As an example, suppose a ticket's archive shows hours 0 to 119 unhealthy and hours 120 to 167 healthy, and a buyer-only hold was active from  $W + 100$  h to the end of the week. Hours 100 to 119 are both held and unhealthy, so the authorized relief is  $20 \times 3,600$  seconds, or 20,000,000 micro-hours. Hours 0 to 99 were lost before the hold and stay chargeable to the seller, and the held hours 120 to 167 were healthy and need no excuse. If an arbitration award also covered hour 100, already held, and hour 90, unhealthy and not held, the relief would be 21,000,000 micro-hours: hour 100 is not counted twice. Had the hold in hour 100 lasted only 1,800 seconds and the award covered only hour 90, hour 100 would contribute 500,000 micro-hours and the total would be 20,500,000.

## 10.10 Buyer default

A ticket in buyer default (§8) never had a committed key, so it cannot gain a replacement node, and its certificate carries disposition 1, zero healthy hours and zero relief, even if a buyer hold was recorded on it. Settlement then pays the original escrow to the seller, releases the collateral and charges neither a delivery fee nor damages (§11 Settlement).

## 10.11 Terminal paths

Every ticket reaches a certificate and then settlement, whatever happened to it. Two rules make this hold. First, the fault-filing window and certificate eligibility meet exactly at  $W + 192$  hours: no fault can be filed once a certificate may be posted. Second, an open dispute blocks both the certificate and settlement, and every dispute closes by cure, probe verdict or arbitration; `advance_arbitration` is permissionless, so no party can hold a dispute open by inaction. A fault filed in the last second of the challenge window reaches arbitration no later than  $W + 264$  hours.

| State at the end          | Route to certificate                  | Certificate                       |
|---------------------------|---------------------------------------|-----------------------------------|
| 0, key committed, no node | provider failed to assign             | zero healthy hours, disposition 0 |
| 0, no key                 | <code>mark_buyer_default</code> first | disposition 1                     |

| State at the end                | Route to certificate                                   | Certificate                                       |
|---------------------------------|--------------------------------------------------------|---------------------------------------------------|
| 1, acceptance failed or missing | certify actual health                                  | normally zero healthy hours                       |
| 2, access pending               | <code>finalize_access</code> if eligible, else certify | zero healthy hours; disposition 2 if buyer-caused |
| 3, access confirmed             | epochs complete                                        | committed healthy hours                           |
| 4, objection                    | dispute closes first                                   | per resolution (above)                            |
| 5, compliance hold              | release restores state, or certify under hold          | buyer-held hours excused                          |
| 6, buyer default                | none needed                                            | disposition 1                                     |
| 7, certified                    | <code>settle_ticket</code> , permissionless            | —                                                 |
| 8, settled                      | <code>collect_ticket_cash</code> for credits           | —                                                 |

Positive healthy hours always require an independently accepted and confirmed service. A provider that never delivered is certified at zero hours and settles as a full refund with ordinary damages; a buyer that never committed a key settles as buyer default; every other outcome is the committed hourly evidence, adjusted only by a signed reason or excused only within the relief described above.

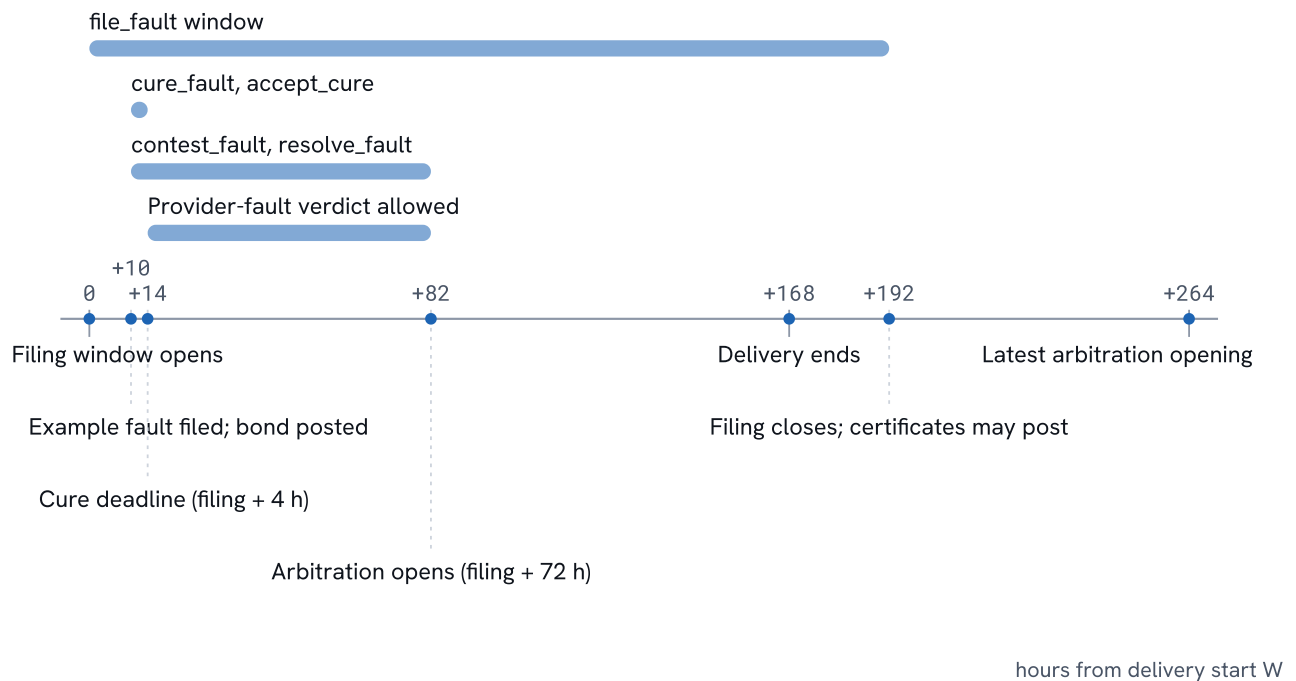


Figure 27 Fault windows for an example fault filed ten hours into delivery, with the filing window and the latest arbitration opening.

## 11 Settlement

Settlement is the terminal financial step of a delivery ticket. One instruction, `settle_ticket`, reads the committee's immutable `DeliveryCertificate` for a ticket and, in a single transaction, pays the seller for certified healthy hours, refunds the buyer for every hour not delivered, charges liquidated damages to the seller's collateral for shortfall beyond the tolerance, takes the delivery fee, repays an attached lender first and unlocks the rest of the collateral. After it runs, the ticket is closed and its escrow, open interest and collateral lock no longer exist.

Settlement exists so that no party depends on another party's arithmetic or timing. The certificate fixes the facts: healthy time, reference price and cause. The program computes every amount from those facts, and any signer may submit the transaction. The same formulas run in the browser as `previewSettlement` and in the program as `math::settlement`, with identical integer arithmetic, so a preview and a payout agree to the base unit.

### 11.1 From certificate to payout

Delivery runs for 168 hours from Monday 00:00 UTC (W). A 24-hour challenge window follows, during which the buyer can still file a bonded fault (§10 Faults, disputes and arbitration).

`post_certificate` is accepted only at or after  $W + 192$  h, only while the ticket has no open fault, and only with two distinct current committee signatures over the canonical report (§9 Evidence and the committee). Posting moves the ticket to status 7 (certified) and increments the obligation's `certified_unsettled_quantity`.

The certificate has two evidence modes. In epoch-backed mode the certified healthy time must equal the ticket's counter in the completed `EpochLedger`, unless the committee signs an exact adjustment with a nonzero reason hash. In manual mode, used only when the ledger is incomplete, the certificate waits an additional configured liveness timeout after the challenge window (86,400 seconds at launch) and may carry neither excused hours nor an adjustment.

`settle_ticket` takes no arguments and accepts any signer. It requires that the ticket's stored certificate is the canonical `["certificate", ticket]` account and that no fault is open. It is idempotent: a ticket with a nonzero `settled_at` returns success without moving funds. The service's worker submits it for certified, unsettled tickets as it finds them; a buyer, seller or lender can submit the same transaction from a wallet.

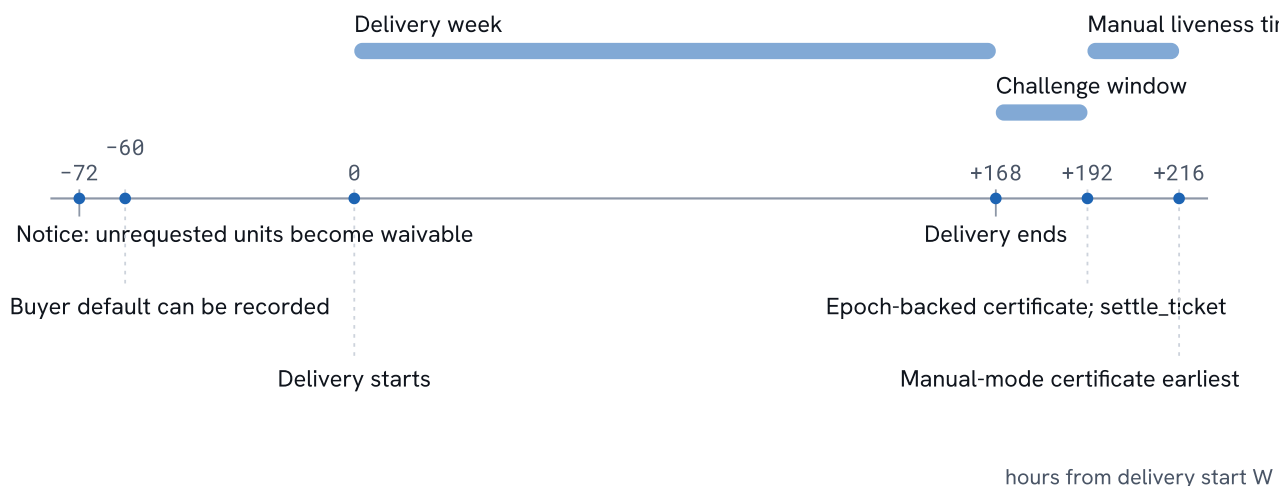


Figure 28 Deadlines that bound settlement, in hours from delivery start W, with the launch manual-certificate timeout.

## 11.2 Inputs

Every input to the payout is either stored on the ticket at delivery request or fixed in the certificate. The ticket carries the price and collateral share of the original issuance lot it was allocated from (§8 Delivery), so the contract price is the seller's lot price, not any later resale price. Series-level economics are snapshotted when the series is created.

| Input             | Symbol | Where it comes from                                            | Unit              |
|-------------------|--------|----------------------------------------------------------------|-------------------|
| Contract price    | P      | <code>DeliveryTicket.price</code>                              | USDC per GPU-hour |
| Ticket escrow     | N      | <code>DeliveryTicket.escrow_notional</code> = $P \times 1,344$ | USDC              |
| Ticket collateral | C      | <code>DeliveryTicket.collateral_allocated</code>               | USDC              |
| Healthy time      | H      | <code>DeliveryCertificate.healthy_hour_micros</code>           | node-hours        |
| Excused time      | X      | <code>excused_unhealthy_hour_micros</code>                     | node-hours        |
| Reference spot    | S      | <code>DeliveryCertificate.reference_spot</code>                | USDC per GPU-hour |
| Disposition       | —      | 0 ordinary, 1 buyer default, 2 buyer-caused                    | code              |
| Tolerance         | —      | <code>Series.tolerance_bps</code> = 9,900                      | bps               |
| Damages floor     | —      | <code>Series.damages_floor_bps</code> = 2,500                  | bps               |
| Delivery fee      | —      | <code>Series.delivery_fee_bps</code> = 250                     | bps               |

Healthy time is measured hourly. For each of the 168 delivery hours the committee commits one epoch whose healthy bitmap has one bit per ticket; a set bit adds 1,000,000 micro-hours to that ticket's counter in the `EpochLedger`, and an immutable archive keeps every hour's bitmap. After the 168th commit the counter is the measured healthy time, between 0 and 168,000,000 micro-hours.

Excused time is the verified buyer-caused part of the shortfall. It is the sum, over hours the archive records as unhealthy for this ticket, of the buyer-only compliance-hold seconds in that hour, or the whole hour where a two-of-three arbitration award covered it. Seconds convert to micro-hours rounding down. Healthy hours and outages before a hold are never excused. A certificate with nonzero excused time must use disposition 2, and in epoch-backed mode the excused amount must equal this archive-derived value exactly.

The reference spot is the committee's attested spot price for the grade, bound to a nonzero source hash and capped by `Config.reference_spot_max`, which is 20 USDC per GPU-hour at launch. A certificate above the cap is rejected.

### 11.3 Per-ticket formulas

The program settles one ticket per instruction, so every formula below is per node-week. In plain units:

$$G = N \times H / 168$$

$$R = N - G$$

$$A = 168 \times (10000 - 9900) / 10000 = 1.68$$

$$E = \max(0, (168 - H) - X - A)$$

$$D = \min(C, \max(0.25 \times P, S - P) \times 8 \times E)$$

$$F = 0.025 \times G$$

G is the escrow released for delivered hours (seller gross), R the buyer's pro-rata refund, A the tolerance allowance in node-hours, E the node-hours eligible for damages, D the liquidated damages, and F the delivery fee. P and S are prices per GPU-hour; the factor 8 converts a per-GPU-hour rate into a per-node-hour rate for the eight-GPU node. H and X are in node-hours, N and C in USDC.

#### 11.3.1 Delivered and missing hours

Missing time is  $168 - H$  node-hours. Every missing hour is refunded: G pays only for certified healthy time and the residual R returns to the buyer, whatever the cause. The healthy fraction  $H / 168$  is the delivery ratio; the program also records it in basis points, rounded down.

#### 11.3.2 Tolerance allowance

The contract tolerates 1% shortfall. The allowance is computed from `tolerance_bps` = 9,900 as  $168,000,000 \times 100 / 10,000 = 1,680,000$  micro-hours, which is 1.68 node-hours or 6,048 seconds. Shortfall within the allowance is refunded but carries no damages. At exactly 166.32 healthy hours damages are zero; one micro-hour less makes one micro-hour eligible.

### 11.3.3 Liquidated damages

Damages accrue per eligible node-hour at the greater of two per-GPU-hour rates: the floor, 25% of the contract price, and the spread of the reference spot over the contract price. Eligible hours are the missing hours left after subtracting verified buyer-caused hours and then the 1.68-hour allowance. The rate is held as a rational numerator over 10,000 until the final amount, so the floor and the spread are compared without rounding.

Damages are paid only from the collateral allocated to the ticket:  $D$  never exceeds  $C$ , and the uncapped figure is recorded separately with a `cap_hit` flag. The amount at stake on one ticket is therefore bounded by its escrow plus its collateral.

### 11.3.4 Delivery fee and seller net

The delivery fee is 2.5% of delivered escrow  $G$ , charged once at settlement and moved to the fee vault (§12 Fees). The seller's net escrow is  $G - F$ . No trade fee is charged at settlement:

`settle_ticket` passes a trade fee rate of zero, because the buyer's acquisition fee was charged when its order materialized.

### 11.3.5 Collateral release

The ticket's collateral less damages,  $C - D$ , is released inside the seller's collateral vault: the amount moves from `locked_collateral` to `free_collateral` on the `SellerProfile`, where the seller can withdraw it or reuse it for new asks. Released collateral never enters the cash waterfall, so it is never paid to a lender.

## 11.4 Rounding

All amounts are integers in USDC base units (six decimals) with 128-bit intermediates. Healthy and excused time are integer micro-hours. Each rounding direction is fixed:

$$G = \lfloor N \times h / (168 \times 10^6) \rfloor$$

$$R = N - G$$

$$D^* = \lceil \max(P \times 2500, \max(0, S - P) \times 10000) \times 8 \times e / 10^{10} \rceil$$

$$D = \min(D^*, c)$$

$$F = \lfloor G \times 250 / 10000 \rfloor$$

Here  $N$ ,  $P$  and  $S$  are in base units,  $c$  is the ticket collateral in base units,  $h$  and  $e$  are healthy and eligible micro-hours,  $D^*$  is the uncapped damages amount, and the divisor  $10^{10}$  removes the basis-point scale ( $10^4$ ) and the micro-hour scale ( $10^6$ ). Seller gross and the delivery fee round down; the refund is the exact escrow residual, so any rounding remainder of escrow goes to the buyer; damages round up, in the buyer's favor, before the collateral cap.

| Amount               | Rounding              | Favors           |
|----------------------|-----------------------|------------------|
| Seller gross G       | down                  | buyer, through R |
| Refund R             | exact escrow residual | buyer            |
| Damages D            | up, then capped       | buyer            |
| Delivery fee F       | down                  | seller           |
| Collateral (at fill) | up                    | buyer            |

At an example contract price of 2.20 USDC per GPU-hour and 166.319999 certified healthy hours, the exact seller gross is 2,927,231,982.4 base units: G is 2,927,231,982, the refund is 29,568,018, the exact fee of 73,180,799.55 becomes 73,180,799, and one eligible micro-hour at the 0.55 USDC floor rate gives an exact 4.4 base units of damages, paid as 5.

A multi-node-week preview rounds each unit exactly as its own ticket would and then sums. Collateral locked for an n-unit fill is split as the quotient to every unit plus one base unit to each of the first remainder units, which is how issuance records each unit's collateral share. By construction the summed preview equals the sum of single-ticket settlements, and  $G + R = N$ ,  $D + \text{released collateral} = C$  and  $(G - F) + F = G$  hold for every input.

## 11.5 Dispositions

The certificate's disposition is a committee-certified cause. It is never inferred from missing heartbeats.

| Disposition     | Seller gross       | Refund  | Damages            | Delivery fee |
|-----------------|--------------------|---------|--------------------|--------------|
| 0 ordinary      | $N \times H / 168$ | $N - G$ | on missing - A     | 2.5% of G    |
| 1 buyer default | N                  | 0       | 0                  | 0            |
| 2 buyer-caused  | $N \times H / 168$ | $N - G$ | on missing - X - A | 2.5% of G    |

Buyer default applies when the buyer never committed an access key. `mark_buyer_default` records it after  $W - 60$  h for a requested ticket with no access hash, no node and no compliance hold. Its certificate must state zero healthy hours and zero excused hours. The seller then receives the full escrow with no delivery fee, collateral unlocks in full, and the buyer receives nothing back.

Buyer-caused nonaccess (disposition 2) keeps every refund and removes only the verified excused hours from the damages base. Provider shortfall outside that scope remains chargeable. At the example price of 2.20 and an example reference spot of 2.60, a week with 12 missing hours of which 10 are excused refunds 211.20 USDC and charges damages on 0.32 node-hours, 1.408 USDC.

A positive healthy time is admissible only for a ticket whose access passed independent acceptance, was confirmed by the buyer or by two hours of silence, has an assigned node and has no active access objection. A ticket that never reached confirmed service is therefore certified with zero healthy hours and settles as a full refund plus damages on its unexcused missing hours, unless the buyer defaulted.

Claims that are never converted into tickets do not reach `settle_ticket`. Strictly after notice, `waive_delivery(unit_index)` releases each unrequested issuance unit: its full original notional goes to the seller (an attached lender first), no delivery fee is charged, and the unit's collateral unlocks. The worker submits waivers for every unrequested unit once notice has passed.

## 11.6 The cash waterfall

`settle_ticket` moves funds in a fixed order inside one transaction. The series funding vault is the transit account; the escrow and collateral vaults are debited only by their program-derived authorities.

1. Escrow vault to fee vault: the delivery fee  $F$ , signed by the `Series` PDA.
2. Escrow vault to the series funding vault:  $N - F$ .
3. Collateral vault to the funding vault: damages  $D$ , signed by the `SellerProfile` PDA.
4. Funding vault to the buyer:  $R + D$ .
5. Funding vault to the lender's repayment vault:  $L = \min(G - F, \text{outstanding principal})$ , when the obligation is pledged (§13 Credit).
6. Funding vault to the seller:  $G - F - L$ .

The ranking that results is: buyer refund and damages first, then the delivery fee, then lender principal, then the seller. The refund is computed from undelivered escrow and damages come from collateral, so neither fees nor the lender can reduce what the buyer receives. The delivery fee is a deduction from the seller's gross, and the lender is repaid only out of the seller's net.

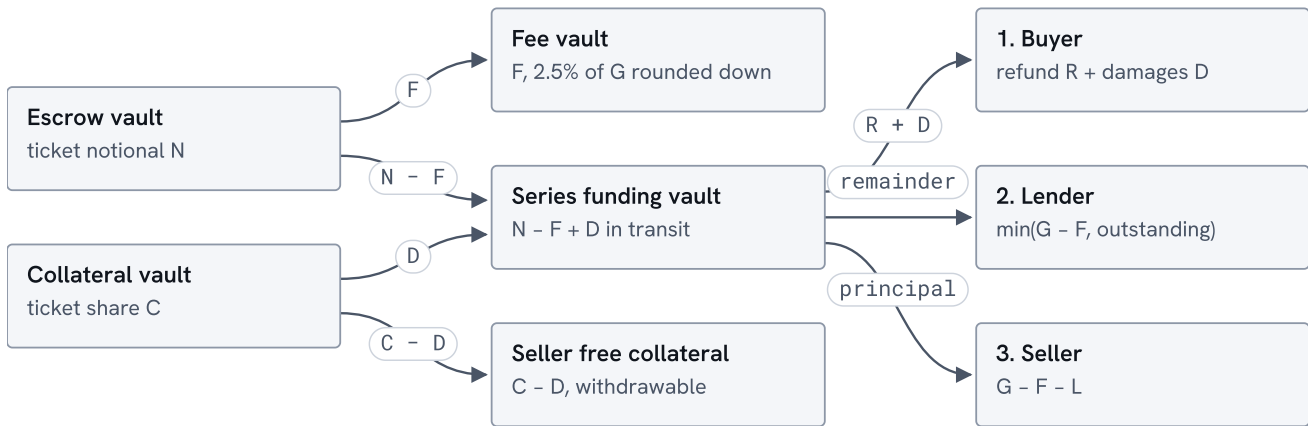


Figure 29 Where each part of a ticket's escrow and collateral goes when `settle_ticket` runs, numbered by payment rank.

Before moving funds the program checks that the recomputed notional equals the ticket's stored escrow, and that the escrow vault holds at least the series' accounted escrow. Tokens transferred into the escrow vault outside the protocol's instructions are recorded as `escrow_surplus` and never change an obligation balance or block a release. The same check runs again after the transfers.

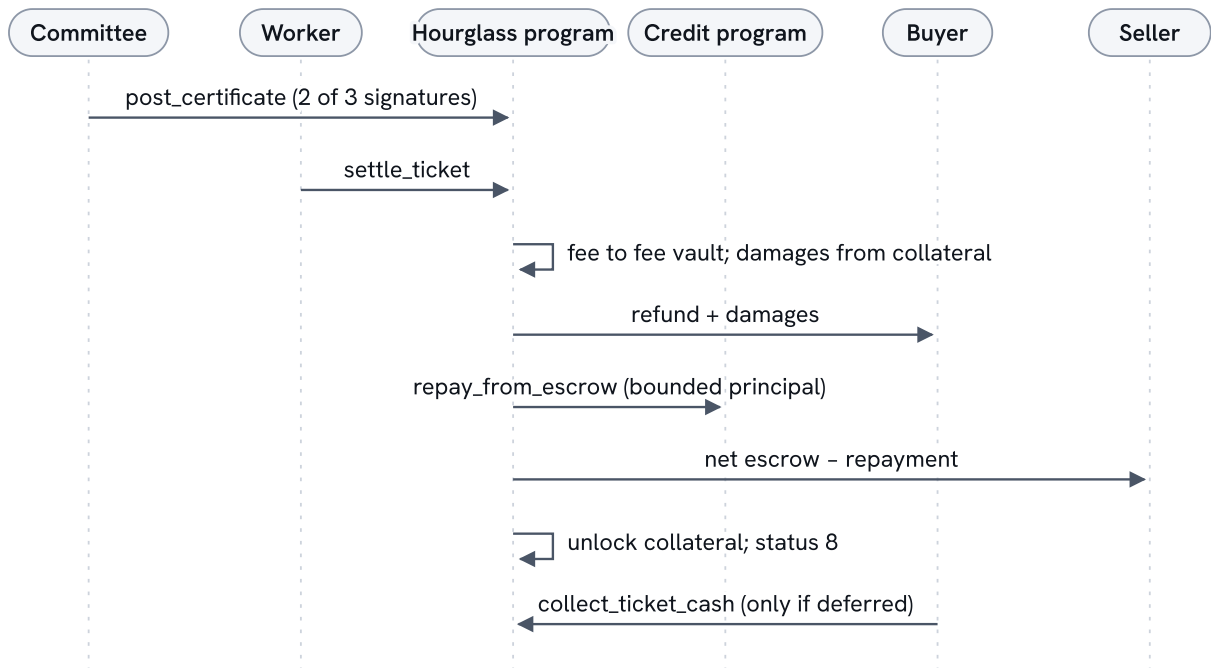


Figure 30 Messages from certificate to collected cash for one pledged ticket.

## 11.7 Durable credits and collection

A payee's account can be closed, frozen or reassigned. Settlement never fails for that reason. Each payment is attempted only to the beneficiary's canonical USDC associated token account, and only if that account is owned by the token program, belongs to the beneficiary, holds the configured USDC mint and is initialized and unfrozen. Otherwise the exact amount stays in the series funding vault and is recorded as a credit owned by the beneficiary.

| Credit field                                    | Created by              | Collected with                         |
|-------------------------------------------------|-------------------------|----------------------------------------|
| <code>DeliveryTicket.pending_buyer_usdc</code>  | settlement, bond payout | <code>collect_ticket_cash(0)</code>    |
| <code>DeliveryTicket.pending_seller_usdc</code> | settlement, bond payout | <code>collect_ticket_cash(1)</code>    |
| <code>Obligation.pending_usdc</code>            | waiver                  | <code>collect_obligation_cash</code>   |
| <code>Order.pending_usdc</code>                 | order materialization   | <code>collect_cash</code>              |
| <code>Pledge.pending_lender_usdc</code>         | lender repayment        | <code>collect_lender_cash</code> (§13) |

Every collection instruction follows one rule. Anyone may collect to the beneficiary's canonical account, so the worker sweeps credits as soon as that account becomes valid. A different destination is accepted only if it is an initialized USDC account owned by the beneficiary and the beneficiary signs. Collection zeroes the credit in the same instruction, so a replay transfers nothing. Credits are independent of eligibility and pause state: a revoked participant still recovers its cash.

## 11.8 Bookkeeping and series close

In the same transaction the program writes the realized amounts onto the ticket (`settlement_seller_gross`, `settlement_refund`, `settlement_damages`, `settlement_delivery_fee`, `settlement_lender_repayment`), sets `settled_at` and moves the ticket to status 8. It reduces the obligation's escrow, locked collateral, short and assigned quantities, decrements `certified_unsettled_quantity`, and increments its settled count. On the series it reduces accounted escrow, open interest and open quantity and increments `settled_count`; the global open interest on `Config` falls by N.

When a sealed series has no open quantity, no open interest and no accounted escrow, and its settled plus waived units equal its original issuance, the series releases its reserved share of the global notional cap exactly once. `settle_ticket` and `waive_delivery` do this when they close the last unit; `release_series_cap` performs the same check on its own. Once its cash credits are collected and the challenge window has passed, the series can retire, which closes its operator-funded accounts and returns their rent (§4 Series and the calendar).

## 11.9 Worked examples

The examples use one node-week at an example clearing price of 2.20 USDC per GPU-hour and an example reference spot of 2.60 USDC per GPU-hour, so the 25% floor (0.55 USDC per GPU-hour) exceeds the spread (0.40). Escrow is  $1,344 \times 2.20 = 2,956.80$  USDC. The fill is assumed at the W - 4 Thursday auction at 16:00 UTC, 3.476191 weeks before delivery, so the collateral rate is  $20\% + 1.5\% \times 3.476191 = 25.2142865\%$  and the ticket's collateral is 745.536024 USDC, rounded up. Every figure below is computed with `previewSettlement`.

### 11.9.1 (a) A clean week

The committee certifies 168 healthy hours. Nothing is refunded and no damages accrue. The delivery fee is 2.5% of 2,956.80, which is 73.92 USDC, and the seller's net escrow is 2,882.88 USDC. All 745.536024 USDC of collateral is released. The buyer paid 2,971.584 USDC at fill, which is the notional plus the 14.784 USDC trade fee.

---

|                   |             |               |
|-------------------|-------------|---------------|
| Ticket escrow     | <div></div> | 2,956.80 USDC |
| Delivery fee      | <div></div> | -73.92 USDC   |
| Seller net escrow | <div></div> | 2,882.88 USDC |

---

Figure 31 Escrow of one fully delivered node-week at an example price of 2.20 USDC per GPU-hour.

### 11.9.2 (b) Six missing hours

The committee certifies 162 healthy hours. Gross escrow is  $2,956.80 \times 162 / 168 = 2,851.20$  USDC; the buyer is refunded 105.60 USDC for all six missing hours. Eligible hours are  $6 - 1.68 = 4.32$ , so damages are  $0.55 \times 8 \times 4.32 = 19.008$  USDC from collateral, and 726.528024 USDC of collateral is released. The delivery fee is 71.28 USDC and the seller's net is 2,779.92 USDC. The buyer receives 124.608 USDC in total, for a net cost of 2,846.976 USDC including the trade fee.

With the same shortfall at an example reference spot of 3.00, the spread (0.80) exceeds the floor and damages become  $0.80 \times 8 \times 4.32 = 27.648$  USDC. Refund, fee and seller net are unchanged, because the reference price affects only damages.

---

|                   |             |               |
|-------------------|-------------|---------------|
| Ticket escrow     | <div></div> | 2,956.80 USDC |
| Refund to buyer   | <div></div> | -105.60 USDC  |
| Delivery fee      | <div></div> | -71.28 USDC   |
| Seller net escrow | <div></div> | 2,779.92 USDC |

---

Figure 32 Escrow of one node-week with six missing hours at an example price of 2.20 USDC per GPU-hour.

### 11.9.3 (c) A cured fault

At hour 40 the node fails and the buyer files a fault, posting the 5% bond of 147.84 USDC. The seller proposes a replacement node and an independent inspector accepts it at hour 42.75, inside the four-hour cure window that ends at hour 44. Hours 40, 41 and 42 are unhealthy; from hour 43 the replacement delivers. The accepted cure resolves the fault in the buyer's favor and returns the bond to the buyer, which clears the open-fault gate for certification.

The certificate states 165 healthy hours. The cure stopped further loss but does not forgive the three hours already missed: the buyer is refunded 52.80 USDC, damages accrue on  $3 - 1.68 = 1.32$  node-hours for 5.808 USDC, the delivery fee is 72.60 USDC and the seller's net is 2,831.40 USDC. Collateral of 739.728024 USDC is released.

### 11.9.4 Summary

| Case             | Refund   | Damages    | Fee   | Seller net |
|------------------|----------|------------|-------|------------|
| (a) 168 h        | 0        | 0          | 73.92 | 2,882.88   |
| (b) 162 h        | 105.60   | 19.008     | 71.28 | 2,779.92   |
| (c) 165 h, cured | 52.80    | 5.808      | 72.60 | 2,831.40   |
| 166.5 h          | 26.40    | 0          | 73.26 | 2,857.14   |
| 0 h, spot 20.00  | 2,956.80 | 745.536024 | 0     | 0          |
| Buyer default    | 0        | 0          | 0     | 2,956.80   |

The 166.5-hour row is inside the tolerance: 1.5 missing hours are refunded with no damages. In the zero-delivery row the example reference spot is set to the launch ceiling of 20.00 USDC per GPU-hour; uncapped damages are  $17.80 \times 8 \times 166.32 = 23,683.968$  USDC; the cap limits them to the ticket's 745.536024 USDC of collateral, and the buyer receives 3,702.336024 USDC against 2,971.584 USDC paid. For three node-weeks under case (b), the preview sums three identical tickets: 316.80 USDC refunded, 57.024 USDC of damages and 8,339.76 USDC of seller net.

## 12 Fees

The programs charge three fees, all in USDC and all on value that has actually changed hands: a trade fee on filled acquisition notional, a delivery fee on delivered escrow and a pledge fee on a lender's advance. A fourth amount, the fault bond, is collected the same way but is a bond: it is held in its own vault and paid back to one of the two parties, never to the fee vault.

Nothing else is charged on-chain. Holding a claim costs nothing

(`HOURLASS_SPEC.fees.heldPositionFee` is 0), and accreditation and inspection are listed in the specification as seller costs passed through at cost, which no program instruction collects.

### 12.1 Schedule

| Fee          | Payer  | Charged on                  | Rate    | Rounding           |
|--------------|--------|-----------------------------|---------|--------------------|
| Trade fee    | bidder | filled notional per order   | 50 bps  | up, once per order |
| Delivery fee | seller | delivered escrow per ticket | 250 bps | down, per ticket   |
| Pledge fee   | lender | advance                     | 50 bps  | up                 |
| Fault bond   | buyer  | ticket escrow               | 500 bps | up; refundable     |

Each rate is fixed when the record it prices is created, so a later configuration cannot change an existing position's cost. `Config` receives the trade and delivery rates from the program constants at `init_config` and the pledge rate of 50 bps. An `Order` snapshots the trade rate at placement, a `Series` snapshots the delivery rate at creation, and a `Pledge` snapshots its fee rate and amount at origination. The fault bond rate is a program constant. Changing any of them requires a reviewed program release.

### 12.2 Trade fee

The trade fee is paid by the side that acquires claims: every bid, whether it fills against fresh issuance or against a resale ask. Sellers of fresh asks and holders reselling claims pay no trade fee. At `place_order` a bid funds its full limit notional plus the fee on that notional:

$$B = Q \times 1344 \times L + [Q \times 1344 \times L \times 50 / 10000]$$
$$T = [Q_f \times 1344 \times P_c \times 50 / 10000]$$

B is the deposit, Q the ordered node-weeks, L the limit price, Q\_f the filled node-weeks, P\_c the clearing price and T the fee actually charged, all in USDC base units. When the order materializes, T moves from the series funding vault to the fee vault and the unused deposit  $B - (Q_f \times 1,344 \times P_c + T)$  returns to the bidder. The fee is computed once on the order's whole fill,

rounded up to a base unit, so a multi-unit order carries at most one base unit of rounding; with tick-aligned prices and the 50 bps rate it is always an exact number of base units. A cancelled, unfilled or evicted bid pays no fee and receives its full deposit.

At an example bid of two node-weeks with a limit of 2.30 USDC per GPU-hour that clears at an example price of 2.20, the deposit is  $6,182.40 + 30.912 = 6,213.312$  USDC. The fill costs 5,913.60 USDC plus a trade fee of 29.568 USDC, and 270.144 USDC is refunded. The fee follows the clearing price, not the limit.

### 12.3 Delivery fee

The delivery fee is 2.5% of the escrow released to the seller for certified healthy hours, charged by `settle_ticket` once per ticket and rounded down (§11 Settlement). It is transferred from the series escrow vault to the fee vault before any other payment and is deducted from the seller's gross, so it never reduces a buyer's refund and is always paid before lender principal. Undelivered hours carry no delivery fee. Waived units and buyer-default tickets release their full escrow to the seller and are charged no delivery fee.

### 12.4 Pledge fee

The pledge fee is 0.5% of a lender's advance, rounded up, paid by the lender in the same transaction as the advance: `pledge_obligation` transfers the principal from the lender's USDC account to the seller and the fee from the same account to the fee vault (§13 Credit). The fee is not added to the seller's debt; only principal is repaid.

### 12.5 Fault bond

The fault bond is security for the buyer's claim, not a charge. The buyer stakes 5% of the ticket's escrow, rounded up, when it calls `file_fault` or `object_access`. The amount moves into a token account owned by the `Dispute` PDA, `["fault-bond", dispute]`, and stays there until the dispute closes.

On resolution the whole bond goes to the side the outcome favors, never to the fee vault. It is paid through the series funding vault under the same pay-or-defer rule as settlement payouts. It returns to the buyer when an independently accepted replacement cures the fault, when a probe finds provider fault after the cure window, or when arbitration upholds the buyer. It is forfeited to the seller when a contested probe finds the node healthy or the buyer blocking, or when arbitration finds for the seller. A seller-winning probe verdict is accepted only for a fault the seller contested and only with a nonzero independent probe report.

### 12.6 The fee vault

All fee cash accrues in a single USDC token account, the fee vault, created by `init_config` at the PDA `["fees", config]` with the `Config` PDA as its token authority. Its address is stored in `Config.fee_vault`, and every instruction that charges a fee constrains its destination to exactly that address. One further inflow reaches it: when a series retires,

`sweep_retired_surplus` moves whatever USDC is left in its funding and escrow vaults after every receipt has been audited, such as an unsolicited transfer, to the same address (§4 Series and the calendar).

## 12.7 \$HOURGLASS

\$HOURGLASS is the protocol's token. Its supply is fixed at 1,000,000,000, and all of it went to one public launch on equal terms: no presale, and no allocation to a team, investors, an airdrop or any reserve, so nothing vests.

The token's link to the market is the fee vault. The protocol keeps every fee the programs charge — the trade fee on filled acquisitions, the delivery fee on delivered escrow, the pledge fee on advances — and shares none of it with liquidity providers, lenders or agents. The fee vault's USDC is reserved for one use: buying \$HOURGLASS on the market and burning it. The vault is a program account whose token authority is the `Config` PDA, and the only instruction that moves USDC out of it is `claim_buyback`, which the program accepts only as the first of three instructions that end in a burn. No key can redirect the fees (§15 Security design).

Burned tokens do not come back, so every node-week the market delivers reduces the supply.

In the worked example below one delivered, pledged node-week adds 99.792 USDC to the fee vault, all of it for buying and burning \$HOURGLASS.

## 12.8 The buyback

The administrator registers the token once with `set_token_mint(keeper)`: the \$HOURGLASS mint, its token program (SPL Token or Token-2022) and the keeper key allowed to start a buyback, which at launch is the worker's key. The program accepts only a fixed-supply mint: initialized, with a positive supply, no mint authority and no freeze authority, and on Token-2022 no extension other than metadata. Once any USDC has been spent, the mint can no longer change. Until a mint is registered, every buyback is refused.

A buyback is one transaction with three top-level instructions of the market program, in this order and with nothing after them:

1. `claim_buyback(sequence, amount, minimum_output)`, signed by the keeper. The sequence must be the next in `Config`; the amount must be positive and no more than the vault holds. It records the custody balances in a new `BuybackReceipt` at `["buyback", config, sequence]` and moves the amount from the fee vault to the USDC account of the buyback authority, a PDA at `["buyback-authority", config]` that no key signs for.
2. `buyback_swap(sequence, data)` calls Jupiter's v6 program, signed by the buyback authority, with the route the keeper supplies. The route may not name the fee vault, `Config` or the receipt.
3. `finish_buyback(sequence)` requires the authority's USDC balance to be back where it was before the claim, so the whole amount was spent, and the authority's \$HOURGLASS balance to have grown by at least `minimum_output`. It burns everything that arrived,

advances the sequence, adds the amounts to `Config`'s totals and closes the `BuybackReceipt`, returning its rent to the keeper.

Each instruction checks the other two through the instructions sysvar — program, instruction name, sequence and, for the finish, every account — so a claim that is not followed by this swap and this burn fails, and with it the whole transaction. The custody accounts are canonical associated token accounts of the authority with no delegate and no close authority.

The worker runs the buyback every 300 seconds once the vault holds at least 100 USDC, in batches of up to 1,000 USDC. It asks Jupiter for a USDC → \$HOURGLASS quote with 1% slippage, uses the quote's minimum as `minimum_output`, and narrows the route (48, then 40, then 32 accounts) until the transaction fits the 1,232-byte limit with the lookup table. On a cluster without a market for the token it records that there is no route and waits. The totals — USDC spent, tokens burned, the last buyback's time and amounts — are public through the API's `buyback.get`.

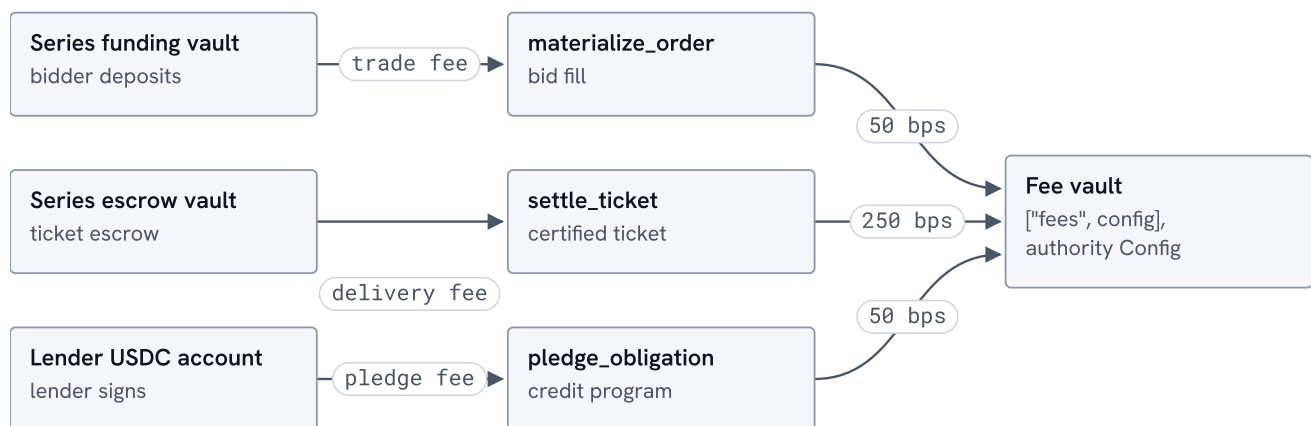


Figure 33 The three instructions that pay fees into the fee vault and the account each fee is debited from.

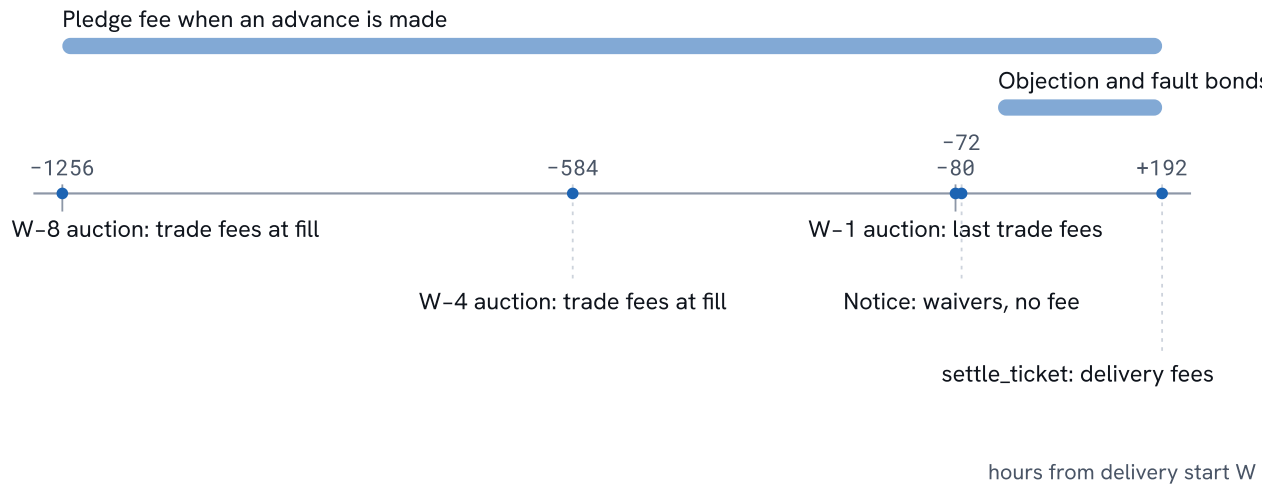


Figure 34 When each fee is charged for a series, in hours from delivery start W with Thursday 16:00 UTC auctions.

## 12.9 Worked example

One node-week clears at an example price of 2.20 USDC per GPU-hour, the seller pledges the obligation for a 75% advance, and the week is delivered in full. The notional is 2,956.80 USDC.

| Item            | Payer  | Base               | Amount (USDC) |
|-----------------|--------|--------------------|---------------|
| Trade fee       | buyer  | 2,956.80 filled    | 14.784        |
| Pledge fee      | lender | 2,217.60 advance   | 11.088        |
| Delivery fee    | seller | 2,956.80 delivered | 73.92         |
| Fee vault total | —      | —                  | 99.792        |
| Fault bond      | —      | none filed         | 0             |

Trade and delivery fees together are 3.0% of the notional; the pledge fee adds 0.5% of the advance, here 0.375% of the notional. Had the buyer filed a fault, it would have posted a bond of 147.84 USDC and received it back or lost it to the seller on resolution; the fee vault total would be unchanged. With six missing hours the delivery fee falls to 71.28 USDC, because only the 2,851.20 USDC of delivered escrow bears it.

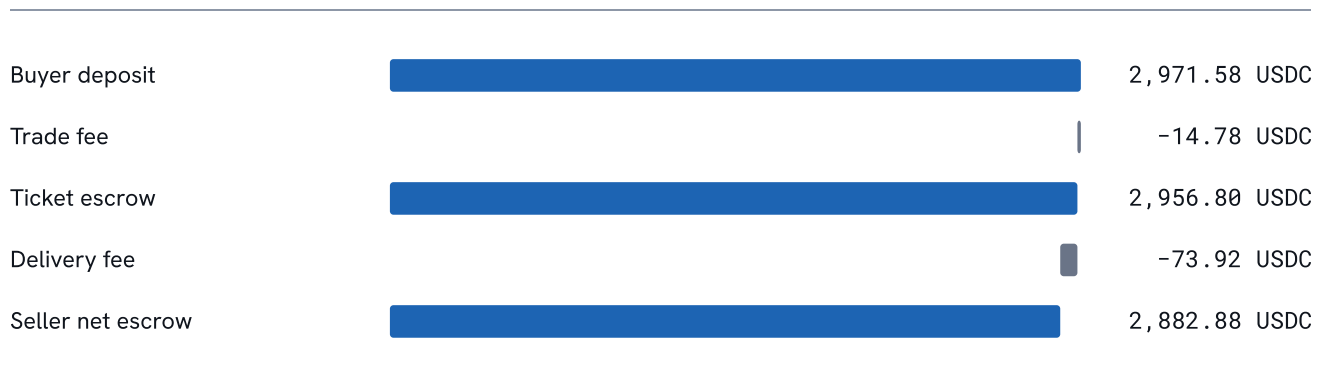


Figure 35 Fees taken from one node-week's cash at an example price of 2.20 USDC per GPU-hour, from the buyer's deposit to the seller's net escrow.

## 13 Credit

A seller with sold node-weeks holds a receivable that buyers have already paid in full: the escrow behind its obligation. Hourglass credit lets one allowlisted lender advance USDC to the seller against that escrow. In exchange the lender becomes the first payee on every escrow release to the seller, until its principal is repaid. The buyer's position is unchanged: refunds and damages are paid before the lender sees any cash.

The lender's repayment depends only on how many hours the seller delivers, because it is paid out of escrow released for certified healthy hours. What a lender's diligence team assesses is therefore the seller's delivery performance on specific, fully prepaid node-weeks; the amounts and their priority are fixed by the programs.

### 13.1 Programs and accounts

Credit is split across two programs. The credit program, `hourglass_credit`, owns the `Pledge` record, the lender's repayment vault and repayment receipts, and exposes five instructions: `pledge_obligation`, `repay_from_escrow`, `repay_pledge`, `collect_lender_cash` and `release_pledge`. The market program, `hourglass`, owns the obligation, the escrow and the payout order. It exposes `configure_credit` to the administrator, and `attach_pledge` and `detach_pledge`, which only the credit program's pledge PDA can sign. Its settlement and waiver instructions call into credit through one bounded bridge.

|                                                                                                                                                    |       |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| <b>Config</b><br>["config"] (market)<br>credit_advance_bps, pledge_fee_bps, fee vault                                                              | —     |
| <b>Obligation</b><br>["obl", series, seller] (market)<br>escrow_notional, certified_unsettled_quantity, pledge<br>→ Pledge pledge                  | 157 B |
| <b>Pledge</b><br>["pledge", obligation] (credit)<br>lender, advance, repaid, cap, fee, state<br>→ Obligation obligation → Config rates snapshotted | 208 B |
| <b>Repayment vault</b><br>["repayment-vault", pledge] (credit)<br>lender USDC; authority Pledge<br>→ Pledge token authority                        | 165 B |
| <b>RepaymentReceipt</b><br>["repayment", pledge, nonce] (credit)<br>payer, amount, paid_at<br>→ Pledge pledge                                      | 93 B  |
| <b>Credit-repayment signer</b><br>["credit-repayment", obligation] (market)<br>no data; signs bookkeeping CPI<br>→ Obligation seed                 | 0 B   |
| <b>Series funding vault</b><br>["funding", series] (market)<br>settlement cash in transit<br>→ Repayment vault bounded repayment                   | 165 B |

## Fee vault

165 B

```
["fees", config] (market)  
pledge fee
```

Figure 36 Accounts involved in a pledge and how they reference each other across the market and credit programs.

The `Pledge` is 208 bytes and binds one obligation to one lender. Because its address is derived from the obligation, an obligation carries one pledge record, and the obligation's single `pledge` field points to it while it is active.

| Field                                                                                     | Meaning                                         |
|-------------------------------------------------------------------------------------------|-------------------------------------------------|
| <code>obligation</code> , <code>series</code> , <code>seller</code> , <code>lender</code> | the bound tuple, checked on every use           |
| <code>advance</code>                                                                      | principal paid to the seller                    |
| <code>repaid</code>                                                                       | principal repaid so far                         |
| <code>pending_lender_usdc</code>                                                          | repaid principal held for the lender            |
| <code>escrow_at_pledge</code> , <code>cap</code>                                          | escrow basis and advance ceiling at origination |
| <code>max_advance_bps</code> , <code>fee_bps</code> , <code>fee</code>                    | rates and fee snapshotted at origination        |
| <code>state</code>                                                                        | 1 active, 2 released                            |

## 13.2 Enabling credit

New advances require an explicit policy. `Config.credit_advance_bps` starts as `None` at `init_config` , and while it is unset `attach_pledge` fails, so no pledge can originate. The administrator sets it with `configure_credit(Some(bps))` , where bps is from 1 to 10,000, or clears it with `None` . The service launches with 7,500 bps: an advance may be at most 75% of the obligation's escrowed notional. The pledge fee rate, `Config.pledge_fee_bps` , is 50 bps from initialization.

## 13.3 Origination

`pledge_obligation(advance, expected_escrow)` is one atomic transaction signed by both the seller and the lender. The service prepares it with both wallets listed as required signers, and each signs the same unchanged message: the seller first, the lender when accepting (below). The lender pays the network fee; the seller funds the rent of the new `Pledge` and repayment vault.

### 13.3.1 Offers signed at different times

A seller and a lender rarely sign within the lifetime of one recent blockhash, so the service carries the pledge as an offer. The seller names the obligation, the lender's wallet and the advance. The service creates a durable nonce account for the offer, with the worker key as its nonce authority, and prepares the pledge transaction with the nonce advance as its first instruction and the lender as fee payer. The seller signs and the service stores the seller-signed bytes after checking them against the prepared message. The lender later signs exactly those bytes; the service adds the nonce authority's signature last, simulates the transaction and relays it. Because the nonce, not a recent blockhash, carries the transaction, the second signature can follow the first by days.

An offer cannot outlive its terms. It lapses after seven days by default, and it lapses at once if the obligation's escrow basis moves away from the amount both parties signed. Either side may cancel an open offer. Lapse and cancellation advance or close the nonce, which makes the stored transaction permanently invalid, so a signature collected for an offer can never land after the offer ends. A wallet that changes the message after the first signature produces bytes that no longer match the stored message, and the service refuses them instead of relaying.

The program checks, in order:

1. Seller and lender are different wallets.
2. The seller's eligibility is approved (KYC, sanctions clear, allowlisted), KYB-verified and holds the seller role; the lender's is approved and holds the lender role.
3. The series is not paused, its cap has not been released, and the obligation has no pledge.
4. A credit policy is configured.
5. The obligation's current escrow equals `expected_escrow`, and no ticket of the obligation is certified but unsettled.
6. The advance is positive and no larger than the cap.
7. Both USDC accounts are initialized and unfrozen.

$$K = \lfloor E \times b / 10000 \rfloor$$

$$\phi = \lceil A \times f / 10000 \rceil$$

K is the advance cap, E the obligation's escrowed notional, b the configured advance rate in basis points, A the advance with  $0 < A \leq K$ ,  $\phi$  the pledge fee and f the pledge fee rate (50 bps). Amounts are USDC base units; the cap rounds down and the fee rounds up.

The expected-escrow argument makes both counterparties sign the exact basis of the cap. If the obligation's escrow changes between quote and execution, through a new fill, a settlement or a waiver, the transaction fails with `EscrowChanged` and must be quoted again. The certified-unsettled check keeps a ticket that is about to refund its buyer from counting as collateral for a new advance.

When the checks pass, the lender's account pays A to the seller's canonical USDC account and  $\phi$  to the fee vault. The program records the pledge, including the escrow basis, cap, fee and both rates, then calls the market's `attach_pledge` signed by the pledge PDA. The market confirms that the signer is the canonical `["pledge", obligation]` address owned by the credit program, that credit is enabled, and that the obligation has escrow and no pledge, and then sets `Obligation.pledge`.

The debt is the principal A alone. The lender pays the fee on top of the advance, and nothing is added to what the seller owes. Repeating an executed origination with identical arguments returns success without moving funds; a different payload against an existing pledge fails with `ReplayConflict`.

Before preparing the transaction, the service runs the same checks against its index and quotes the terms with the program's exact integer formulas. It also confirms that the lender's USDC balance covers  $A + \phi$ , and it refuses a quote whose expected escrow no longer matches the indexed obligation, so a stale offer fails before either party signs.

### 13.4 Repayment from escrow releases

Every release of escrow to the seller passes through the lender first. There are three such releases, and each hands the bridge the amount the seller would otherwise receive:

| Release          | Instruction                 | Proceeds offered to the lender |
|------------------|-----------------------------|--------------------------------|
| Certified ticket | <code>settle_ticket</code>  | seller net escrow $G - F$      |
| Buyer default    | <code>settle_ticket</code>  | full ticket escrow N           |
| Unrequested unit | <code>waive_delivery</code> | full original notional         |

$$L = \min(\Pi, A - R_p)$$

L is the repayment,  $\Pi$  the proceeds offered, A the advance and  $R_p$  the principal already repaid. The seller receives  $\Pi - L$ . Because the market computes  $\Pi$  only after the buyer's refund and damages and the delivery fee are settled (§11 Settlement), the ranking is fixed: buyer refunds first, then delivery fees, then lender principal, then the seller.

Collateral is never pledged. The seller's collateral pays only the buyer's damages, and its released remainder returns to the seller's free collateral outside the waterfall. The lender has no claim on it.

#### 13.4.1 The CPI boundary

The series funding vault holds settlement cash for every ticket in the series, so the credit program never touches it. The market performs the only debit from that vault itself. When the obligation is pledged, settlement and waiver require exactly five additional accounts: the

credit program, the pledge, the repayment vault, the lender's USDC account and the market's credit-repayment signer. The market then checks:

- the credit program is the configured, executable credit program ID, and the pledge is owned by it, writable, equal to `Obligation.pledge` and at the canonical `["pledge", obligation]` address;
- the repayment vault is the canonical `["repayment-vault", pledge]` token account, owned by the pledge and holding the USDC mint;
- the lender account is the canonical USDC account of the lender recorded in the pledge;
- the pledge decodes as a 208-byte credit `Pledge` of the current layout, is active, matches the obligation, series and seller, and has `repaid ≤ advance` and `pending ≤ repaid`.

It computes `L`, transfers `L` from the funding vault to the repayment vault under the `Series` signature, and only then invokes the credit program's `repay_from_escrow(L)`. That call is signed by the market PDA `["credit-repayment", obligation]`, which holds no data and no token authority. The credit program receives that signer, the series, `Config`, the obligation, the pledge, the repayment vault, the lender's account and the token program. It never receives the funding vault, the escrow or collateral vaults, or the `Series` as a signer, and the market ignores its return data.

On the credit side, `repay_from_escrow` accepts only that exact market signer for that obligation, requires an active pledge, rejects an amount above outstanding principal and requires the repayment vault to hold the pending lender cash plus the new amount. It then pays the lender and adds `L` to `repaid`. The instruction has no other caller: the service rejects it as a user action, and only certified settlement or a waiver can produce it.

### 13.5 Lender cash

Lender payments follow the same deferral rule as the market's payouts. The credit program pays only the lender's canonical USDC account, and only if it is initialized, unfrozen, owned by the lender and holds USDC. Otherwise the amount stays in the repayment vault and is added to `pending_lender_usdc`. `collect_lender_cash` transfers the whole pending amount: anyone may send it to the canonical account, and the lender alone may name another initialized USDC account it owns. The instruction does not depend on the pledge's state, so cash remains collectable after release.

### 13.6 Monitoring a pledge

The lender's console is built from the indexed accounts. For each of its pledges it shows the advance, principal repaid, snapshotted cap rate and fee, cash pending collection and a status of funded, repaid or released. It also shows the waterfall summed over the obligation's settled tickets: buyer refunds paid first, lender repayment and seller remainder, read from the amounts each ticket records at settlement. Obligations that hold escrow and carry no pledge are listed with their escrow, locked collateral and node-week count as pledgeable.

## 13.7 Direct repayment and release

Principal that escrow releases have not covered stays outstanding on the pledge. The seller repays it with `repay_pledge(amount, nonce)`, which moves `amount` from the seller's USDC account through the repayment vault to the lender and records an immutable `RepaymentReceipt` at `["repayment", pledge, nonce]`. The amount must be positive and at most the outstanding principal. Resubmitting the same receipt is harmless, and a different payload under a used nonce fails.

`release_pledge` is signed by the seller and succeeds only when `repaid` equals `advance`. It calls the market's `detach_pledge` with the pledge PDA's signature, which clears `Obligation.pledge`, and marks the pledge released with a timestamp. Later escrow releases from the obligation then pay the seller directly. A second release returns success only if the market pointer is already clear.

## 13.8 Worked example

A seller has an obligation of one node-week sold at an example clearing price of 2.20 USDC per GPU-hour, with 2,956.80 USDC in escrow. At the launch advance rate of 7,500 bps the cap is 2,217.60 USDC. The lender advances the full cap and pays a pledge fee of  $0.5\% \times 2,217.60 = 11.088$  USDC, a total debit of 2,228.688 USDC. The seller receives 2,217.60 USDC weeks before delivery.

The week is delivered in full and certified at 168 hours. Settlement refunds nothing, takes the 73.92 USDC delivery fee and leaves a seller net of 2,882.88 USDC. The lender is repaid  $\min(2,882.88, 2,217.60) = 2,217.60$  USDC, and the seller receives the remaining 665.28 USDC. The pledge is fully repaid, and the seller can call `release_pledge`. Across origination and settlement the seller has received  $2,217.60 + 665.28 = 2,882.88$  USDC, its unpledged net, with 2,217.60 USDC of it in advance.

---

|                   |             |                |
|-------------------|-------------|----------------|
| Ticket escrow     | <div></div> | 2,956.80 USDC  |
| Delivery fee      | <div></div> | -73.92 USDC    |
| Seller net escrow | <div></div> | 2,882.88 USDC  |
| Lender principal  | <div></div> | -2,217.60 USDC |
| Seller remainder  | <div></div> | 665.28 USDC    |

---

Figure 37 Escrow release of a pledged, fully delivered node-week at an example price of 2.20 USDC per GPU-hour with a 75% advance.

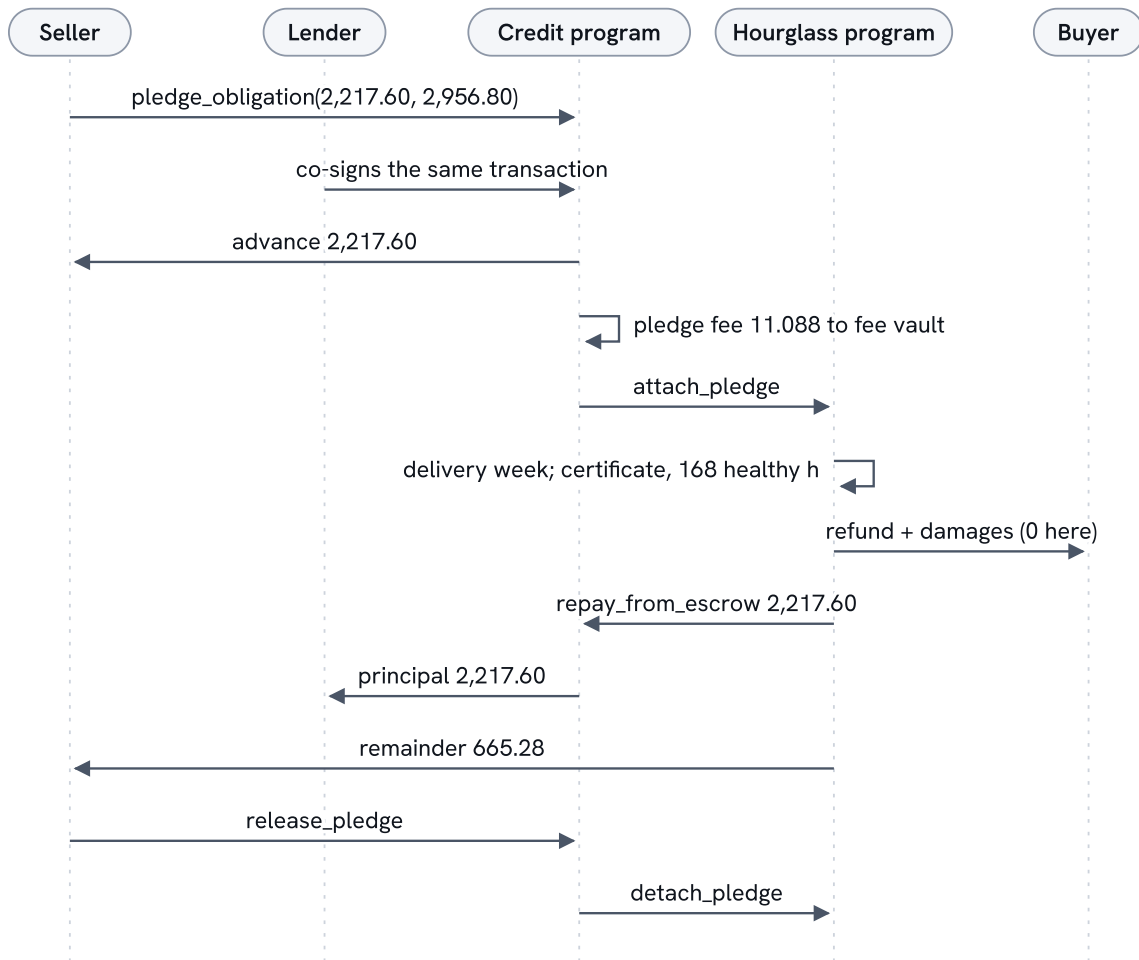


Figure 38 Pledge, delivery and repayment for the worked example.

With six missing hours instead, the buyer is first paid its 105.60 USDC refund and 19.008 USDC of damages, the seller's net falls to 2,779.92 USDC, the lender is still repaid 2,217.60 USDC and the seller receives 562.32 USDC. Damages come from collateral and do not reduce the lender's proceeds.

For an obligation of two node-weeks at the same example price, escrow is 5,913.60 USDC, the cap 4,435.20 USDC and the fee 22.176 USDC. Suppose one unit is requested and delivered clean and the other is never requested. Waivers run strictly after notice, before delivery starts, so the waiver comes first: it offers the unit's full 2,956.80 USDC notional, all of which goes to the lender and leaves 1,478.40 USDC outstanding. At settlement the delivered ticket's net of 2,882.88 USDC repays that 1,478.40 USDC, and the seller receives 1,404.48 USDC.

## 14 Program architecture

Hourglass runs on three Anchor programs, built with Anchor 0.31.1. `hourglass` is the market: it owns the configuration, the contract specification, every series and its auction, the claim mint, all USDC custody, delivery, evidence, faults and settlement. `hourglass_credit` holds only finance: lender pledges against seller obligations and the bookkeeping of their repayment. `hourglass_hook` is the Token-2022 transfer hook that decides whether a claim token may move. This chapter describes the accounts each program owns, the instructions each caller may send, where one program invokes another, and the compute and packet budgets that the heavy instructions are built to.

### 14.1 Three programs and one layout

The split follows two rules. Everything that holds or moves market value stays in one program, so the auction, custody, delivery and settlement never need a cross-program call between themselves. Logic that must be invoked from outside the market lives elsewhere. The transfer hook is the clearest case: Token-2022 calls the hook during every claim transfer, and many of those transfers are started by the market itself (an order escrowing claims, inventory moving to a buyer). Solana forbids a program from being re-entered indirectly through another program, so a hook inside the market would make those transfers impossible. The hook therefore lives in its own program. Credit is separate because a lender relationship is an optional layer over an obligation, with its own signer set and its own accounts.

The three programs share one account layout, version 11. `Config` and `Series` both begin with `layout_version: u16` immediately after the 8-byte Anchor discriminator. Every market instruction that reads `Config` requires `layout_version == STATE_LAYOUT_VERSION`, the evidence and credit contexts require it of `Series` as well, and both the hook and the credit program read those two bytes before they decode a market account. An account written under any other layout is rejected before its fields are interpreted. Changing a shared layout requires a reviewed migration and a new version number.

The programs are compiled separately. The credit and hook crates depend on the `hourglass` crate for its account types, and a single workspace build would unify Cargo features across them, which could strip the market's own entrypoint. Building each program on its own keeps every executable complete.

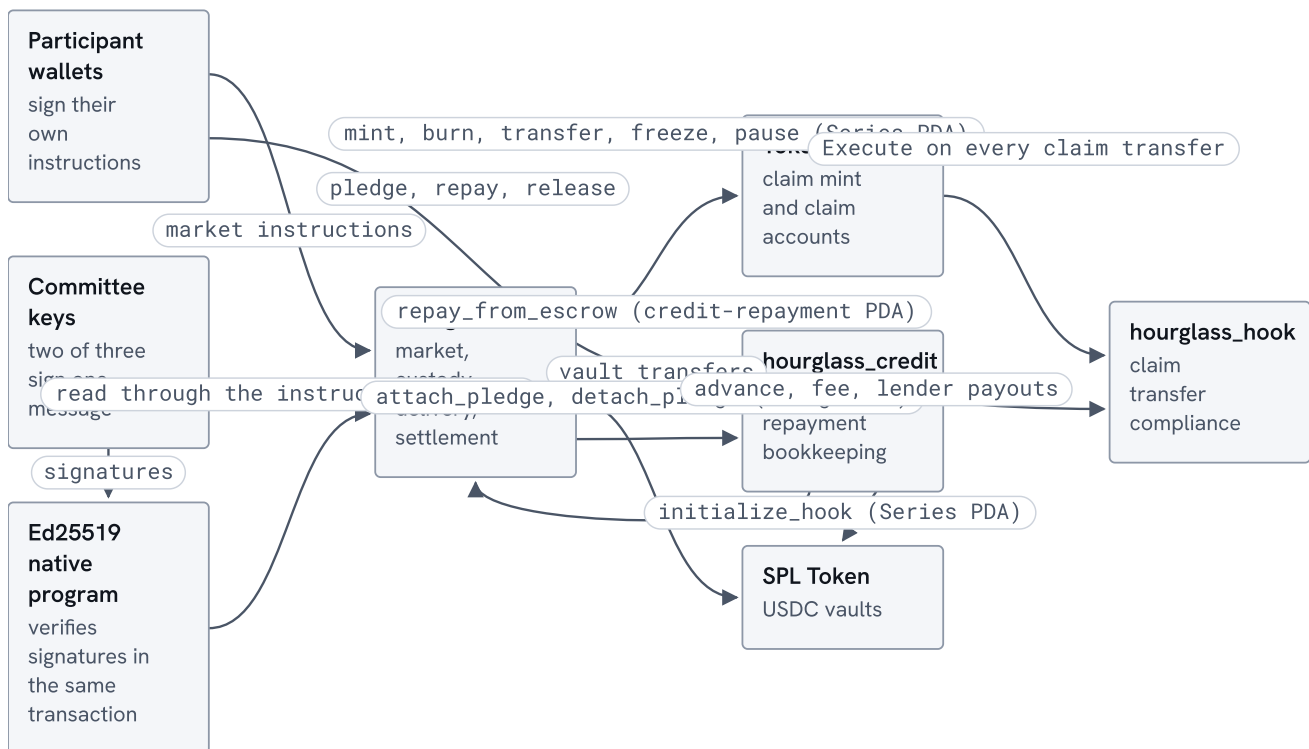


Figure 39 The three Hourglass programs, the system programs they call, and which program-derived address signs each cross-program call.

## 14.2 Encoding conventions

All three programs use the same encodings. Quoted seeds are ASCII byte strings; public-key seeds are the 32 raw key bytes. Numeric seeds are little-endian: a delivery week is the Monday 00:00 UTC Unix time as `u64`, round, ticket and dispute indices are `u16`, and order nonces and arrival sequences are `u32`. USDC amounts and prices are `u64` in base units (six decimals), and every product of two amounts or of an amount and a rate is computed in checked `u128` before it is narrowed back. Quantities are whole node-weeks.

Small enumerations are fixed integers. Region is `0` for NA and `1` for EU. Order side is `0` for a bid, `1` for a fresh ask and `2` for a claim sell. Participant roles are bit flags in one `u8` on the `Eligibility` account, and `set_eligibility` accepts any combination from 1 through 63.

| Role   | Bit | Role      | Bit |
|--------|-----|-----------|-----|
| buyer  | 1   | inspector | 8   |
| seller | 2   | lender    | 16  |
| trader | 4   | admin     | 32  |

## 14.3 Market program accounts

Every market account is a program-derived address (PDA) of `hourglass`. Sizes below are the compiled account sizes including the 8-byte discriminator; raw accounts carry an 8-byte tag instead. USDC vaults are ordinary SPL Token accounts of 165 bytes whose authority is the PDA named in the last column.

### 14.3.1 Configuration, identity and sellers

| Account           | Seeds                         | Size  | Holds                                                                                                                                                                                                                                                                |
|-------------------|-------------------------------|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Config            | config                        | 613 B | admin, KYC authority and pending successors, USDC mint, three committee keys, caps and open interest, fees, auction policy, genesis hash, signing domain, credit policy, the \$HOURGLASS mint, its token program and buyback keeper, the buyback sequence and totals |
| Fee vault         | fees, config                  | 165 B | trade, delivery and pledge fees; authority Config                                                                                                                                                                                                                    |
| Buyback authority | buyback-authority, config     | —     | no data; owns the buyback's USDC and \$HOURGLASS custody accounts and signs the Jupiter swap and the burn                                                                                                                                                            |
| BuybackReceipt    | buyback, config, sequence     | 162 B | one buyback: keeper, USDC claimed, minimum and burned output, custody balances before the swap; opened by <code>claim_buyback</code> and closed to the keeper by <code>finish_buyback</code> in the same transaction                                                 |
| ContractSpec      | spec, "HG-H100x8"             | 60 B  | spec hash, hardware floors, tolerance, damages floor, collateral schedule, cure and challenge hours                                                                                                                                                                  |
| Eligibility       | eligibility, wallet           | 46 B  | roles, KYC, KYB, sanctions and allowlist flags for one wallet                                                                                                                                                                                                        |
| SellerProfile     | seller, wallet                | 169 B | KYB hash, X25519 key for access envelopes, free, reserved and locked collateral, node counts                                                                                                                                                                         |
| Collateral vault  | collateral, seller_profile    | 165 B | the seller's USDC collateral; authority SellerProfile                                                                                                                                                                                                                |
| NodeRecord        | node, seller_profile, node_id | 443 B | eight GPU UUID hashes and their set hash, report hash, inspector, region, Tier A flag, expiry, <code>booked_until</code>                                                                                                                                             |
| UuidMarker        | gpu-uuid, uuid_hash           | 49 B  | the node that currently owns one physical GPU identity, and until when                                                                                                                                                                                               |

Config is the only singleton. Its `signature_domain` binds every committee message to one deployment (see §15 Security design). Accreditation claims one UuidMarker per GPU, so a GPU that is already part of an unexpired node cannot be accredited again under a different node, even if seven of the eight identities differ. Expired markers can be reassigned; renewing a node requires that all eight markers still belong to it.

### 14.3.2 Series and trading

| Account         | Seeds                                | Size       | Holds                                                                                                                                                                                                                          |
|-----------------|--------------------------------------|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Series          | series,<br>spec,<br>region,<br>week  | 466 B      | delivery, notice and next-auction times, round, open interest and quantity, snapshotted economics, escrow accounting, allocation counters, pause flag, the payers of its operator-funded accounts, retirement flag and cursors |
| Auction book    | book,<br>series                      | 65,544 B   | raw slab: tag HGB00K02 plus 1,024 order slots of 64 bytes                                                                                                                                                                      |
| Funding vault   | funding,<br>series                   | 165 B      | bid deposits, refunds and deferred cash; authority Series                                                                                                                                                                      |
| Escrow vault    | escrow,<br>series                    | 165 B      | buyer notional backing fresh issuance; authority Series                                                                                                                                                                        |
| Claim mint      | mint,<br>series                      | Token-2022 | zero-decimal node-week claims with seven extensions                                                                                                                                                                            |
| Claim inventory | claims,<br>series                    | Token-2022 | program-held claims; owner Series                                                                                                                                                                                              |
| Order           | order,<br>series,<br>owner,<br>nonce | 161 B      | durable receipt: price, quantity, funding, collateral budget, snapshotted trade fee, fill, pending cash, claims and mints, claims waived at retirement                                                                         |
| AuctionResult   | auction,<br>series,<br>round         | 124 B      | clearing price, volume, new issuance, scheduled time, remaining weeks, grace, fresh mint progress, no-cross reason, rent payer                                                                                                 |
| Obligation      | obl,<br>series,<br>seller_wallet     | 157 B      | one seller's short: capacity, reserved, short, assigned and waived quantities, escrow notional, locked collateral, pending cash, pledge pointer                                                                                |
| CapacitySlot    | capacity,<br>node, week              | 73 B       | one accredited node reserved for one series week                                                                                                                                                                               |

A **Series** copies the collateral schedule, tolerance, damages floor, cure and challenge hours from **ContractSpec**, and the trade and delivery fees from **Config**, at creation. Later configuration changes therefore never alter the economics of a listed series. Each **Order** snapshots its own trade fee, and each round snapshots its failed-auction grace when its first order arrives.

The book is a zero-copy slab that the program reinterprets in place. Each 64-byte slot holds the owner key (32 bytes), price and collateral budget (8 bytes each), quantity, fill and arrival sequence (4 bytes each), new-issuance count (2 bytes), side and state (1 byte each). Only the first 64 slots admit live orders, 32 per economic direction; placement and clearing both reject the account if any of the remaining 960 slots is active. A slot is reused once its order is cancelled or materialized, while the **Order** PDA keeps the permanent financial record, so slot reuse never loses a receipt or resets arrival priority.

The claim mint is created by `init_claim_mint` with zero decimals and the extensions `DefaultAccountState` (Frozen), `MetadataPointer`, `TokenMetadata`, `Pausable`, `MintCloseAuthority`, `PermanentDelegate` and `TransferHook`, the last pointing at `hourglass_hook`. The `Series` PDA holds every authority over it: mint, freeze, pause, metadata, close and permanent delegate. The program uses the permanent delegate in one instruction, `sweep_retired_claims`, which burns the claims left after a series' units were waived so that the empty mint can close at retirement (§4 Series and the calendar). Its metadata names the series and region, uses the symbol `HG-H100x8`, and records the series address, spec hash, delivery start and end, and region as fields. The inventory account carries `ImmutableOwner`, `TransferHookAccount` and `PausableAccount` and is thawed at creation; it is the one claim account the program itself holds. Claims and their transfer rules are described in §7 Claims and transfer.

#### 14.3.3 Delivery, evidence and faults

| Account                          | Seeds                                     | Size     | Holds                                                                                                                                                                         |
|----------------------------------|-------------------------------------------|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Delivery registry                | <code>delivery-alloc, series</code>       | 32,776 B | raw: tag <code>HGDEL006</code> plus 512 issuance units of 64 bytes                                                                                                            |
| <code>DeliveryTicket</code>      | <code>ticket, series, index</code>        | 880 B    | buyer, seller, obligation, node, price, escrow, allocated collateral, access and acceptance state, hold seconds per hour, arbitration hours, settlement amounts, cash credits |
| <code>NodeBooking</code>         | <code>booking, node, week</code>          | 121 B    | the one ticket a node serves in a delivery week                                                                                                                               |
| <code>EpochLedger</code>         | <code>epoch, series</code>                | 7,520 B  | zero-copy: 168 hourly roots, 512 healthy-hour counters, committed-hour bitmap, rolling evidence chain hash                                                                    |
| Health archive                   | <code>epoch-health, series</code>         | 10,800 B | raw: tag <code>HGHEAL06</code> , 48-byte header and 168 healthy-ticket bitmaps of 64 bytes                                                                                    |
| <code>Dispute</code>             | <code>dispute, ticket, index</code>       | 444 B    | cause, claimant evidence, contest and probe hashes, bond, cure and arbitration deadlines, alleged hour interval, verdict, covered hours                                       |
| Fault bond vault                 | <code>fault-bond, dispute</code>          | 165 B    | the claimant's 5% bond; authority <code>Dispute</code>                                                                                                                        |
| <code>DeliveryCertificate</code> | <code>certificate, ticket</code>          | 311 B    | healthy micro-hours, reference spot and source hash, evidence root and mode, disposition, excused micro-hours, health adjustment, report digest, the two signers              |
| Repayment signer                 | <code>credit-repayment, obligation</code> | no data  | signs credit bookkeeping for one obligation                                                                                                                                   |
| <code>AgentBinding</code>        | <code>agent-binding, ticket</code>        | 169 B    | the ticket, its buyer and seller, and the Ed25519 keys of its Delivery Agent and Provider Agent                                                                               |
| <code>InspectorBond</code>       | <code>inspector-bond, inspector</code>    | 73 B     | the inspector's bonded amount, withdrawal request time, liability horizon and release time                                                                                    |

| Account              | Seeds                              | Size  | Holds                                    |
|----------------------|------------------------------------|-------|------------------------------------------|
| Inspector bond vault | inspector-bond-vault,<br>inspector | 165 B | the bonded USDC; authority InspectorBond |

Each registry unit records the selling wallet, the original price and the unit's share of locked collateral (8 bytes each after the 32-byte wallet), the source arrival sequence, the ticket index once requested, the ordinal within its lot, and a state byte: unrequested, requested or waived. Collateral divides by the filled quantity with the remainder assigned to the first ordinals, so the units of one lot sum exactly to the collateral locked for it. `seal_delivery` sorts the units by seller key bytes, sequence and ordinal, and `request_delivery` selects the next unit by greatest proportional seller deficit (§8 Delivery).

`AgentBinding` names the two keys whose minute records count for a ticket. `bind_delivery_agent` is signed by the ticket's buyer and `bind_provider_agent` by its seller; each writes one key, which must be nonzero. A party may replace its key until delivery starts; from the delivery start the key can only be re-sent unchanged, so the identity behind a week's evidence never changes mid-week. Agent keys sign evidence and nothing else: no instruction accepts them as a signer for funds. `InspectorBond` holds an inspector's USDC in a vault owned by the bond account. `deposit_inspector_bond` adds to it from an approved inspector's wallet, and accreditation, renewal, acceptance, cure acceptance and probe resolution each require a bond of at least `Config.inspector_bond_minimum` held by the signing inspector with no withdrawal requested. Each of those instructions also extends the bond's liability horizon. `request_inspector_bond_withdrawal` deactivates the bond at once, and `withdraw_inspector_bond` later returns the full balance and both accounts' rent to the inspector (§3 Participants and eligibility).

The `DeliveryTicket` is the largest fixed account because it stores per-hour state: 168 `u16` counters of buyer-only compliance-hold seconds (336 bytes) and a 21-byte bitmap of whole hours that arbitration has excused. The `EpochLedger` holds the aggregate per-ticket counters that certificates must match; the health archive holds the hour-by-hour bitmaps that prove which excused hours were actually unhealthy. Keeping the archive in a separate account keeps the ledger at 7,520 bytes. Both are advanced together by `commit_epoch`, one closed hour at a time, and neither can be rewritten (§9 Evidence and the committee).

|                                                                                                                                                             |          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| <b>Config</b><br><code>["config"]</code><br>authorities, committee, caps, fees, signing domain                                                              | 613 B    |
| <b>ContractSpec</b><br><code>["spec", "HG-H100x8"]</code><br>grade floors, tolerance, collateral, cure                                                      | 60 B     |
| <b>Eligibility</b><br><code>["eligibility", wallet]</code><br>roles, KYC, KYB, sanctions, allowlist                                                         | 46 B     |
| <b>SellerProfile</b><br><code>["seller", wallet]</code><br>collateral balances, X25519 key<br>→ Eligibility wallet                                          | 169 B    |
| <b>NodeRecord</b><br><code>["node", seller, node_id]</code><br>eight GPU UUID hashes, expiry<br>→ SellerProfile seller                                      | 443 B    |
| <b>Series</b><br><code>["series", spec, region, week]</code><br>schedule, open interest, snapshotted economics<br>→ ContractSpec spec → Config reserved cap | 466 B    |
| <b>Auction book</b><br><code>["book", series]</code><br>1,024 slots, first 64 admitted<br>→ Series seed                                                     | 65,544 B |
| <b>Order</b><br><code>["order", series, owner, nonce]</code><br>durable order receipt<br>→ Auction book slot and sequence → AuctionResult round result      | 161 B    |

|                                                                                                                                   |       |
|-----------------------------------------------------------------------------------------------------------------------------------|-------|
| <b>AuctionResult</b><br>["auction", series, round]<br>clearing price and volume<br>→ Series round                                 | 124 B |
| <b>Obligation</b><br>["obl", series, seller]<br>seller short, escrow, collateral<br>→ Series series → SellerProfile seller wallet | 157 B |

Figure 40 Market program accounts that carry configuration, identity and trading state, and the references between them.

|                                                                                                                                               |          |
|-----------------------------------------------------------------------------------------------------------------------------------------------|----------|
| <b>Series</b><br>["series", spec, region, week]<br>allocation and settlement counters                                                         | 466 B    |
| <b>Delivery registry</b><br>["delivery-alloc", series]<br>512 original issuance units<br>→ Series seed                                        | 32,776 B |
| <b>Obligation</b><br>["obl", series, seller]<br>escrow notional, pledge pointer                                                               | 157 B    |
| <b>DeliveryTicket</b><br>["ticket", series, index]<br>one requested node-week<br>→ Delivery registry allocated unit → Obligation seller short | 880 B    |
| <b>NodeBooking</b><br>["booking", node, week]<br>one ticket per node-week<br>→ DeliveryTicket ticket                                          | 121 B    |
| <b>Dispute</b><br>["dispute", ticket, index]<br>bonded fault or objection<br>→ DeliveryTicket ticket                                          | 444 B    |
| <b>DeliveryCertificate</b><br>["certificate", ticket]<br>immutable signed report<br>→ DeliveryTicket ticket → EpochLedger evidence chain hash | 311 B    |

|                                                                                                                       |          |
|-----------------------------------------------------------------------------------------------------------------------|----------|
| <b>EpochLedger</b><br>["epoch", series]<br>168 roots, 512 counters<br>→ Series seed                                   | 7,520 B  |
| <b>Health archive</b><br>["epoch-health", series]<br>168 hourly bitmaps<br>→ Series seed                              | 10,800 B |
| <b>Pledge</b><br>["pledge", obligation] (credit)<br>advance, repaid, lender credit<br>→ Obligation pledged obligation | 208 B    |

Figure 41 Market and credit accounts that carry delivery, evidence, faults, settlement and finance for one series.

## 14.4 Credit and hook accounts

The credit program owns three accounts. All of them are PDAs of `hourglass_credit`, and its instructions read the market's `Config`, `Series`, `Obligation` and `Eligibility` accounts under the market program's own seeds.

| Account          | Seeds                    | Size  | Holds                                                                                                                   |
|------------------|--------------------------|-------|-------------------------------------------------------------------------------------------------------------------------|
| Pledge           | pledge, obligation       | 208 B | seller, lender, advance, repaid, pending lender cash, escrow at origination, cap and fee with their basis points, state |
| Repayment vault  | repayment-vault, pledge  | 165 B | lender proceeds not yet paid out; authority Pledge                                                                      |
| RepaymentReceipt | repayment, pledge, nonce | 93 B  | one seller-funded repayment: payer, amount, nonce, time                                                                 |

The hook owns one account per claim mint: the SPL extra-account-metas list at `extra-account-metas, mint`, 191 bytes for its five entries. Token-2022 reads it to append the accounts the hook needs to every transfer: the `Series` and `Config` addresses, the market program address, and two eligibility PDAs derived under the market program from the literal `eligibility` and the owner field (bytes 32 to 63) of the source and destination token accounts. The hook has no other state; it decides each transfer from market accounts alone. The list is closed when its series retires.

## 14.5 Instructions by caller

Every instruction names the key that must sign it, and the program checks that key against stored state rather than against anything the caller supplies. Instructions that move money to a fixed beneficiary are permissionless: anyone may pay the fee to run them, and the receipt, ticket or obligation already fixes who is paid.

| Caller            | Instructions                                                                                                                                                                                                                                                                                               | Check                                                                                                                                                                                                                             |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Admin             | <code>init_config</code> , <code>set_committee</code> , <code>configure_auction</code> , <code>configure_credit</code> , <code>set_token_mint</code> , <code>create_spec</code> , <code>create_series</code> , <code>pause_series</code> , <code>propose_admin</code> , <code>propose_kyc_authority</code> | <code>has_one = admin</code> on <code>Config</code> ; <code>init_config</code> also requires the program's upgrade authority                                                                                                      |
| Pending authority | <code>accept_admin</code> , <code>accept_kyc_authority</code>                                                                                                                                                                                                                                              | signer equals the proposed key                                                                                                                                                                                                    |
| Series rent payer | <code>grow_book</code> , <code>grow_delivery_registry</code>                                                                                                                                                                                                                                               | signer equals <code>Series.rent_payer</code> , the admin that created the series                                                                                                                                                  |
| KYC authority     | <code>set_eligibility</code> , <code>thaw_claim_account</code> , <code>freeze_claim_account</code> , <code>mark_compliance_hold</code> , <code>release_compliance_hold</code>                                                                                                                              | <code>has_one = kyc_authority</code> or equality with <code>Config.kyc_authority</code>                                                                                                                                           |
| Seller            | <code>register_seller</code> , <code>deposit_collateral</code> , <code>withdraw_collateral</code> , <code>reserve_capacity</code> , <code>place_order</code> (fresh ask), <code>assign_node</code> , <code>bind_provider_agent</code> , <code>contest_fault</code> , <code>cure_fault</code>               | signer is the profile, obligation or ticket seller; role, KYB and eligibility for registration, asks and assignment                                                                                                               |
| Buyer or trader   | <code>open_claim_account</code> , <code>place_order</code> (bid, claim sell), <code>cancel_order</code> , <code>request_delivery</code> , <code>set_access_hash</code> , <code>bind_delivery_agent</code> , <code>confirm_access</code> , <code>object_access</code> , <code>file_fault</code>             | signer is the order owner or <code>ticket.buyer</code> ; role and eligibility for entry, requests and access; for <code>open_claim_account</code> , the paying signer and the account's owner are both approved buyers or traders |
| Inspector         | <code>deposit_inspector_bond</code> , <code>accredit_node</code> , <code>renew_node</code> , <code>record_acceptance</code> , <code>accept_cure</code> , <code>resolve_fault</code> , <code>request_inspector_bond_withdrawal</code> , <code>withdraw_inspector_bond</code>                                | inspector role, current committee seat, funded bond with no withdrawal requested, independent of buyer and seller; the bond's own authority for the two exit instructions                                                         |
| Committee         | <code>commit_epoch</code> , <code>post_certificate</code> , <code>resolve_arbitration</code>                                                                                                                                                                                                               | two distinct current committee signatures in a preceding Ed25519 instruction                                                                                                                                                      |
| Buyback keeper    | <code>claim_buyback</code> , <code>buyback_swap</code> , <code>finish_buyback</code>                                                                                                                                                                                                                       | signer equals <code>Config.buyback_keeper</code> ; the three run together in one transaction (\$12 Fees)                                                                                                                          |

| Caller                           | Instructions                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Check                                                                                 |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Anyone                           | <code>init_claim_mint</code> , <code>init_epoch_ledger</code> , <code>init_epoch_health</code> ,<br><code>grow_epoch_health</code> , <code>clear_auction</code> , <code>claim_fill</code> ,<br><code>materialize_order</code> , <code>collect_claims</code> , <code>collect_cash</code> ,<br><code>enforce_claim_freeze</code> , <code>seal_delivery</code> , <code>waive_delivery</code> ,<br><code>finalize_access</code> , <code>mark_buyer_default</code> , <code>advance_arbitration</code> ,<br><code>settle_ticket</code> , <code>collect_ticket_cash</code> ,<br><code>collect_obligation_cash</code> , <code>release_series_cap</code> ,<br><code>begin_retire_series</code> , <code>audit_retirement_order</code> ,<br><code>audit_retirement_ticket</code> , <code>audit_retirement_unit</code> ,<br><code>sweep_retired_claims</code> , <code>sweep_retired_surplus</code> ,<br><code>close_retired_auction</code> , <code>close_retired_part</code> ,<br><code>close_retired_series</code> | deadlines, state, fixed<br>beneficiaries and recorded<br>rent payers checked on chain |
| Lender and<br>seller<br>(credit) | <code>pledge_obligation</code> , <code>repay_pledge</code> , <code>collect_lender_cash</code> ,<br><code>release_pledge</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | both signatures for a pledge;<br>seller for repayment and<br>release                  |

Some instructions change their authorization with their arguments. `cancel_order` requires the order owner; its permissionless counterpart `claim_fill` (with the alias `materialize_order`) applies a cleared fill to the fixed owner. `collect_cash` , `collect_ticket_cash` , `collect_obligation_cash` and `collect_lender_cash` pay the beneficiary's canonical USDC account for anyone, and pay an alternate account only when the beneficiary signs. `mark_compliance_hold` and `release_compliance_hold` share the account set of `finalize_access` but compare the signer to the KYC authority. Two further instructions are callable only by another program: `attach_pledge` and `detach_pledge` in the market accept only the credit program's Pledge PDA as signer, and `repay_from_escrow` in the credit program accepts only the market's repayment PDA.

The retirement instructions declare the same three accounts: any signer, `Config` and the mutable `Series` . The record each one audits or closes, and the key it refunds, arrive as remaining accounts. The program derives every address from the series, takes records in the order of the series' cursors, and compares each refund recipient with the payer recorded when the account was funded, so the caller chooses neither the next record nor the recipient. `close_retired_part(part)` names its target by number: 0 the book, 1 the delivery registry, 2 the epoch ledger, 3 the health archive, 4 the funding vault, 5 the escrow vault, 6 the claim inventory, 7 the hook's account list and 8 the claim mint. The sequence is described in §4 Series and the calendar.

The production build compiles without the test-clock feature. In that build `effective_now` always reads the Solana clock sysvar, `init_config` rejects a request for local time, and `set_local_time` fails.

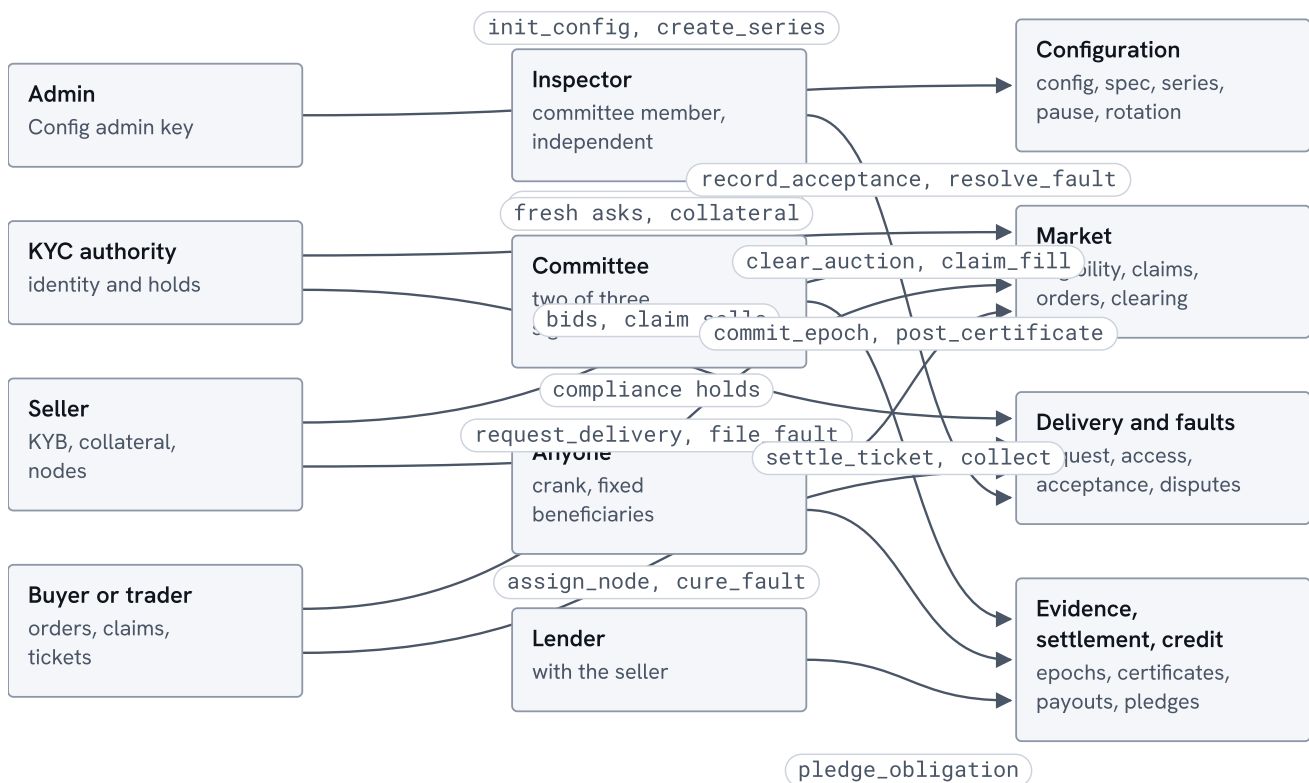


Figure 42 Which caller sends which group of market and credit instructions.

## 14.6 Cross-program boundaries

### 14.6.1 Market to Token-2022

The market invokes Token-2022 for every claim operation: `mint_to_checked` into inventory, `transfer_checked` between inventory and canonical owner accounts, `burn_checked` when a claim converts into a ticket, a waived unit is extinguished or retirement clears the remaining supply, `freeze_account` and `thaw_account`, Pausable `pause` and `resume`, and `close_account` for the empty inventory and mint at retirement. Every one of these calls is signed with the `Series` PDA seeds `series`, `spec`, `region`, `week`, `bump`; burns from a participant's account during a delivery request are signed by that participant. `open_claim_account` also calls the associated token account program's idempotent `create`, paid by its signer, before it thaws the new account. Transfers go through Token-2022's on-chain helper, which resolves the hook's extra accounts from the transaction's remaining accounts.

Two internal paths cross a pause or a freeze without opening a window for anyone else. Minting deferred during a pause is caught up by `collect_claims`, which resumes the mint, mints to inventory and pauses it again inside one instruction. A delivery request burns an owner's claim even from a frozen account or a paused mint: it thaws or resumes, burns, then restores the previous state, and no transfer, and therefore no hook, is involved.

#### 14.6.2 Token-2022 to the hook

Token-2022 invokes `hourglass_hook` with the `Execute` instruction of the `transfer-hook` interface on every claim transfer, whoever starts it. The hook accepts only `Execute` and checks that the source account's `TransferHookAccount` extension has its `transferring` flag set, which Token-2022 sets only for the duration of a real transfer. It then confirms that the extra-metas account is the canonical one for the mint, that `Config` is the market's PDA, that both market accounts carry layout version 11, that the series' mint is the mint being moved, that the series is not paused and that the clock is before notice. Each endpoint must be either the series inventory, owned by the `Series` PDA, or the canonical associated token account of a wallet whose `Eligibility` is approved and holds the buyer or trader role.

The market also calls the hook twice per series, at its start and at its retirement.

`init_claim_mint` invokes `initialize_hook` with the `Series` PDA as a signer; the hook verifies that the signer is market-owned, carries the current layout and names this mint, then writes the extra-metas list. `close_retired_part` invokes `close_hook` with the same signer; the hook requires the series to have passed every retirement audit and the recipient to be the series' recorded mint payer, then moves the list's lamports to that payer. No other caller can produce that signature, so no one else can create, replace or close a mint's hook configuration.

#### 14.6.3 Credit to market

`pledge_obligation` in the credit program runs the whole origination atomically: it checks both signers' eligibility, the configured advance percentage of the obligation's current escrow, and that the escrow equals the amount both parties signed for; it pays principal from the lender to the seller and the pledge fee from the lender to the fee vault; and it calls the market's `attach_pledge` with the `Pledge` PDA as signer. The market accepts that signer only if it is owned by the credit program and equals `PDA(["pledge", obligation], credit)`, only while `Config.credit_advance_bps` is set, and only if the obligation has escrow and no pledge yet. `release_pledge` calls `detach_pledge` the same way once principal is fully repaid. Credit terms are described in §13 Credit.

#### 14.6.4 Market to credit

Every seller escrow release, whether from settlement, waiver or buyer default, passes through `repay_delivered` in the market before the seller is paid. If the obligation carries a pledge, the caller supplies five remaining accounts: the credit program, the pledge, its repayment vault, the lender's canonical USDC account and the market's repayment PDA. The market checks each address by derivation, decodes the 208-byte `Pledge` itself (discriminator, layout version, active state, and the obligation, series and seller it names), and computes the repayment as the lesser of the seller's net proceeds and the outstanding principal.

$$R = \min(N, A - P)$$

Here `R` is the repayment, `N` the seller proceeds of this release after buyer refunds, damages and fees, `A` the pledge's advance and `P` the amount already repaid. All three are USDC base units, so no rounding arises.

The market then moves `R` itself, from the series funding vault to the repayment vault, signed by the `Series` PDA. Only after that does it invoke `repay_from_escrow` with eight accounts: the repayment PDA `credit-repayment`, `obligation` as the only signer, the series, config and obligation read-only, and the pledge, repayment vault and lender account writable, plus the token program. The funding vault and the `Series` signer never enter the credit call. The credit program verifies the signer's derivation under the market program, that the vault holds at least its pending lender cash plus `R`, then pays the lender or records the amount as pending lender cash.

## 14.7 Lookup table and transaction limits

Transactions the service prepares are version-0 messages compiled against one protocol address lookup table, whose address is part of the deployment configuration. Fixed protocol accounts that recur across instructions are then referenced by a one-byte table index instead of a 32-byte key. Before it returns any transaction for signing, the service rejects a message that references more than 64 distinct accounts or serializes to more than 1,232 bytes, the Solana packet limit for legacy and version-0 transactions.

## 14.8 Compute budgets

### 14.8.1 Clearing

`clear_auction` is the heaviest instruction. It accepts exactly its six declared accounts and rejects any remaining account, so extra metas cannot add work. It runs over the first 64 slots with fixed bitonic sorting networks whose comparator counts do not depend on prices or quantities: at most 128 event rows sorted with 1,792 compare-exchanges, and at most 64 priority rows sorted with 672. The 960 tail slots are only checked for inactivity. Clearing makes no token call and fetches no price; its only cross-program call creates the fixed-size `AuctionResult` through the System Program.

The clear transaction requests a 256 KiB heap frame and a compute limit of 1,400,000 units. The market installs its own ascending bump allocator over the full 256 KiB, because the default allocator uses only 32 KiB even when a larger frame is requested. A reviewed bound over the compiled control-flow graph and the runtime's charges caps a successful canonical clear well inside the request.

| Component                                               | Ceiling (CU) |
|---------------------------------------------------------|--------------|
| Two PDA searches, at most 256 attempts at 1,500 CU each | 768,000      |
| Auction arithmetic and allocation                       | 460,000      |
| Entrypoint, Anchor validation, serialization, handler   | 130,000      |
| Other syscalls and transaction setup                    | 12,000       |
| Reviewed ceiling                                        | 1,370,000    |
| Requested limit                                         | 1,400,000    |

The remaining margin under the reviewed ceiling is 30,000 units (2.14%). Executions of 24 adversarial 64-order book shapes against the reviewed binary peaked at 180,314 units, and every clear transaction serialized to 401 bytes. The ceiling is pinned to the program's SHA-256 and to the runtime's cost schedule; a change to the source, toolchain, cost schedule or admission bound requires a new review. Raising admission beyond 64 orders is not a configuration change.

#### 14.8.2 Other heavy instructions

Measured on the same binary, each of the other large instructions stays below the runtime's default allocation of 200,000 compute units per instruction.

| Instruction                | Case                                                 | Compute (CU)     | Transaction |
|----------------------------|------------------------------------------------------|------------------|-------------|
| <code>place_order</code>   | better price evicting the worst open order           | 49,607 to 89,717 | 878 B       |
| <code>seal_delivery</code> | sorting 512 issuance units                           | 33,298           | 325 B       |
| <code>settle_ticket</code> | healthy, missing, capped, default and deferred cases | 60,614 to 66,271 | 655 B       |

#### 14.8.3 Account growth

Solana lets an instruction grow an account by at most 10,240 bytes. The three large raw accounts are therefore created at 10,240 bytes and grown in steps that top up rent from the payer. The book reaches 65,544 bytes in six `grow_book` calls, and the registry reaches 32,776 in three `grow_delivery_registry` calls; both accept only the series' recorded rent payer, so every lamport in them returns to one key at retirement. The health archive reaches 10,800 bytes in one `grow_epoch_health` call, which anyone may send; a grower other than the archive's initializer has its exact top-up recorded on the `Series`. Placement refuses a book that is not fully grown, and the first fresh materialization refuses a registry that is not.

### 14.9 Packet budget for committee signatures

`commit_epoch`, `post_certificate` and `resolve_arbitration` carry two committee signatures without passing them as arguments. The transaction places one native Ed25519 instruction immediately before the Hourglass instruction; the runtime verifies the signatures, and the market reads that instruction back through the instructions sysvar to confirm which keys signed which bytes (§15 Security design). Both signatures share one copy of the message, so the instruction data is a 2-byte header, a 14-byte offsets record per signature, each 32-byte key and 64-byte signature, and the message once.

$$D = 2 + 14 \times k + 96 \times k + m$$

$$D_2 = 222 + m$$

`D` is the Ed25519 instruction data in bytes, `k` the number of signatures and `m` the message length; with `k = 2` the fixed part is 222 bytes.

| Message     | Content                                                                                               | m        | D <sub>2</sub> |
|-------------|-------------------------------------------------------------------------------------------------------|----------|----------------|
| Epoch       | prefix, program, config, domain, series, hour, root, 64-byte bitmap, evidence hash                    | 276<br>B | 498<br>B       |
| Arbitration | prefix, program, config, domain, series, ticket, dispute, verdict, award hash, 21-byte hour<br>bitmap | 270<br>B | 492<br>B       |
| Certificate | SHA-256 of the 318-byte canonical report                                                              | 32 B     | 254<br>B       |

Certificates sign the digest of their report rather than the report itself because `post_certificate` also carries the canonical mutable `Obligation`, its certificate and both evidence accounts; a 318-byte message would not leave room for them under 1,232 bytes. The digest is stored once as `certificate_hash`, and the full report bytes remain available from the service for review before signing. The program also accepts two consecutive single-signature Ed25519 instructions, each carrying its own copy of the message.

## 15 Security design

Hourglass enforces its rules at three layers: the programs, which hold all value and decide every state transition; the service, which prepares transactions and runs the background work; and the participants' own keys. This chapter lists the mechanisms in each layer and, for each one, what it enforces and how the code enforces it. The account layout and instruction set the mechanisms act on are described in §14 Program architecture.

### 15.1 The signing domain

Every message the committee signs is bound to one deployment: one cluster, one market program and one `Config` account. The binding is a 32-byte value, the signing domain, stored in `Config.signature_domain`.

```
domain = SHA-256("HOURGLASSGENESIS-DOMAINV7" || G || programid || config)
```

`G` is the cluster's genesis hash decoded from its base58 text to 32 bytes, `programid` the market program address, `config` the address of the `Config` PDA, and `||` byte concatenation.

`init_config` receives both the genesis hash and the domain as arguments, rejects an all-zero genesis hash, recomputes the domain from the genesis hash, its own program id and the `Config` address it is creating, and fails if the two differ. Both values are then stored. Solana offers programs no native way to read the genesis hash, so the stored value is checked off-chain at deployment against the genesis hash that the cluster's RPC actually reports; deployment and service loading refuse a `Config` whose stored genesis or domain does not match.

Each committee message starts with a versioned prefix ( `HOURGLASS_EPOCH_V7` , `HOURGLASS_CERTIFICATE_V7` or `HOURGLASS_ARBITRATION_V7` ), followed by the program id, the `Config` address and the domain, then the series and the object being attested: the hour, the ticket, or the ticket and dispute. A signature over an epoch can therefore never be replayed as a certificate, on another series, under another configuration of the same program, or on another cluster that runs the same program id. The minute records signed by participants' agents carry the same domain in their context.

The service re-derives the domain on every committee payload it generates, every transaction it prepares and every committee signature it produces. It reads the RPC's genesis hash and the live `Config` account, recomputes the domain, and requires the stored genesis, the stored domain and the indexed copy of `Config` all to agree. Any mismatch stops signing with a configuration error; an RPC failure stops it with an unavailability error. The committee signer then compares the full context of the payload (program, `Config` , domain, series) with that fresh result before it touches a private key. Each persistent cluster uses its own program id.

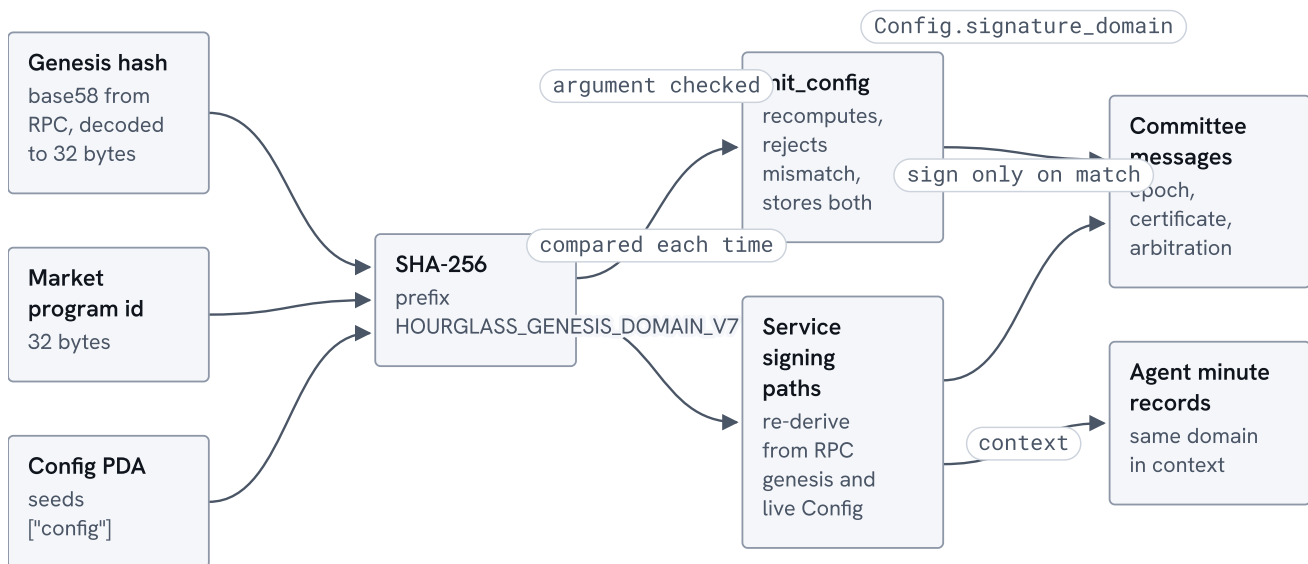


Figure 43 How the signing domain is derived, checked at initialisation and re-derived before every committee signature.

## 15.2 Committee threshold signatures

`Config` holds three committee keys, and `set_committee` and `init_config` require all three to be nonzero and pairwise distinct. Three instructions act on committee authority alone: `commit_epoch`, `post_certificate` and `resolve_arbitration`. Each requires two distinct current committee members to have signed its exact canonical message.

The signatures are not instruction arguments. The transaction carries a native Ed25519 instruction immediately before the Hourglass instruction, and the Solana runtime rejects the whole transaction if any signature in it fails. The native program verifies cryptography but knows nothing about committees, so `verify_committee_signatures` reads the preceding instruction through the instructions sysvar and binds it to Hourglass:

- the instruction belongs to the Ed25519 program, has no accounts, and declares one or two signatures;
- every offsets record points into the same instruction (all three instruction-index fields are `0xFFFF`), and every offset lies past the header, so no signature, key or message can be borrowed from another instruction;
- the message referenced by each record has exactly the expected length and every byte equal to the message the program builds from its own accounts and arguments;
- with one signature, the instruction before it must be a second single-signature proof over the same message;
- the two keys are distinct and both appear in the current `Config.committee`.

Because the program builds the expected message itself, a signature over any other value for the hour, root, healthy bitmap, health figures, reference price, disposition, verdict or covered hours fails. The two verified keys are recorded in each `DeliveryCertificate`. The committee's process for producing evidence and certificates is described in §9 Evidence and the committee.

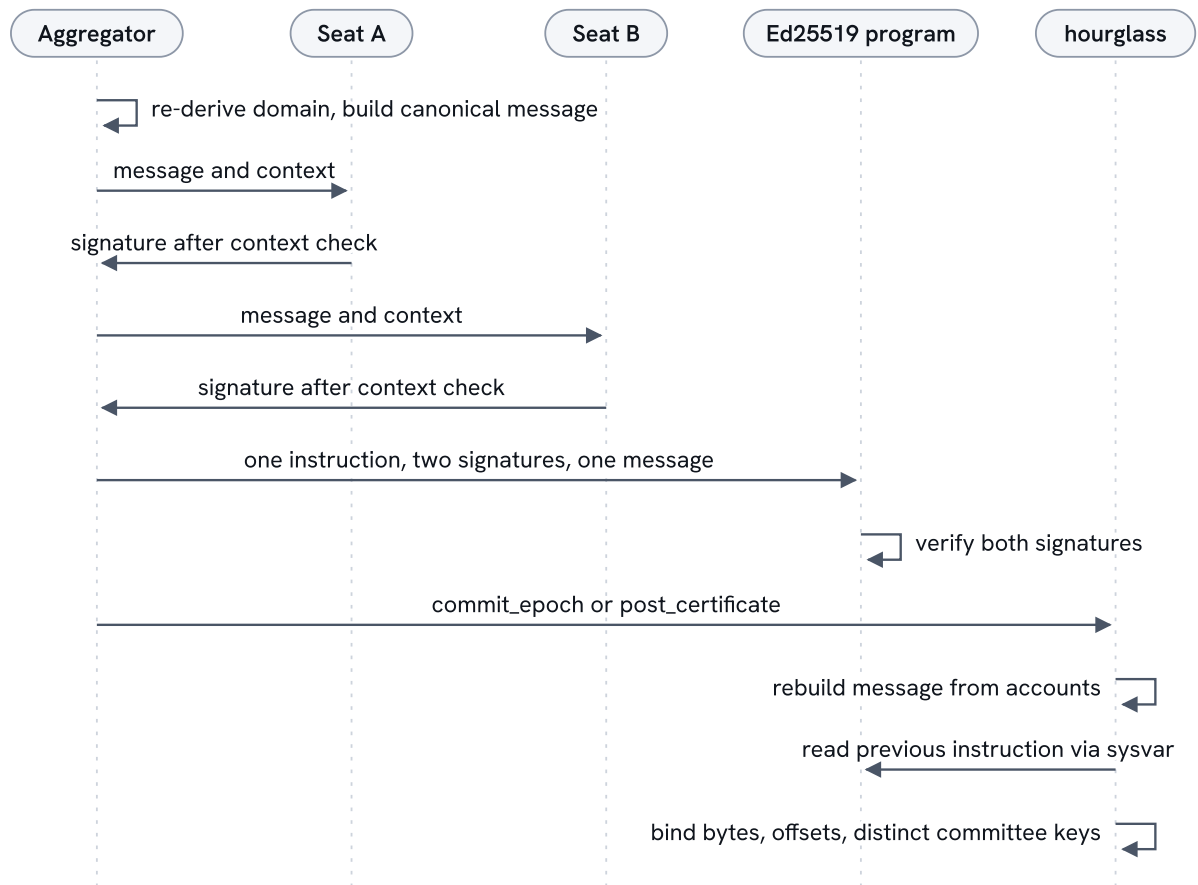


Figure 44 Two committee seats sign one message and the market program checks the native proof that precedes its instruction.

### 15.3 Eligibility on physical actions and claim transfers

A wallet's standing is one `Eligibility` account written by the KYC authority: role bits and the flags `kyc`, `kyb`, `sanctions_clear` and `allowlisted`. A wallet is approved when it has passed KYC, is sanctions-clear and is allowlisted. A seller must additionally have passed KYB and hold the seller role; a buyer-side action needs the buyer or trader role.

The programs read this account at the moment of each action, never a cached approval. Placing an order, registering as a seller, requesting delivery and committing an access key check the acting wallet. Assigning a node, recording acceptance, confirming or finalising access and accepting a cure check both counterparties and also require that no compliance hold is active on the ticket. Accreditation and every probe check the inspector. A pledge checks the seller's KYB and the lender's role. A revoked wallet is thus stopped at its next physical step, wherever it is in the lifecycle.

Cash is deliberately not gated. Refunds, deferred cash collection and settlement payouts proceed for a wallet whose eligibility has lapsed, so a revocation never traps money owed to it. When a counterparty loses eligibility during the delivery week, only the KYC authority may record a compliance hold, only inside `[delivery_start, delivery_end)` and only while a qualification is actually missing. The ticket then accumulates buyer-caused seconds per hour, and `release_compliance_hold` requires restored qualifications. Holds and their effect on settlement are described in §10 Faults, disputes and arbitration.

## 15.4 Frozen-by-default claim accounts

The claim mint's `DefaultAccountState` extension is `Frozen`, so every token account opened for a series' claims starts unable to send or receive. The program thaws one in two instructions, and each requires the owner's eligibility to be approved, the clock to be before notice and the account to be the owner's canonical associated token account. `open_claim_account` is signed by the participant who pays for the account; it further requires payer and owner both to hold the buyer or trader role and the series not to be retiring, creates the canonical account if it is missing and thaws it. `thaw_claim_account` is signed by the KYC authority, thaws an existing account and refuses the program's inventory. When eligibility is revoked, or once notice passes, anyone may call `enforce_claim_freeze`, which freezes the canonical account and returns without effect if it is already frozen.

The transfer hook is a second, independent gate. Even between two thawed accounts, a transfer succeeds only if both endpoints are canonical accounts of currently approved buyer or trader wallets (or the series inventory), the series is not paused and the clock is before notice. A thaw therefore cannot be used to reach a wallet that has since lost its approval.

## 15.5 Pause

`pause_series` lets the admin stop a series in one instruction. It sets `Series.paused` and calls Token-2022's `Pausable` `pause` on the claim mint with the `Series` PDA as authority, so token movements stop at the token program as well as in the hook. While a series is paused, the market rejects new orders, clearing, capacity reservations and outward claim collection; fresh issuance materialised during the pause is recorded as pending mint and caught up later. Cancellations, refunds and cash collection continue, and a delivery request still burns its claim by resuming and re-pausing the mint within the same instruction. A paused series cannot back a new pledge.

## 15.6 Authority rotation

The admin and the KYC authority are replaced in two steps. `propose_admin` and `propose_kyc_authority`, both signed by the current admin, record a nonzero successor in `pending_admin` or `pending_kyc_authority`. `accept_admin` and `accept_kyc_authority` succeed only when signed by exactly that pending key, then install it and clear the pending field, so a

mistyped key never takes control and a rotation can be accepted only once. The admin proposes both successors, so a lost KYC key cannot block recovery; until the new key accepts, eligibility writes, thaws and holds continue under the current KYC authority.

## 15.7 Upgrade authority

`init_config` requires its signer to be the upgrade authority recorded in the market program's ProgramData account, and checks that this ProgramData belongs to the program. The protocol's configuration can therefore only be created by whoever controls the program's code. All three programs share one upgrade authority: a Squads multisig vault with a 48-hour timelock. Any change to market, credit or hook code is visible on chain for 48 hours before it can take effect.

## 15.8 Bounded disputes

Every fault fixes its scope when it is filed. `file_fault` takes an alleged hour interval `[first, end)` that must be nonempty, within the 168-hour week, and no later than the current delivery hour, so a fault can describe past and present hours but never pre-authorise future relief. An access objection's interval runs from the moment of objection to its cure deadline. The interval is stored in the `Dispute` and never rewritten.

A dispute escalates only on a clock. If no probe or cure resolves it within 72 hours, anyone may call `advance_arbitration`; `resolve_arbitration` then needs two committee signatures over a message that names the exact dispute, the verdict, the hash of the external award and a 21-byte bitmap of covered hours. Only the buyer-blocking verdict may cover hours, and each covered hour must lie inside the filed interval, inside the delivery week and before the award. A single inspector's probe never grants damages relief, and a probe that would forfeit the claimant's bond requires a real seller contest first.

Independence is checked by key. An inspector cannot accredit its own seller's node; acceptance and probes require an inspector distinct from both buyer and seller; and the inspector who resolves an access objection must differ from the inspector whose acceptance was objected to. A ticket carries at most one open dispute, and an open dispute blocks its certificate.

## 15.9 Inspector bond exit

An inspector's bond stays in custody through the challenge window of every work it backs and the 72-hour arbitration handoff after it. Every guarded inspector instruction receives the canonical `InspectorBond` account, requires it to hold at least the configured minimum with no withdrawal requested, and raises its liability horizon to at least the end of that work's challenge window plus the 72-hour arbitration handoff. `request_inspector_bond_withdrawal` deactivates the bond in the same instruction, so no further accreditation, acceptance, probe or cure acceptance can be signed against it, and fixes the release time at the later of the

horizon and 72 hours after the request. `withdraw_inspector_bond` pays out only from that time, only to a USDC account of the bond's own authority, and always the full balance. The liability horizon of each kind of work is given in §3 Participants and eligibility.

### 15.10 Durable credits instead of failed payments

A transfer to a closed, frozen or reassigned token account would fail and revert the instruction that contains it, letting one party block a settlement owed to others. Hourglass never sends to an account it has not first validated. Each payout checks that the beneficiary's canonical USDC account is an initialised SPL Token account owned by the beneficiary for the configured mint. If it is not, the exact amount is recorded as a durable credit instead:

`pending_usdc` on the `Order` or `Obligation`, pending buyer or seller cash on the `DeliveryTicket`, or pending lender cash on the `Pledge`. Anyone may later collect a credit into the canonical account; collecting into any other account requires the beneficiary's signature.

### 15.11 No automatic slashing from missing heartbeats

The buyer runs the Delivery Agent on the delivered node and holds its signing key, so whether buyer heartbeats arrive is under the buyer's control. Missing heartbeats therefore move no money. A missing or conflicting record makes the aggregator request a fault process; it never authorises a refund, a collateral slash or a bond forfeiture. An hour in which the buyer is silent can be certified healthy when the seller's telemetry is clean and no fault is open. Money moves only through a committee certificate or a resolved, bonded dispute.

### 15.12 Exposure bounds

Every position has a fixed maximum. A buyer prepays the full notional plus the trade fee, and a ticket's damages are capped by the collateral allocated to that ticket, so the most that can change hands over one ticket is its escrow plus its collateral. A series' open interest is capped by its reserved share of the global cap, at most 250,000 USDC per series and 1,000,000 USDC overall; clearing trims fresh issuance to the remaining headroom, and a series issues at most 512 node-weeks. The reference spot price in a certificate cannot exceed

`Config.reference_spot_max`.

Custody is reconciled at every financial transition: the escrow vault must hold at least the sum of all obligations' escrow notional. Any excess, such as an unsolicited transfer into the vault, is recorded separately as surplus, is never paid to a participant and cannot block a legitimate release. When the series retires, `sweep_retired_surplus` moves it to the fee vault.

### 15.13 One exit for the fees

The fee vault's token authority is the `Config` PDA, and the market program signs a transfer out of it in exactly one place: `claim_buyback`. That instruction runs only as the first of three top-level instructions in one transaction — the claim, the Jupiter swap signed by the buyback authority PDA, and the finish that burns — and each of the three checks the other two through the instructions sysvar. The keeper key that may start a buyback never holds the USDC or the

tokens; between the instructions they sit in the authority's own accounts, and the finish fails unless every claimed USDC was spent and at least the quoted minimum of \$HOURLASS arrived and was burned. A buyback that cannot complete leaves the vault as it was.

### 15.14 Retirement after a complete audit

Retirement closes a series' accounts only after every financial record of the series has been checked. `begin_retire_series` requires the series to be financially closed and past its challenge window, and no sweep or closure runs until three cursors have passed every order, every ticket and every issuance unit, with its obligation and pledge, each free of pending cash, pending mints, open faults and unpaid principal. Each closure pays the payer recorded when the account was funded, so the caller who sends it cannot choose the recipient. The surplus sweep pays only `Config.fee_vault`. A token account closes only at a zero balance and the mint only at zero supply. The mint's permanent delegate is used by one instruction, `sweep_retired_claims`, which runs only after the audit. Participants' records and token accounts stay on chain. The sequence is described in §4 Series and the calendar.

### 15.15 API protections

The API authenticates wallets, not accounts with passwords. A sign-in starts with `auth.nonce`, which returns a challenge bound to the wallet, the requesting origin and its host, with a random 16-byte nonce and a 300-second expiry. The origin must be one of the configured web origins. The wallet signs the message text exactly:

```
<host> wants you to sign in with your Solana account:
<wallet address>

Sign in to Hourglass. This signature does not move assets.

URI: <origin>
Version: 1
Chain ID: solana:<cluster>
Nonce: <32 hex characters>
Issued At: <time>
Expiration Time: <time + 300 s>
Request ID: <challenge id>
```

`auth.verify` checks the Ed25519 signature against the wallet's public key, the expiry, and that the request comes from the same origin, then deletes the challenge so it cannot be used again.

A successful sign-in creates a session token of 32 random bytes. The service stores only its SHA-256 digest and returns the token in the `hg_session` cookie, marked `HttpOnly`, `SameSite=Lax`, `Path=/`, `Secure` in production, with a lifetime of `SESSION_TTL_SECONDS` (3,600 by default, from 60 to 86,400). The web service forwards the API on its own origin, so this cookie is first-party (§16 Service infrastructure). The session body carries a separate 24-byte

CSRF token, and every write made with a session must repeat it in the `X-Hourglass-CSRF` header. On each request the session's roles and identity flags are refreshed from the wallet's current on-chain eligibility.

Further checks apply to every request:

| Mechanism   | Rule                                                                                                                                                     |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Origins     | a request with an <code>Origin</code> outside the configured list is refused before routing                                                              |
| Roles       | each endpoint declares public, session, seller, inspector or lender access                                                                               |
| Idempotency | <code>Idempotency-Key</code> of at most 128 characters on prepare, submit, access and application writes; a key reused with a different body is rejected |
| Rate limits | 120 writes per minute per client address by default; 10 applications per hour                                                                            |
| Input       | JSON only, at most 64 KiB, 10-second request timeout                                                                                                     |
| Responses   | <code>Cache-Control: no-store</code> by default, <code>X-Content-Type-Options: nosniff</code>                                                            |
| Logs        | request id, operation, status, error code and duration; never bodies, URLs, cookies or keys                                                              |

Idempotency keys are scoped to the session and the operation, and the request body is fingerprinted with SHA-256. A retry with the same key and body returns the original result, including while the first attempt is still running; results are kept for one hour. Behind the web service the API sees the proxy's address, so it accepts a forwarded client address for rate limiting only together with the shared proxy secret.

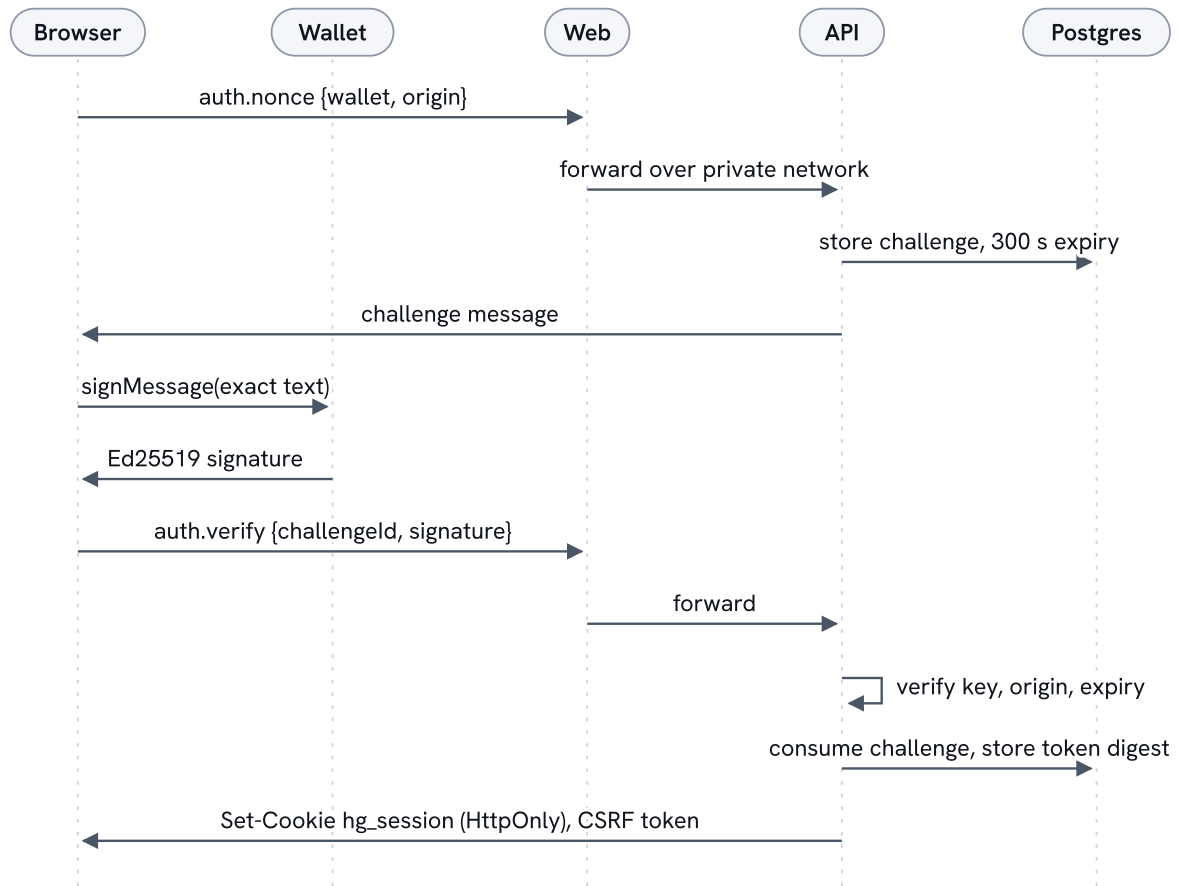


Figure 45 Wallet sign-in: a single-use challenge, a wallet signature over its exact text, and an HttpOnly session cookie.

## 15.16 Server-built transactions

Participants never assemble Hourglass transactions. For every action the API runs a preflight that reports each failed rule with the amount required and available, then builds the version-0 transaction itself from current chain state with the participant's wallet as fee payer, simulates it, and stores the serialized message together with the session, the idempotency key, a digest of the request, the cluster's genesis hash, the program id, the recent blockhash and its last valid block height. A prepared action expires after 60 seconds or when its blockhash does, whichever is first. The wallet signs the message it is shown.

`actions.submit` accepts a signed transaction only if it is the prepared one. The check compares instruction by instruction:

- the prepared instructions appear unchanged and in their original order;
- the only additions are ComputeBudget instructions and Lighthouse assertion instructions, which wallets insert to set fees or guard balances;
- the recent blockhash and the set of required signers are those prepared, with the session wallet as the first signer;
- no account is writable that was read-only in the prepared message;

- every required signature verifies over the submitted message.

The accepted transaction is simulated again before it is relayed. A transaction prepared against a different genesis hash or program id is refused as expired, so a prepared action never crosses deployments.

The receipt is written to Postgres before the transaction is sent, so a lost RPC response cannot lose the authorisation. `actions.status` reads the signature's status, resends the identical signed bytes while the blockhash is valid, marks the receipt expired once the block height passes it, and reports a confirmed transaction as indexed once the read model has caught up to its slot.

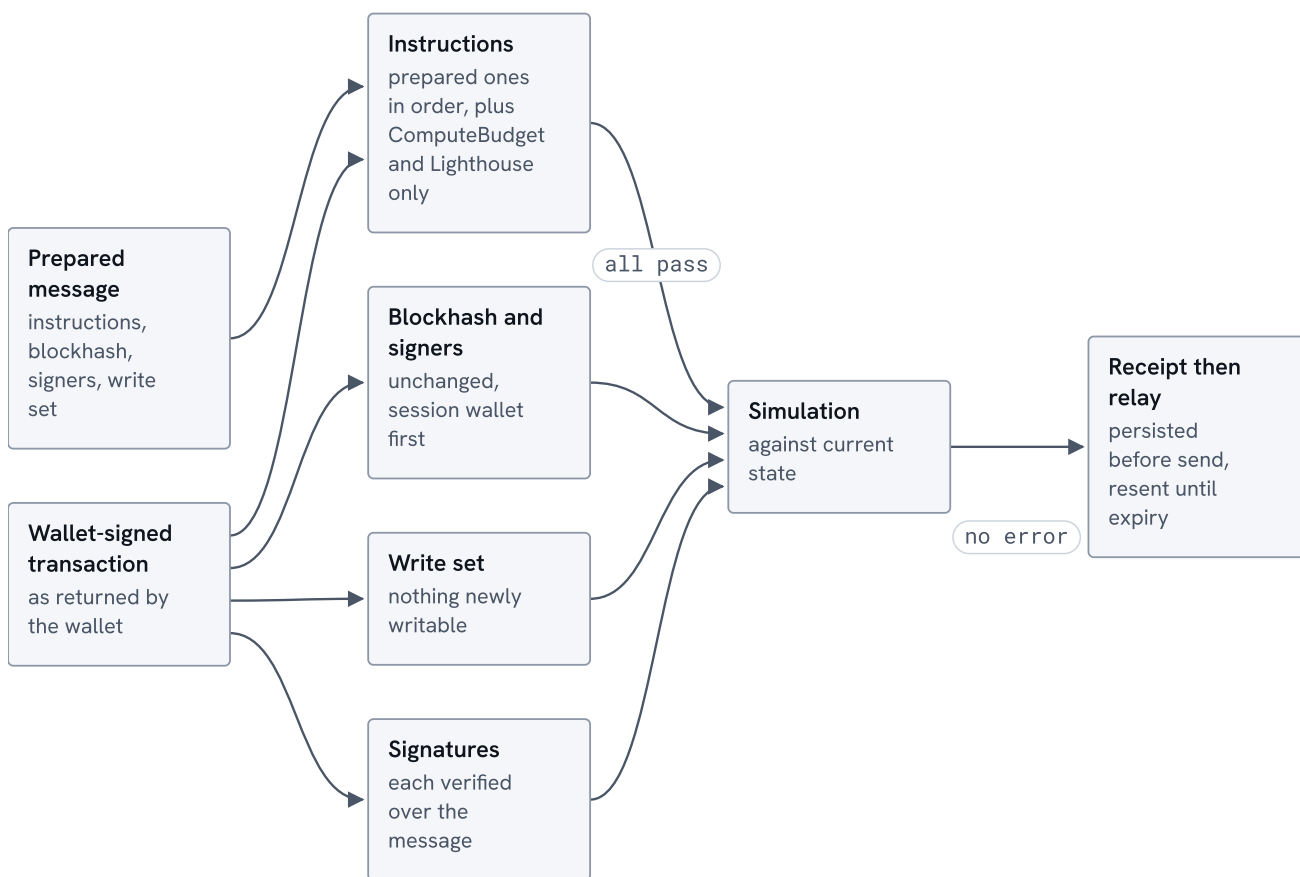


Figure 46 The submit check compares a wallet-signed transaction with the prepared message before anything is relayed.

## 15.17 Key handling

The operator's keys (the admin, the KYC authority, the worker's fee payer and the operator's committee seats) are supplied to the services as environment variables, loaded into memory at start and never written to logs or error responses. The RPC endpoint URL, which can carry an access key, is never logged either: each RPC call produces one log line with its method name only. Errors returned to clients are stable codes with redacted messages.

Participants sign with their own wallets, and no endpoint accepts a wallet secret. The Delivery and Provider Agents hold their Ed25519 seeds in memory, read from a key file on the participant's own machine; their signer checks the full deployment and ticket context before signing anything. Access to a delivered node never passes through Hourglass in readable form: the buyer encrypts an SSH public key to the seller's X25519 key with `x25519-xsa1sa20-poly1305`, the relay stores only that ciphertext, and the chain stores only its SHA-256 hash, which the seller pins when assigning the node.

## 16 Service infrastructure

The Hourglass service is the off-chain half of the system: it serves the site and the app, prepares every transaction a participant signs, runs the work that keeps each series moving, and keeps a read model of the chain. It runs as one Railway project with four services: **web**, **api**, **worker** and **Postgres**. The three programs run on Solana (§14 Program architecture), and the Delivery and Provider Agents run on participants' own machines. None of the services holds participant funds: custody is entirely in program vaults, and every value transfer is a Solana transaction signed by the party it binds or by a program-derived address.

### 16.1 The four services

| Service  | Runtime             | Does                                                                                        | Reaches              |
|----------|---------------------|---------------------------------------------------------------------------------------------|----------------------|
| web      | Next.js             | the site, the documentation, this paper and the app; forwards <code>/v1/*</code> to the API | browsers, api        |
| api      | Node 22 HTTP        | reads, sign-in, the action pipeline, access relay, evidence intake, operator surface        | Postgres, Solana RPC |
| worker   | Node 22             | indexers, crank, series scheduler, committee aggregator, settlement, buyback, rent reclaim  | Postgres, Solana RPC |
| Postgres | Railway<br>Postgres | read model, sessions, prepared actions, relay ciphertext, evidence, applications            | api, worker          |

The api and worker are built from the same backend package and the same configuration. The worker's loops are switched on by the job flags `INDEXER_ENABLED` , `EPOCH_INDEXER_ENABLED` , `CRANK_ENABLED` , `SERIES_SCHEDULER_ENABLED` , `AGGREGATOR_ENABLED` , `SETTLEMENT_ENABLED` , `BUYBACK_ENABLED` and `RENT_RECLAIM_ENABLED` , so background work runs in the worker and the api serves requests. Services reach each other over Railway's private network, and cross-service settings use Railway reference variables, such as `DATABASE_URL=${Postgres.DATABASE_URL}` . No service hardcodes a port; each listens on the `PORT` Railway injects.

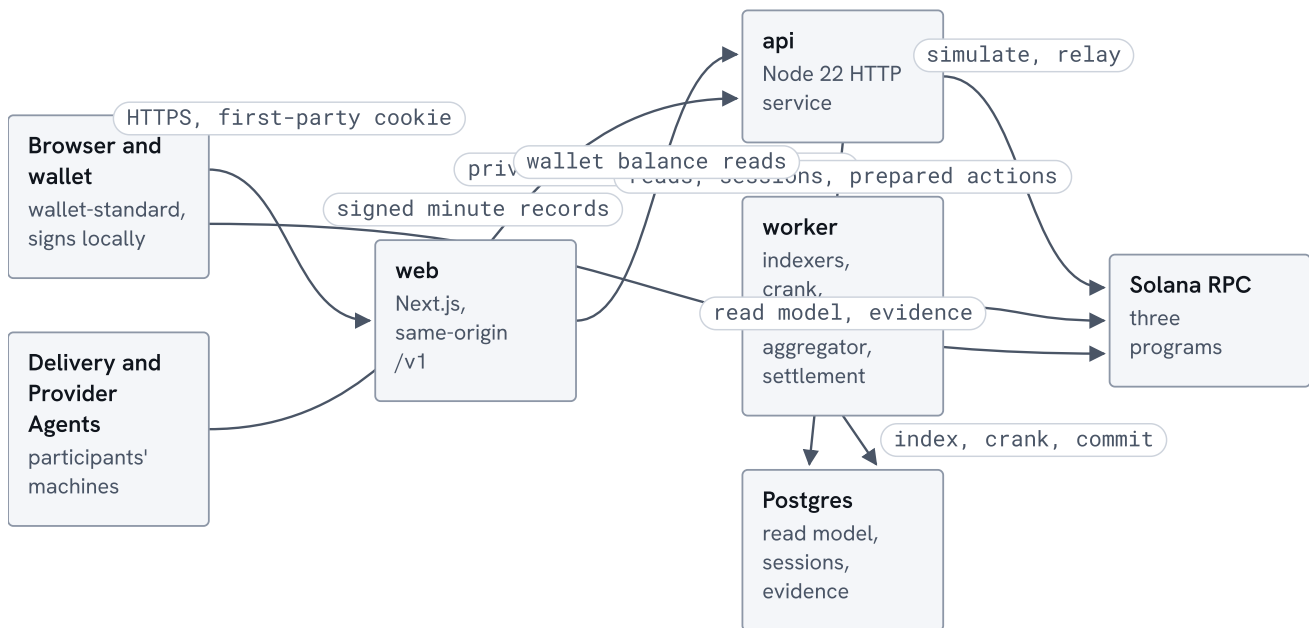


Figure 47 The four Railway services, the participants' machines and Solana, with the data path each connection carries.

#### 16.1.1 web

The web service is a Next.js application that serves the public site, the documentation, the whitepaper and the app. Participants connect real browser wallets through the Wallet Standard and use two of its features: `solana:signMessage` for sign-in and `solana:signTransaction` for prepared transactions. Wallet connections that depend on USB transports are not offered.

The browser never calls the API on another domain. Every `/v1/*` request goes to the web service's own origin, which forwards it to the API's private network address. Railway's public service domains form a public suffix, so a cookie set by one service's subdomain would be third-party to another; forwarding on the site's own origin keeps the session cookie first-party. The forwarder passes only the request headers the API reads ( `Accept` , `Content-Type` , `Cookie` , `Origin` , `User-Agent` , `Idempotency-Key` and the CSRF header), adds the visitor's address together with a shared proxy secret, returns the API's `Set-Cookie` headers unchanged, marks every response `no-store` , and answers with a `NETWORK_ERROR` envelope if the API does not respond within 30 seconds. Server-rendered pages call the API directly over the private network.

#### 16.1.2 api

The api is a Node 22 HTTP service. Railway builds it with `npm ci && npm run build` , which compiles the contract and agent packages, runs `prisma generate` and compiles the server. The build phase cannot reach Railway's private network, so migrations run at start. The api and the worker share one start command, `npm start` , and the `SERVICE_ROLE` variable ( `api` or `worker` ) chooses the process: the api applies `prisma migrate deploy` and then serves, the worker never

migrates and waits until the schema is current. Both listen on `PORT`, the api on the dual-stack address `::` so it is reachable over Railway's private network, and both answer `/health`, which Railway probes and which restarts a service on failure. `/health` answers 200 with status `ok` when the database and the chain are reachable and `waiting`, with the reason, while they are still coming up, so a deploy is not marked failed during a slow dependency start. Sessions and sign-in challenges are stored in Postgres, so a redeploy or a second api replica keeps everyone signed in.

The api serves:

- **reads** from the Postgres read model: series, books, statistics, portfolios, tickets and the seller, inspector and lender consoles;
- **sign-in and sessions** (§15 Security design);
- **the action pipeline**: preflight, prepare, wallet signature, submit and status, with simulation before each transaction is returned and again before it is relayed;
- **the access relay**, which stores the buyer's encrypted SSH key envelope for its ticket: between 16 and 8,192 bytes of ciphertext, accepted only from an approved buyer before the commitment deadline, readable only by the ticket's two parties and kept until the end of the delivery week;
- **the evidence endpoint**, which receives the agents' signed minute records;
- **the operator surface**: participant applications and their approval or revocation, node submissions for accreditation, reference-price records, and an overview of configuration, committee, authorities, fee-payer balances, the series schedule and job state.

An operator approval writes the wallet's roles and identity flags on chain with `set_eligibility`, signed by the KYC authority key, and thaws the claim accounts the wallet already holds; it opens none, because claim accounts are paid for by the participants whose transactions need them (§3 Participants and eligibility). A revocation writes the record again without approval, which every later physical step and claim transfer reads (§15 Security design). A reference-price record names three providers with their published prices per GPU-hour and stores the median and a hash of the sources, which certificates then cite (§11 Settlement).

### 16.1.3 worker

The worker runs its loops one after another in a single cycle, against one shared snapshot of the chain (see Solana RPC usage below). Each loop takes a Postgres transaction-level advisory lock for its run, so a second worker replica skips work the first is already doing, and records a heartbeat with its last start, completion and error code.

| Loop    | Work                                                                                                                                                                                                                                                                                                       |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Indexer | once per poll interval, checks the three programs for new transactions; when one has changed or the reconcile interval has passed, snapshots market and credit program accounts, the token accounts of known owners and vaults, and the clock sysvar, then replaces the stored snapshot in one transaction |

| Loop                 | Work                                                                                                                                                                                                                                                                                            |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Epoch indexer        | within each full snapshot, recovers committed hourly roots and bitmaps from transaction logs, accepting only events whose root equals the ledger's                                                                                                                                              |
| Compliance holds     | after each new snapshot, marks, updates and releases ticket compliance holds with the KYC authority key                                                                                                                                                                                         |
| Crank                | clears due auctions, materializes cleared orders, collects deferred cash and claims, and enforces freezes                                                                                                                                                                                       |
| Series scheduler     | at most every 60 seconds, after a new snapshot, lists the upcoming weekly series in each region, runs their setup instructions and thaws the existing claim accounts of approved wallets                                                                                                        |
| Committee aggregator | when an hour or a certificate is due, opens the series' evidence accounts if needed, verifies minute records, builds hourly roots, signs with the operator's committee seats and posts epochs and certificates                                                                                  |
| Settlement           | seals series, waives unrequested units, finalizes access, records buyer defaults, advances due disputes to arbitration, settles certified tickets, collects their credits and releases each fully closed series cap; the same switch runs action-receipt reconciliation and credit-offer upkeep |
| Buyback              | every 300 seconds, once the fee vault holds 100 USDC, buys \$HOURLASS through Jupiter with up to 1,000 USDC and burns it in one transaction (\$12 Fees)                                                                                                                                         |
| Rent reclaim         | every 60 seconds, moves up to two finished series one transaction further through retirement: the start, the audits in batches, the sweeps and the closures, each refunding the recorded payer (\$4 Series and the calendar)                                                                    |

The indexer writes each snapshot keyed by cluster and program, together with the slot it was read at, and never moves backwards in slots. Each snapshot also records the genesis hash; a different genesis behind the same RPC endpoint starts a new namespace rather than mixing state. The epoch indexer reads transaction logs only from invocation frames that succeeded, so an event emitted by an inner call that later failed is never recorded.

The crank computes each due auction's canonical clearing price with the same `previewAuction` code the browser uses, then sends `clear_auction` with a 256 KiB heap frame and a 1,400,000 compute-unit limit; the program recomputes the price and rejects any other. Every maintenance transaction is permissionless and pays only the beneficiary fixed in the receipt it acts on, so the crank needs no authority beyond a fee payer.

The series scheduler keeps the calendar full. For each region it lists every Monday delivery week from two to eight weeks ahead that has no series, creates it with `create_series` and an explicit share of the global cap, and runs its setup with the admin key: six `grow_book` steps, three `grow_delivery_registry` steps and `init_claim_mint`. The committee aggregator later opens the series' evidence accounts, `init_epoch_ledger`, `init_epoch_health` and one `grow_epoch_health` step, once the series is sealed with requested tickets and its first delivery hour has closed. Series and the auction calendar are described in §4 Series and the calendar.

The committee aggregator holds the operator's committee seats. For every closed delivery hour it verifies the minute records of each ticket, builds the hour's Merkle root and healthy-ticket bitmap, re-derives the signing domain from the RPC and live `Config`, signs, and sends `commit_epoch` with the native Ed25519 instruction. After the challenge window it posts each

ticket's certificate the same way. The settlement loop then runs `settle_ticket` for every certified ticket. Each loop retries failed work on its next cycle and logs the first failure and every thirtieth with a stable code.

#### 16.1.4 Postgres

Postgres holds everything the service needs that the chain does not, and a projection of what it does. Projections never originate a balance: balances are copied from chain state only.

| Group        | Models                                                                                                           | Holds                                                                                                               |
|--------------|------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| Read model   | <code>ChainAccount</code> , <code>ChainCheckpoint</code> ,<br><code>ChainEvent</code> , <code>ChainCursor</code> | decoded accounts by cluster and address, the slot and genesis of each program's snapshot, indexed events            |
| Sessions     | <code>SignInNonce</code> , <code>Session</code>                                                                  | nonce and message digests with origin and expiry; session token digests                                             |
| Actions      | <code>PreparedChainAction</code> ,<br><code>ChainActionReceipt</code> ,<br><code>TransactionIntent</code>        | prepared messages with their blockhash, genesis and program id; signed bytes and receipt status                     |
| Access relay | <code>ChainAccessRelay</code>                                                                                    | ciphertext, sender key, nonce and key hash, unique per ticket and hash                                              |
| Evidence     | <code>EvidenceObject</code>                                                                                      | content hash, storage key, kind, owner wallet, ticket and retention                                                 |
| Applications | <code>PilotApplication</code> and participant applications                                                       | role, organisation, contact, region interest; idempotency key                                                       |
| Jobs         | <code>BackgroundJob</code> , <code>WorkerHeartbeat</code>                                                        | deduplication key, attempts, next run, lease; each worker loop's status, last start, last completion and error code |

Evidence objects are addressed by the SHA-256 of their content, so a record submitted twice is stored once, and a party holding its own copy can prove that the stored record is the one the committee hashed.

## 16.2 Configuration

Each service reads its configuration from Railway variables. The api and worker refuse to start in chain mode without a database URL, and report as missing any of the three program ids, the USDC mint or the lookup table address.

| Variable                                                 | Purpose                                                                | Default           |
|----------------------------------------------------------|------------------------------------------------------------------------|-------------------|
| <code>SERVICE_MODE</code> , <code>SOLANA_CLUSTER</code>  | <code>chain</code> on <code>devnet</code> or <code>mainnet-beta</code> | cluster<br>devnet |
| <code>SOLANA_RPC_URL</code> , <code>SOLANA_WS_URL</code> | RPC endpoints                                                          | —                 |

| Variable                                                                                                                                                                                                                                                                               | Purpose                                                                                                                                               | Default                                                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| <code>HOURLASS_PROGRAM_ID</code> ,<br><code>HOURLASS_CREDIT_PROGRAM_ID</code> ,<br><code>HOURLASS_HOOK_PROGRAM_ID</code>                                                                                                                                                               | program addresses                                                                                                                                     | —                                                      |
| <code>USDC_MINT</code> , <code>LOOKUP_TABLE_ADDRESS</code>                                                                                                                                                                                                                             | settlement mint, protocol lookup table                                                                                                                | —                                                      |
| <code>WEB_ORIGINS</code> , <code>PROXY_SECRET</code>                                                                                                                                                                                                                                   | allowed origins, forwarder secret                                                                                                                     | —                                                      |
| <code>SESSION_TTL_SECONDS</code> ,<br><code>WRITE_RATE_LIMIT_PER_MINUTE</code>                                                                                                                                                                                                         | session lifetime, write limit                                                                                                                         | 3,600; 120                                             |
| <code>ADMIN_KEYPAIR</code> , <code>KYC_AUTHORITY_KEYPAIR</code> ,<br><code>WORKER_KEYPAIR</code> , <code>COMMITTEE_KEYPAIRS</code>                                                                                                                                                     | operator keys                                                                                                                                         | —                                                      |
| <code>COMPUTE_UNIT_PRICE_MICROLAMPORTS</code>                                                                                                                                                                                                                                          | priority fee on service-sent transactions                                                                                                             | 0                                                      |
| <code>INDEXER_ENABLED</code> , <code>EPOCH_INDEXER_ENABLED</code> ,<br><code>CRANK_ENABLED</code> , <code>SERIES_SCHEDULER_ENABLED</code> ,<br><code>AGGREGATOR_ENABLED</code> , <code>SETTLEMENT_ENABLED</code> ,<br><code>BUYBACK_ENABLED</code> , <code>RENT_RECLAIM_ENABLED</code> | worker loop switches                                                                                                                                  | <code>true</code>                                      |
| <code>RPC_POLL_INTERVAL_SECONDS</code>                                                                                                                                                                                                                                                 | worker cycle and newest-signature check,<br>30 to 60                                                                                                  | 30                                                     |
| <code>RPC_RECONCILE_INTERVAL_SECONDS</code>                                                                                                                                                                                                                                            | longest time between full snapshots, 60 to<br>600                                                                                                     | 600                                                    |
| <code>INDEXER_INTERVAL_MS</code>                                                                                                                                                                                                                                                       | loop interval of the local-chain service                                                                                                              | 2,000                                                  |
| <code>INITIAL_FEE_PAYER_TRANSACTIONS</code> ,<br><code>INITIAL_ONBOARDING_WALLETS</code>                                                                                                                                                                                               | transactions and onboarded wallets that<br>size the fee and rent reserve<br><code>programs:init</code> transfers to the admin,<br>KYC and worker keys | 1,000; 10                                              |
| <code>HOURLASS_TOKEN_MINT</code>                                                                                                                                                                                                                                                       | the \$HOURLASS mint, registered by<br><code>programs:init</code> or <code>programs:set-<br/>token-mint</code>                                         | —                                                      |
| <code>BUYBACK_MIN_USDC_MICROS</code> ,<br><code>BUYBACK_MAX_USDC_MICROS</code>                                                                                                                                                                                                         | smallest vault balance that starts a<br>buyback, largest batch                                                                                        | 100 USDC;<br>1,000 USDC                                |
| <code>BUYBACK_SLIPPAGE_BPS</code> ,<br><code>BUYBACK_INTERVAL_SECONDS</code>                                                                                                                                                                                                           | quote slippage, loop interval                                                                                                                         | 100; 300                                               |
| <code>JUPITER_SWAP_API_URL</code>                                                                                                                                                                                                                                                      | quote and swap-instruction endpoint                                                                                                                   | <code>https://lite-<br/>api.jup.ag/sw<br/>ap/v1</code> |

Initialisation inputs are supplied the same way and written on chain by `init_config` and `configure_auction`: the auction time (`AUCTION_UTC_TIME` , 16:00 by default), the minimum order notional (100 USDC), the per-wallet open-order limit (8), the failed-auction grace (3,600 seconds), the reference-price ceiling (20 USDC per GPU-hour) and the manual-certificate timeout (86,400 seconds).

### 16.3 Solana RPC usage

Every RPC call costs quota, so the service budgets them. Only the worker polls the chain. The api serves reads from Postgres; its own RPC calls are the deployment identity check and the work of preparing, submitting and tracking participants' transactions. The genesis hash is fetched once per process and RPC endpoint and held for the process lifetime; a failed fetch is not kept. The deployment identity check that guards ordinary reads (program accounts present, `Config` owned by the market and carrying the expected genesis and domain) is cached for 30 seconds; signing paths re-read `Config` every time (§15 Security design).

Each worker cycle shares one refresh of the chain, taken at most once per `RPC_POLL_INTERVAL_SECONDS`, 30 seconds by default. The refresh first asks each of the three programs for its newest finalized transaction signature. When none has changed and the last full snapshot is younger than `RPC_RECONCILE_INTERVAL_SECONDS`, 600 seconds by default, the worker advances the stored chain clock by the elapsed time and every loop works from the snapshot it already holds; otherwise it takes a full snapshot. The compliance-hold and scheduler loops run only after a new full snapshot, and the aggregator runs only when an hour or a certificate is due. Background reads wait so that no more than 12 leave the worker, and no more than 2 leave the api, per rolling minute and endpoint. The api's preflight, preparation and submission of participants' transactions, and a worker transaction once its simulation has succeeded, are outside that budget.

The service's own diagnostic measures this budget by running the real api and worker against a local JSON-RPC fixture and counting every request the fixture receives. While the programs are not yet deployed, the api and worker together make at most 2 calls in any minute. With the programs deployed and every loop switched on, the idle worker makes 3 calls a minute, at most 12 in any minute including its startup, and the api makes none while serving repeated reads. The minute that contains one auction clear and one epoch commit takes 20 worker calls.

When a snapshot runs, it reads program accounts once per program and token accounts in batches of 100 addresses per call, all pinned to a minimum context slot so that a later read never reports an earlier state. Epoch recovery pages through at most four pages of 1,000 signatures per series, fetching eight transactions at a time, and stops as soon as every committed hour is accounted for. Every call has an 8-second timeout and is not retried automatically on rate limiting; the loop's next cycle retries instead. Each HTTP or WebSocket request writes one JSON log line containing only its method name, never the endpoint URL, which can carry an access key.

Transactions the service sends itself, from the crank, the scheduler, the aggregator, settlement and rent reclaim, carry a compute-unit price set by `COMPUTE_UNIT_PRICE_MICROLAMPORTS`. They can also be routed through a configured sender endpoint with a fixed tip ( `SENDER_URL`, `SENDER_TIP_ACCOUNT`, `SENDER_TIP_LAMPORTS` ). Each is sent with up to three RPC retries and confirmed within a 30-second wait; when confirmation does not arrive, the loop rereads chain state before it tries again, so work that did land is never repeated. Participant transactions keep whatever compute-budget instructions their wallet adds.

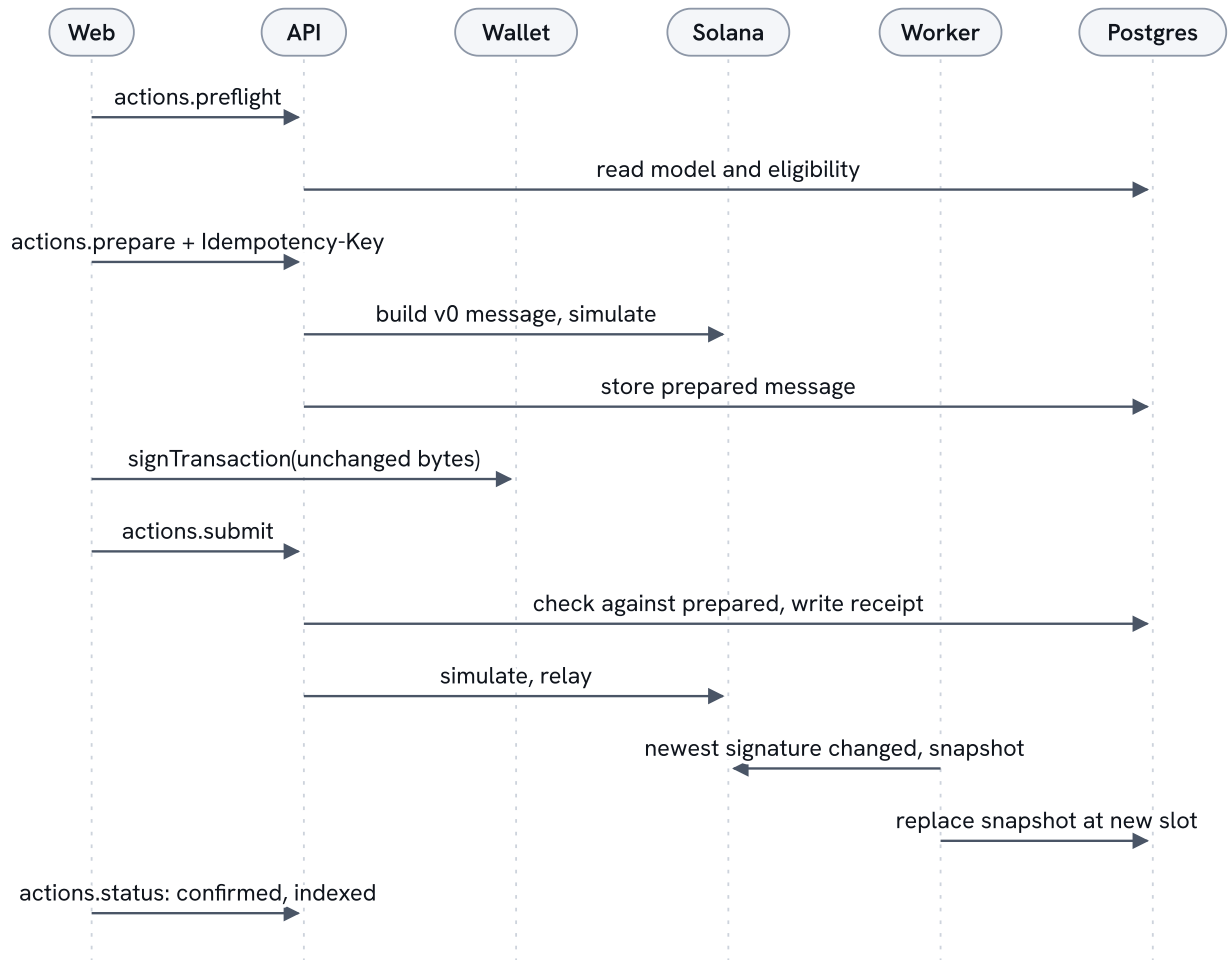


Figure 48 One participant action from the app through the API, the wallet and the chain, until the worker's index reflects it.

## 16.4 Agents

The Delivery Agent and the Provider Agent are single-file Node 22 command-line programs with no dependencies outside Node. Participants run them on their own machines: the buyer runs the Delivery Agent on the delivered node, and the seller runs the Provider Agent on its management plane. Each takes its Ed25519 seed from a key file on that machine, keeps it in memory, and requires an explicitly configured list of gating XID codes before it starts.

Both agents sample once per delivery minute, aligned to the delivery start, and stop at the end of the week. The Delivery Agent reports the hash of the visible GPU UUID set, the count of gating XID events, whether all eight GPUs are visible and whether the node is reachable. The Provider Agent reports BMC power and link state and DCGM health. Each record is a version-5 minute record whose context names the program, `Config`, signing domain, series, ticket and delivery start; it carries the minute index (0 through 10,079), a recent Solana slot, the signer's public key and an Ed25519 signature.

Agents post each signed record to the service's evidence endpoint. The api verifies the signature and context and stores the record by content hash; a repeated record returns the original receipt. The Delivery Agent also serves `GET /.well-known/hourglass/health?nonce=<32-`

`byte hex>` , through which an inspector probes the node: it samples the hardware afresh and signs the inspector's challenge with the observation, without reading workload files or keys. How minute records become healthy hours is described in §9 Evidence and the committee.

## 16.5 Program deployment

Programs are deployed in a fixed order, and each step is checked before the next.

1. **Production build.** Each program is compiled on its own with Solana platform-tools 1.52 and without the test-clock feature, and the build records that it is a production build. IDLs are generated from the same sources.
2. **Fixed buffer.** Each program binary is written to a buffer account and deployed from it.
3. **Byte verification.** The deployed program data of each of the three programs is read back and its SHA-256 compared with the local build. A mismatch stops the deployment.
4. **Initialisation.** `init_config` is signed by the market's upgrade authority with the committee, the KYC authority, the auction policy, the reference-price ceiling, the manual-certificate timeout, the minimum order notional, and the genesis hash and signing domain derived from the cluster's RPC. The hook program is confirmed at the address the market writes into every claim mint's TransferHook extension; the credit policy is set with `configure_credit` ; `create_spec` records the HG-H100x8 specification hash; the protocol lookup table is created and extended; and the scheduler creates the first series in each region.

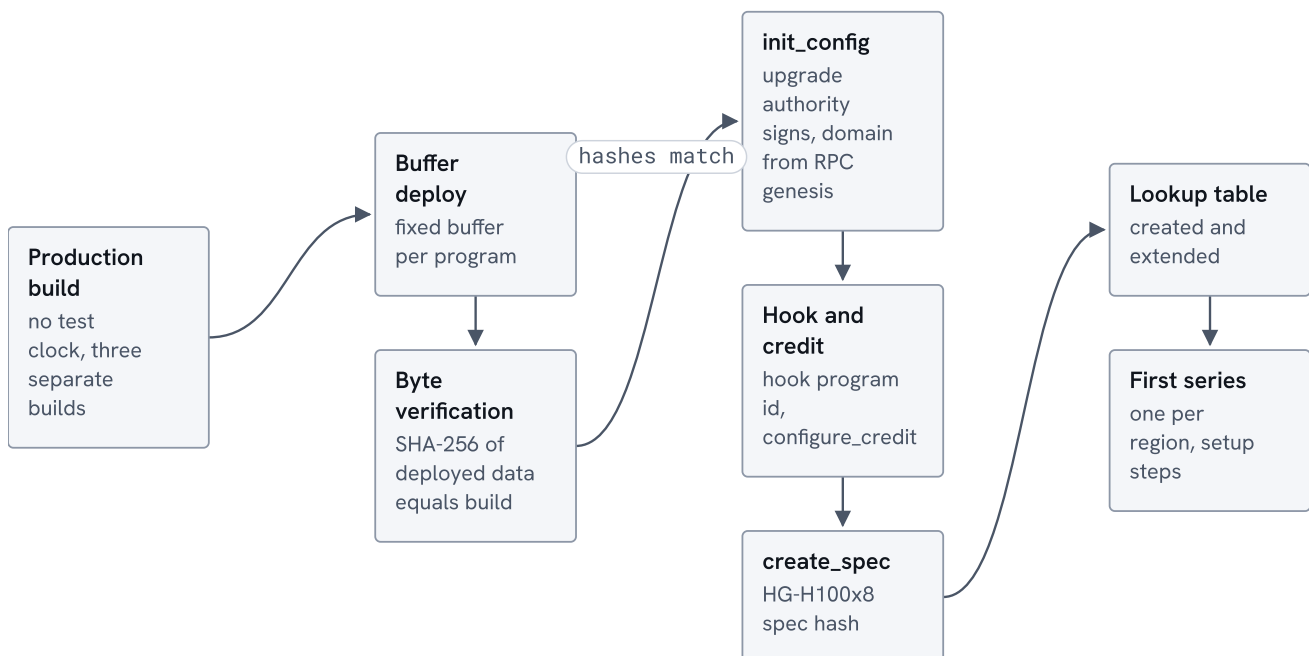


Figure 49 Program deployment from the production build to the first listed series, with the check that gates each step.

## 17 Interfaces

Participants reach Hourglass through two interfaces: a public HTTP API under the base path `/v1`, served at the service's API base URL and on the web app's own origin, and the TypeScript SDK `@hourglass/contract`, which wraps that API and carries the protocol's pure arithmetic. Both describe the same operations with the same types. Every state change still ends as a Solana transaction that the participant's own wallet signs; the API prepares it and relays it, and the programs decide it (§14 Program architecture).

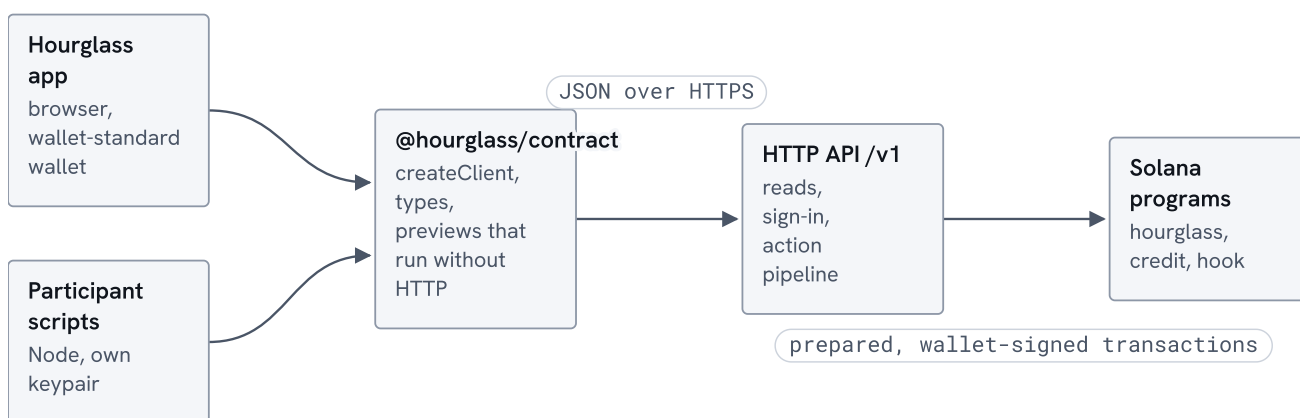


Figure 50 How the app and participants' scripts reach the programs through the SDK and the HTTP API.

### 17.1 Conventions

Requests use JSON. `GET` operations take their inputs as query parameters; `POST` operations take a JSON body of at most 64 KiB with `Content-Type: application/json`. Values follow fixed encodings:

| Kind                 | Encoding                                                    | Example                             |
|----------------------|-------------------------------------------------------------|-------------------------------------|
| USDC amounts         | integer string in base units of $10^{-6}$ USDC              | "2956800000" is 2,956.80 USDC       |
| Prices               | integer string, base units per GPU-hour, multiple of 10,000 | "2200000" is 2.20 USDC per GPU-hour |
| Hours and weeks      | exact decimal string, at most six decimal places            | "1.68" , "4.000000"                 |
| Quantities           | integer node-weeks, 1 to 512                                | 1                                   |
| Times                | ISO 8601 UTC                                                | "2026-11-09T00:00:00.000Z"          |
| Wallets and accounts | base58 public keys                                          | —                                   |

Amounts never travel as floating-point numbers, so a client can reproduce every figure the programs compute to the base unit. Identifiers for series, orders, tickets, disputes and pledges are taken from responses rather than constructed. Three headers carry protocol state: the `hg_session` cookie, `X-Hourglass-CSRF` on writes made with a session, and `Idempotency-Key` on operations that create work. Every response carries `X-Request-Id`, and a rate-limited response carries `Retry-After`.

## 17.2 The result envelope

Every response, successful or not, has the same shape.

```
type ApiResult<T> =
  | { ok: true; data: T; meta: Meta }
  | { ok: false; error: ApiError; meta: Meta };

interface ApiError {
  code: ErrorCode;           // stable machine code, e.g. "DEADLINE_PASSED"
  message: string;           // one sentence for people
  retryable: boolean;
  details: Record<string, string | number | boolean | null>;
  fieldErrors: Record<string, string>;
  retryAfterSeconds: number | null;
}

interface Meta {
  requestId: string; idempotentReplay: boolean;
  environment: "devnet" | "mainnet" | ...; source: "chain_index" | "database" | ...;
  isMarketData: boolean; asOf: string; chainSlot: string | null; contractVersion: string;
}
```

`code` is one of a fixed list that clients render directly: identity states ( `KYC_PENDING`, `KYB_REQUIRED`, `SANCTIONS_BLOCKED`, `NOT_ALLOWLISTED` ), market states ( `SERIES_PAUSED`, `DEADLINE_PASSED`, `CAP_REACHED`, `CLAIMS_FROZEN` ), resource states ( `INSUFFICIENT_FUNDS`, `INSUFFICIENT_COLLATERAL`, `INSUFFICIENT_CAPACITY` ), request faults ( `VALIDATION_ERROR`, `CSRF_INVALID`, `IDEMPOTENCY_CONFLICT`, `TRANSACTION_MISMATCH`, `TRANSACTION_EXPIRED` ) and service states ( `RATE_LIMITED`, `RPC_UNAVAILABLE`, `CHAIN_UNAVAILABLE` ). A transaction the programs reject returns `CHAIN_REJECTED`, or the specific code that the program's error maps to. `fieldErrors` names the offending input fields of a `VALIDATION_ERROR`.

The HTTP status follows the code: 400 for validation, 401 for missing or expired authentication, 403 for permission, identity, origin and CSRF failures, 404 for unknown resources, 409 for domain conflicts, 429 for rate limits and 503 for an unavailable RPC or chain. `retryable` is true exactly for 429, 503 and internal errors, and `retryAfterSeconds` gives the wait. `meta.chainSlot` is the slot of the read model that answered, and `meta.idempotentReplay` is true when the response repeats an earlier one.

## 17.3 Operations

Each operation has one method and path. Access is `public` , `session` (any signed-in wallet), or a role that the wallet's current on-chain eligibility must hold. The health check is served at `/health` , outside `/v1` .

| Group            | Operation                                                                                                                                        | Method        | Path                                                                       | Access              |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|---------------|----------------------------------------------------------------------------|---------------------|
| Service          | <code>health</code> , <code>environment</code>                                                                                                   | GET           | <code>/health</code> ,<br><code>/v1/environment</code>                     | public              |
| Spec             | <code>spec.get</code>                                                                                                                            | GET           | <code>/v1/spec</code>                                                      | public              |
| Series           | <code>series.list</code> , <code>series.get</code>                                                                                               | GET           | <code>/v1/series</code> ,<br><code>/v1/series/detail</code>                | public              |
| Markets          | <code>markets.book</code> , <code>statistics</code>                                                                                              | GET           | <code>/v1/markets/book</code> ,<br><code>/v1/statistics</code>             | public              |
| Previews         | <code>preview.settlement</code> ,<br><code>preview.collateral</code> , <code>preview.calendar</code>                                             | POST          | <code>/v1/previews/{settlement,<br/>collateral, calendar}</code>           | public              |
| Auth             | <code>auth.nonce</code> , <code>auth.verify</code>                                                                                               | POST          | <code>/v1/auth/nonce</code> ,<br><code>/v1/auth/verify</code>              | public              |
| Auth             | <code>auth.session</code> / <code>auth.logout</code>                                                                                             | GET /<br>POST | <code>/v1/auth/session</code> ,<br><code>/v1/auth/logout</code>            | public /<br>session |
| Portfolio        | <code>portfolio.get</code>                                                                                                                       | GET           | <code>/v1/portfolio</code>                                                 | session             |
| Tickets          | <code>tickets.get</code>                                                                                                                         | GET           | <code>/v1/tickets/detail</code>                                            | session             |
| Seller           | <code>seller.get</code>                                                                                                                          | GET           | <code>/v1/seller</code>                                                    | seller              |
| Inspector        | <code>inspector.get</code>                                                                                                                       | GET           | <code>/v1/inspector</code>                                                 | inspector           |
| Lender           | <code>lender.get</code>                                                                                                                          | GET           | <code>/v1/lender</code>                                                    | lender              |
| Credit<br>offers | <code>credit.offers.create</code> , <code>.sign</code> , <code>.accept</code> ,<br><code>.cancel</code> / <code>.list</code> , <code>.get</code> | POST /<br>GET | <code>/v1/credit/offers{,<br/>/sign, /accept, /cancel,<br/>/detail}</code> | session             |
| Actions          | <code>actions.preflight</code> , <code>actions.prepare</code> ,<br><code>actions.submit</code>                                                   | POST          | <code>/v1/actions/{preflight,<br/>prepare, submit}</code>                  | session             |
| Actions          | <code>actions.status</code>                                                                                                                      | GET           | <code>/v1/actions/status</code>                                            | session             |
| Access           | <code>access.get</code> / <code>access.handoff</code> ,<br><code>access.confirm</code>                                                           | GET /<br>POST | <code>/v1/access/{detail,<br/>handoff, confirm}</code>                     | session             |
| Committee        | <code>committee.payload</code>                                                                                                                   | POST          | <code>/v1/committee/payload</code>                                         | session             |
| Participants     | <code>participant.apply</code> /<br><code>participant.status</code>                                                                              | POST /<br>GET | <code>/v1/participant/{applicat<br/>ions, status}</code>                   | session             |
| Nodes            | <code>nodes.submit</code> / <code>nodes.list</code>                                                                                              | POST /<br>GET | <code>/v1/nodes</code>                                                     | session             |

| Group    | Operation                                                                                                                                                                  | Method     | Path                                                                                 | Access           |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|--------------------------------------------------------------------------------------|------------------|
| Evidence | <code>evidence.context</code> / <code>evidence.submit</code>                                                                                                               | GET / POST | <code>/v1/evidence/context</code> , <code>/v1/evidence</code>                        | public           |
| Operator | <code>operator.overview</code> , <code>operator.applications</code> / <code>operator.approve</code> , <code>operator.revoke</code> , <code>operator.reference_price</code> | GET / POST | <code>/v1/operator/{overview, applications, approve, revoke, reference-price}</code> | operator wallets |
| Pilot    | <code>pilot.apply</code>                                                                                                                                                   | POST       | <code>/v1/pilot/applications</code>                                                  | public           |

`series.list` pages with `cursor` and a `limit` from 1 to 100 and filters by region and phase. `tickets.get` and `access.get` return data only to the ticket's buyer and seller. `committee.payload` returns the exact bytes for an epoch, certificate or arbitration signature, to current committee members and the ticket's parties: `messageBase64` to sign, `messageHash` , `encoding` ( `raw` , or `sha256` for certificates, whose `canonicalReportBase64` carries the full report for review), `requiredSignatures: 2` and the current `committeeKeys` . `evidence.submit` is public because each record carries its own agent signature; the API verifies it before storing anything. The operator surface ( `/v1/operator/*` ) is restricted to the configured operator wallets.

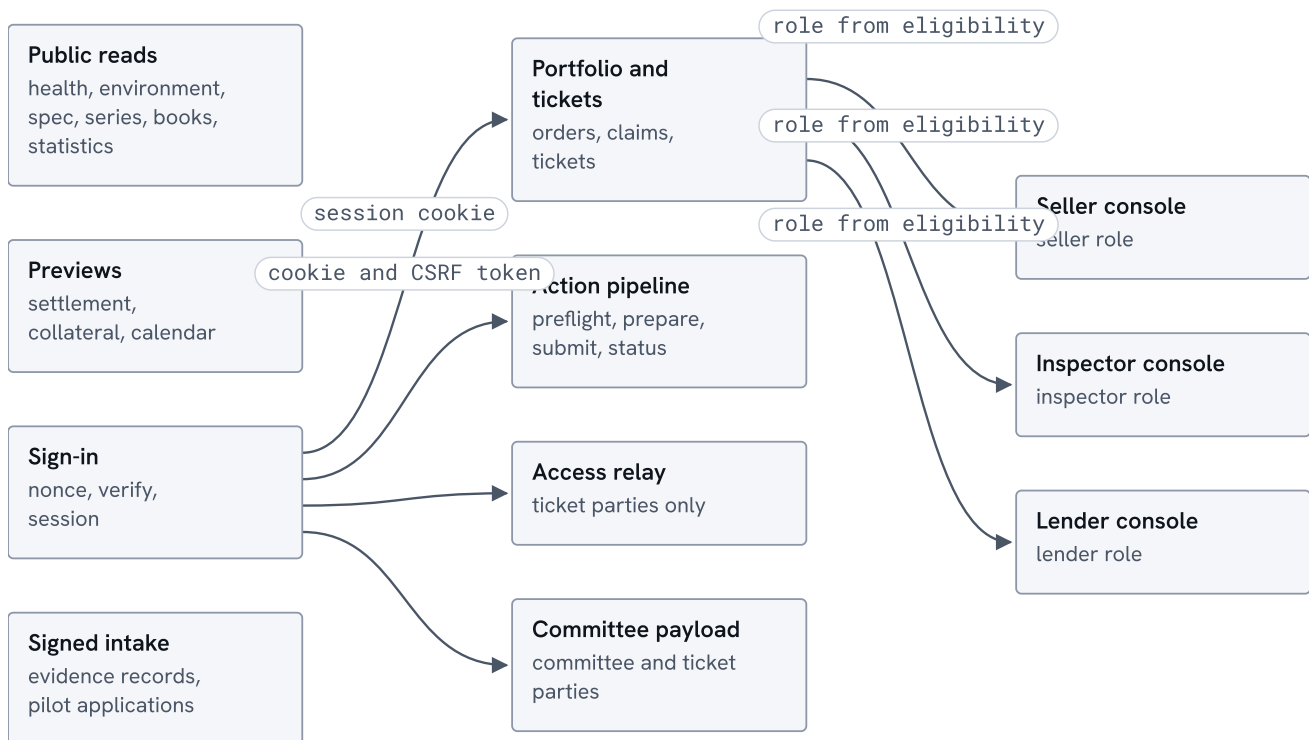


Figure 51 Operation groups by the access they require: none, a signed-in wallet, or a role held in current on-chain eligibility.

## 17.4 The action pipeline

Every operation that changes chain state uses one pipeline of four calls. The client describes the action as data, the API checks it and builds the transaction, the wallet signs the bytes unchanged, and the API relays and tracks them. Actions are a discriminated union on `kind` that names the instruction and its inputs, such as `place_order`, `request_delivery`, `set_access_hash`, `file_fault`, `collect_cash` or `pledge_obligation`.

`actions.preflight` evaluates the action against current state and returns every rule with whether it passed, the amount required and the amount available. For a bid of one node-week at an example price of 2.20 USDC per GPU-hour:

```
POST /v1/actions/preflight
{"action":{"kind":"place_order","seriesId":"<series id>","side":"bid",
          "pricePerGpuHour":"2200000","nodeWeeks":1}}

200 {"ok":true,"data":{"
  "allowed":true,
  "checks":[{"code":"INSUFFICIENT_FUNDS","passed":true,
    "required":"2971584000","available":"<balance>","unit":"USDC base unit",...},...],
  "notional":"2956800000","buyerFunding":"2971584000","collateralRequired":"0",
  "fees":{"trade":"14784000","delivery":"73920000","pledge":"0"},
  "requiredSigners":["<wallet>"],"expiresAt":"<now + 60 s>","simulation":true},
  "meta":{"...}}
```

The notional is  $2,200,000 \times 1,344$  GPU-hours = 2,956,800,000 base units. The buyer prepays it plus the trade fee of 0.5%, rounded up to the base unit: 14,784,000, for a funding requirement of 2,971,584,000 base units (2,971.584 USDC). The delivery fee of 2.5%, 73,920,000 base units, is shown for reference; it is charged to the seller at settlement on delivered value (§12 Fees). When the bidder's claim account for the series is missing or frozen, the response also carries `claimAccountSetup` with the paying wallet, the target wallet, the account address and the rent in lamports quoted from the cluster, and the prepared transaction opens the account before the bid (§7 Claims and transfer).

`actions.prepare` takes the same body and an `Idempotency-Key`, re-runs the preflight, builds and simulates the transaction, and returns it for signing:

```
POST /v1/actions/prepare      Idempotency-Key: <key A>
200 {"ok":true,"data":{"id":"<prepared id>","action":{"..."},"preflight":{"..."},
  "signing":{"kind":"solana_v0","signable":true,"transactionBase64":"<v0 transaction>",
    "recentBlockhash":"<hash>","lastValidBlockHeight":"<height>"},
  "requiredSigners":["<wallet>"],"expiresAt":"<now + 60 s>"},"meta":{"...}}

POST /v1/actions/submit      Idempotency-Key: <key B>
{"preparedActionId":"<prepared id>","mode":"signed","signedTransactionBase64":"<signed>"}
200 {"ok":true,"data":{"id":"<receipt id>","status":"submitted","signature":"<signature>",
  "chainSlot":null,"indexed":false,"pollAfterMs":500},"meta":{"...}}
```

The wallet signs `transactionBase64` as it is; wallets may add compute-budget and Lighthouse assertion instructions, and nothing else (§15 Security design). The client then polls `actions.status` with the receipt id after `pollAfterMs` until `status` is `confirmed` and `indexed` is true, or `failed` with an `error`. A failed receipt is final for that preparation; an expired one needs a new preparation and a new signature. Actions that need more than one wallet, such as `pledge_obligation`, list every signer in `requiredSigners`, and each signs the same message before submission.

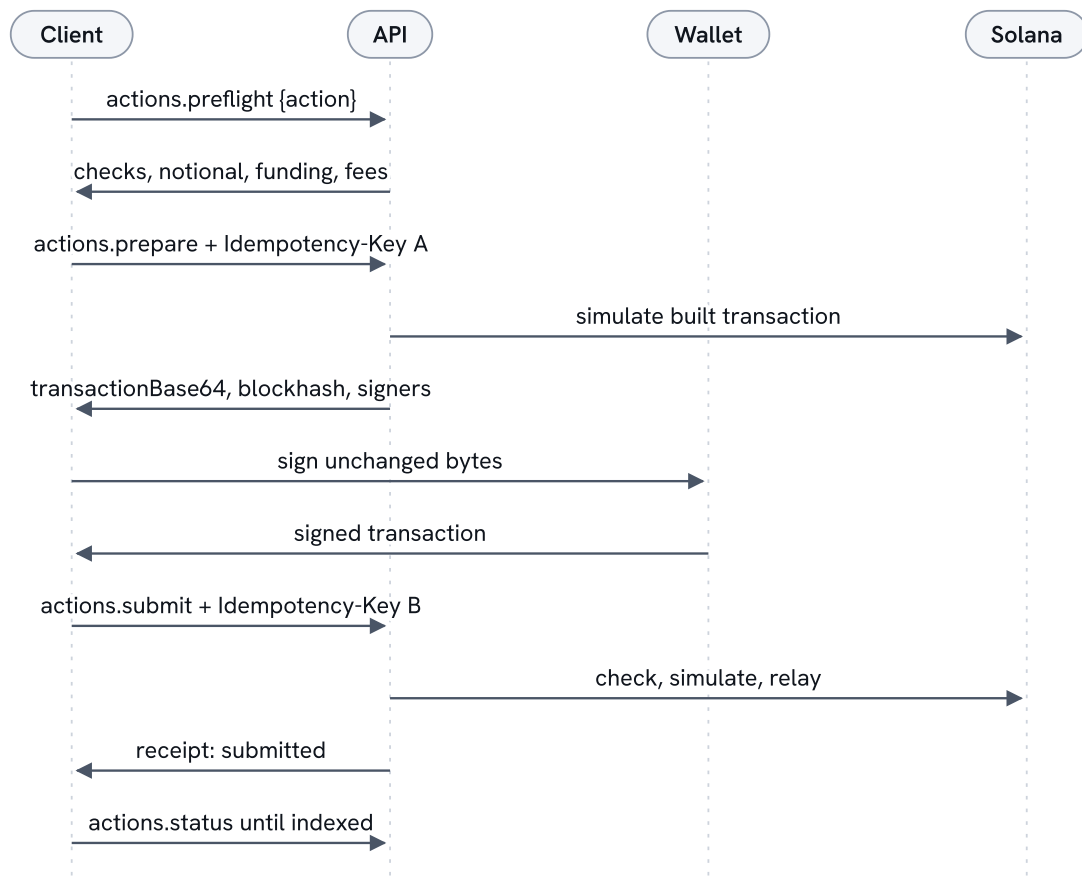


Figure 52 The four calls of the action pipeline for one bid, from preflight to an indexed confirmation.

#### 17.4.1 Idempotency keys

`actions.prepare`, `actions.submit`, `access.handoff`, `access.confirm` and `pilot.apply` require an `Idempotency-Key` of at most 128 characters. A client creates one key per intent and per step, stores it with the pending intent, and reuses it on every retry of that request; a new click is a new intent with new keys. A retry with the same key and body returns the first response with `meta.idempotentReplay: true`, even while the first attempt is still running. The same key with a different body fails with `IDEMPOTENCY_CONFLICT`. Keys are scoped to the session and the operation, and replay results are kept for one hour.

## 17.5 The TypeScript SDK

`@hourglass/contract` is the client package. It has no runtime dependencies and exports the operation table ( `endpoints` ), every request and response type, the error codes, the product parameters ( `HOURLASS_PARAMETERS` , `HOURLASS_SPEC` ), `createClient` , and the pure functions `previewAuction` , `previewSettlement` , `previewCollateral` , `previewFaultBond` , `getSeriesCalendar` and `selectDeliveryAllocation` .

`createClient({ baseUrl, fetch?, csrfToken? })` returns an object with one method, `call(operation, request, { idempotencyKey?, csrfToken?, signal? })` , typed by operation so the request and the `data` of the result are checked at compile time. It sends cookies with every request, sets the CSRF and idempotency headers, and never throws: a network failure returns an `ApiResponse` with code `NETWORK_ERROR` and `retryable: true` .

### 17.5.1 Pure previews

The previews run the protocol's arithmetic locally, with no HTTP call, in integer base units and with the programs' rounding. They are held to the same corpus of reference vectors as the Rust programs, so a preview that a client renders on every pointer move gives the same figure the program will compute. With the published collateral schedule of 20% plus 1.5 percentage points per remaining week, the example bid's notional at four weeks to delivery needs 26% collateral on the seller's side:

```
import { previewAuction, previewCollateral, previewSettlement } from '@hourglass/contract';

previewCollateral({ notionalUsdcMicros: '2956800000', weeksUntilDelivery: '4' });
// { collateralRateBps: '2600', collateralRequiredUsdcMicros: '768768000', rounding:
'up_to_usdc_base_unit' }

previewAuction({
  orders: [
    { id: 'b1', sequence: 1, side: 'bid', pricePerGpuHour: '2400000', nodeWeeks: 2 },
    { id: 'b2', sequence: 2, side: 'bid', pricePerGpuHour: '2200000', nodeWeeks: 2 },
    { id: 'a1', sequence: 3, side: 'ask', pricePerGpuHour: '2000000', nodeWeeks: 2,
collateralBudgetUsdcMicros: '2000000000' },
    { id: 'a2', sequence: 4, side: 'ask', pricePerGpuHour: '2200000', nodeWeeks: 2,
collateralBudgetUsdcMicros: '2000000000' },
  ],
  weeksMicros: '4000000',
  economics: { collateralBaseBps: 2000, collateralWeekBps: 150 },
  limits: { newNotionalHeadroomUsdcMicros: '250000000000', newQuantityHeadroom: 512 },
});
// { outcome: 'cleared', pricePerGpuHour: '2200000', volume: 4, newQuantity: 4, ... }
```

In this example book, with example prices only, the lowest tick at which all four node-weeks trade is 2.20 USDC per GPU-hour: at 2.19 only the 2.00 ask is supplied. Each ask's budget of 2,000 USDC covers its two node-weeks at that price, since  $2 \times 2,956,800,000 \times 26\%$  is

1,537,536,000 base units. `previewSettlement` for one fully delivered node-week at the same price returns a seller escrow of 2,956,800,000, a delivery fee of 73,920,000 rounded down, a net seller escrow of 2,882,880,000 and a collateral release of 768,768,000 base units.

### 17.5.2 Signing in from a script

A participant's own program signs in the same way the app does, with a keypair in place of a browser wallet. The script supplies a `fetch` that keeps the session cookie and sends one of the service's configured web origins, signs the challenge text with Ed25519, and then passes the CSRF token on writes:

```
import { createClient } from '@hourglass/contract';
import { Keypair, VersionedTransaction } from '@solana/web3.js';
import nacl from 'tweetnacl';

const baseUrl = process.env.HOURLASS_API_BASE_URL!; // the service's API base URL
const origin = process.env.HOURLASS_WEB_ORIGIN!; // one of its configured web origins
let cookie = '';
const fetchWithSession: typeof fetch = async (url, init = {}) => {
  const headers = new Headers(init.headers);
  headers.set('Origin', origin);
  if (cookie) headers.set('Cookie', cookie);
  const response = await fetch(url, { ...init, headers });
  const next = response.headers.get('set-cookie');
  if (next) cookie = next.split(';')[0]!;
  return response;
};

const key: Keypair = loadKeypair(); // the participant's own key
const wallet = key.publicKey.toBase58();
let api = createClient({ baseUrl, fetch: fetchWithSession });
const challenge = await api.call('auth.nonce', { wallet, origin });
if (!challenge.ok) throw new Error(challenge.error.code);
const signature = nacl.sign.detached(Buffer.from(challenge.data.message), key.secretKey);
const session = await api.call('auth.verify', {
  challengeId: challenge.data.challengeId, wallet,
  signatureBase64: Buffer.from(signature).toString('base64'),
});
if (!session.ok) throw new Error(session.error.code);
api = createClient({ baseUrl, fetch: fetchWithSession, csrfToken: session.data.csrfToken });

// Signing a prepared action:
const tx = VersionedTransaction.deserialize(Buffer.from(prepared.signing.transactionBase64, 'base64'));
tx.sign([key]);
const signedTransactionBase64 = Buffer.from(tx.serialize()).toString('base64');
```

The challenge expires after 300 seconds and can be used once. The session lasts for the configured session lifetime; when it ends, the script signs in again. A script never sends its secret key to the API, and the API never asks for one.

## Appendix A Parameters

This appendix lists every protocol constant and launch parameter with its value, unit and source. Two kinds of value appear. Protocol constants are compiled into the market program ( `backend/programs/hourglass/src` ) and mirrored for previews in `HOURLASS_PARAMETERS` ( `backend/packages/contract/src/spec.ts` ); changing one requires a reviewed program release, and the two copies hold the same values. Launch parameters are chosen by the service when it initializes `Config` and are stored on-chain there; the administrator can change the ones that have a configuration instruction.

Sources name the file and the identifier: `parameters.rs` and the other `.rs` files are in the market program, `spec.ts` refers to `HOURLASS_PARAMETERS`, and `config.ts` is the service configuration in `backend/src`. USDC amounts are given in USDC with the base-unit value where the code stores one.

### A.1 Contract

The deliverable unit is one node-week of the HG-H100x8 grade (\$2 The contract: HG-H100x8).

| Parameter                 | Value            | Unit              | Source                                                                  |
|---------------------------|------------------|-------------------|-------------------------------------------------------------------------|
| GPUs per node             | 8                | GPU               | <code>GPU_COUNT</code> ; <code>gpusPerNode</code>                       |
| GPU memory floor          | 80               | GB per GPU        | <code>GPU_MEMORY_GB</code>                                              |
| Host memory floor         | 1,000            | GB                | <code>RAM_GB</code>                                                     |
| Local NVMe floor          | 20               | TB                | <code>NVME_TB</code>                                                    |
| Front-end Ethernet floor  | 200              | Gbps              | <code>ETHERNET_GBPS</code>                                              |
| Node-hours per node-week  | 168              | node-hour         | <code>nodeHoursPerWeek</code>                                           |
| GPU-hours per node-week   | 1,344            | GPU-hour          | <code>GPU_HOURS_PER_NODE_WEEK</code> ; <code>gpuHoursPerNodeWeek</code> |
| Price tick                | 0.01<br>(10,000) | USDC per GPU-hour | <code>PRICE_TICK_USDC_MICROS</code> ; <code>priceTickUsdcMicros</code>  |
| USDC decimals             | 6                | decimal place     | <code>usdcDecimals</code>                                               |
| Healthy hour reachability | 55 of 60         | minutes           | <code>healthyHourReachableMinutes</code>                                |
| Basis-point scale         | 10,000           | bps               | <code>BASIS_POINTS</code>                                               |

### A.2 Calendar

Times are relative to the delivery start W, Monday 00:00 UTC. All deadlines use UTC arithmetic.

| Parameter                         | Value              | Unit                                                                                  | Source                                                    |
|-----------------------------------|--------------------|---------------------------------------------------------------------------------------|-----------------------------------------------------------|
| Delivery window                   | 168<br>(604,800 s) | hours                                                                                 | WEEK_SECONDS                                              |
| Hour / day                        | 3,600 /<br>86,400  | seconds                                                                               | SECONDS_PER_HOUR ;<br>SECONDS_PER_DAY                     |
| Listing, earliest                 | 8                  | weeks before W                                                                        | MAX_LISTING_WEEKS ;<br>listingMaximumWeeks                |
| Listing, latest                   | 2                  | weeks before W                                                                        | MIN_LISTING_WEEKS ;<br>listingMinimumWeeks                |
| First auction                     | 8, Thursday        | weeks before W                                                                        | auctionFirstWeekBeforeDelivery                            |
| Last auction                      | 1, Thursday        | weeks before W                                                                        | auctionLastWeekBeforeDelivery                             |
| Notice                            | 72                 | hours before W                                                                        | NOTICE_HOURS ;<br>noticeHoursBeforeDelivery               |
| Access-key<br>commitment closes   | 60                 | hours before W                                                                        | ACCESS_COMMIT_LEAD_HOURS<br>(delivery.rs)                 |
| Assignment deadline               | 48                 | hours before W                                                                        | ASSIGNMENT_HOURS ;<br>assignmentHoursBeforeDelivery       |
| Acceptance opens                  | 24                 | hours before W                                                                        | ACCEPTANCE_LEAD_HOURS ;<br>acceptanceHoursBeforeDelivery  |
| Access confirmation<br>by silence | 2 (7,200 s)        | hours                                                                                 | ACCESS_CONFIRM_SECONDS ;<br>buyerConfirmationSilenceHours |
| Cure window                       | 4                  | hours                                                                                 | CURE_HOURS ; cureHours                                    |
| Challenge window                  | 24                 | hours after delivery                                                                  | CHALLENGE_HOURS ;<br>challengeHours                       |
| Arbitration handoff               | 72 (259,200<br>s)  | hours after filing                                                                    | ARBITRATION_SECONDS (faults.rs);<br>arbitrationHours      |
| Inspector bond<br>handoff         | 72 (259,200<br>s)  | hours after a withdrawal request, and<br>after each signed work's challenge<br>window | INSPECTOR_BOND_HANDOFF_SECONDS<br>(inspector_bond.rs)     |
| Retirement opens                  | 192                | hours after W (delivery end plus<br>challenge window)                                 | begin (retirement.rs)                                     |
| Notice conversion<br>reserve      | 14,400             | seconds                                                                               | MIN_NOTICE_CONVERSION_SECONDS<br>(lib.rs)                 |
| Accreditation<br>validity         | 90                 | days                                                                                  | ACCREDITATION_DAYS ;<br>accreditationDays                 |

### A.3 Auction and capacity

| Parameter                          | Value                      | Unit                    | Source                                                      |
|------------------------------------|----------------------------|-------------------------|-------------------------------------------------------------|
| Book slots                         | 1,024 × 64 B<br>(65,544 B) | slot                    | MAX_BOOK_ORDERS ; BOOK_SIZE ;<br>maxOrdersPerBook           |
| Admitted live orders               | 64                         | order                   | MAX_ADMITTED_AUCTION_ORDERS ;<br>maxAdmittedAuctionOrders   |
| Admitted per direction             | 32                         | order                   | MAX_DIRECTION_ORDERS ;<br>maxAuctionOrdersPerDirection      |
| Open orders per wallet,<br>ceiling | 8                          | order                   | MAX_WALLET_OPEN_ORDERS ;<br>maxWalletOpenOrders             |
| Failed-auction grace,<br>ceiling   | 14,400                     | seconds                 | MAX_FAILED_AUCTION_GRACE_SECONDS (lib.rs)                   |
| Tickets per series                 | 512                        | node-week               | MAX_SERIES_TICKETS ; maxTicketsPerSeries                    |
| Open-interest cap per<br>series    | 250,000                    | USDC filled<br>notional | SERIES_CAP_USDC_MICROS ;<br>seriesOpenInterestCapUsdcMicros |
| Open-interest cap, all<br>series   | 1,000,000                  | USDC filled<br>notional | TOTAL_CAP_USDC_MICROS ;<br>totalOpenInterestCapUsdcMicros   |
| Account growth step                | 10,240                     | bytes                   | ACCOUNT_GROWTH_CHUNK                                        |
| Collateral term scale              | 1,000,000                  | micro-weeks per<br>week | MICRO_WEEKS                                                 |

### A.4 Collateral and fees

Collateral is  $\text{notional} \times (\text{base} + \text{per-week rate} \times \text{remaining weeks})$ , rounded up (§6 Collateral).  
Fee mechanics are in §12 Fees.

| Parameter                        | Value          | Unit                       | Source                                            |
|----------------------------------|----------------|----------------------------|---------------------------------------------------|
| Collateral base                  | 2,000<br>(20%) | bps of notional            | COLLATERAL_BASE_BPS ; collateralBaseBps           |
| Collateral per remaining<br>week | 150 (1.5%)     | bps of notional            | COLLATERAL_PER_WEEK_BPS ;<br>collateralPerWeekBps |
| Trade fee                        | 50 (0.5%)      | bps of filled notional     | TRADE_FEE_BPS ; tradeFeeBps                       |
| Delivery fee                     | 250 (2.5%)     | bps of delivered<br>escrow | DELIVERY_FEE_BPS ; deliveryFeeBps                 |
| Pledge fee                       | 50 (0.5%)      | bps of advance             | init_config (lib.rs); pledgeFeeBps                |
| Fault bond                       | 500 (5%)       | bps of ticket escrow       | FAULT_BOND_BPS (faults.rs); faultBondBps          |

## A.5 Delivery, evidence and settlement

| Parameter                     | Value                 | Unit                     | Source                                                           |
|-------------------------------|-----------------------|--------------------------|------------------------------------------------------------------|
| Healthy-delivery tolerance    | 9,900 (99%)           | bps of 168 hours         | <code>TOLERANCE_BPS</code> ; <code>toleranceBps</code>           |
| Tolerance allowance           | 1.68 (1,680,000)      | node-hours (micro-hours) | <code>math/settlement.rs</code> <code>allowance</code>           |
| Damages floor                 | 2,500 (25%)           | bps of contract price    | <code>DAMAGES_FLOOR_BPS</code> ; <code>damagesFloorBps</code>    |
| Health resolution             | 1,000,000             | micro-hours per hour     | <code>MICRO_HOUR</code>                                          |
| Delivery length               | 168,000,000           | micro-hours              | <code>DELIVERY_MICRO_HOURS</code>                                |
| Epochs per series             | 168                   | hourly commit            | <code>EPOCH_HOURS</code> ( <code>epochs.rs</code> )              |
| Ticket counters per ledger    | 512                   | ticket                   | <code>EPOCH_TICKETS</code> ( <code>epochs.rs</code> )            |
| <code>EpochLedger</code> size | 7,520                 | bytes                    | <code>EpochLedger</code> ( <code>epochs.rs</code> )              |
| Hourly health archive         | 10,800                | bytes                    | <code>HEALTH_SIZE</code> ( <code>epoch_archive.rs</code> )       |
| Delivery allocation registry  | 32,776                | bytes                    | <code>REGISTRY_SIZE</code> ( <code>delivery.rs</code> )          |
| Committee                     | 3 seats, 2 signatures | key                      | <code>Config.committee</code> ; <code>COMMITTEE_THRESHOLD</code> |

## A.6 Credit

| Parameter                        | Value                       | Unit          | Source                                                   |
|----------------------------------|-----------------------------|---------------|----------------------------------------------------------|
| Advance rate at initialization   | <code>unset ( None )</code> | bps of escrow | <code>init_config</code> ( <code>lib.rs</code> )         |
| Advance rate range               | 1 to 10,000                 | bps           | <code>configure</code> ( <code>credit_bridge.rs</code> ) |
| Lenders per obligation           | 1                           | lender        | <code>Obligation.pledge</code>                           |
| Advance cap rounding             | down                        | base unit     | <code>advance_cap</code> ( <code>credit.rs</code> )      |
| Pledge fee rounding              | up                          | base unit     | <code>pledge_fee</code> ( <code>credit.rs</code> )       |
| <code>Pledge</code> account size | 208                         | bytes         | <code>CreditPledgeView::decode</code>                    |

## A.7 Program layout

| Parameter                  | Value                                                                                                     | Unit    | Source                                 |
|----------------------------|-----------------------------------------------------------------------------------------------------------|---------|----------------------------------------|
| Account layout version     | 11                                                                                                        | version | STATE_LAYOUT_VERSION (lib.rs)          |
| Retired account parts      | 9 (book, registry, epoch ledger, health archive, funding vault, escrow vault, inventory, hook list, mint) | account | RETIRED_PARTS (retirement.rs)          |
| Health archive at creation | 10,240                                                                                                    | bytes   | HEALTH_INITIAL_SIZE (epoch_archive.rs) |
| Region codes               | NA = 0, EU = 1                                                                                            | code    | ABI.md                                 |
| Order sides                | bid 0, fresh ask 1, claim sell 2                                                                          | code    | ABI.md                                 |
| Role bits                  | 1 buyer, 2 seller, 4 trader                                                                               | bit     | ABI.md                                 |
| Role bits                  | 8 inspector, 16 lender, 32 admin                                                                          | bit     | ABI.md                                 |

## A.8 Launch parameters

The service passes these values to `init_config` or the configuration instructions when it launches; each is read from the service environment, with the value below as its default. Values stored on-chain are validated there: the wallet order limit must be 1 to 8, the grace 1 to 14,400 seconds, and the minimum notional, reference ceiling, manual timeout and inspector bond minimum must be positive.

| Parameter                  | Value                             | Set by                                                       | Source                             |
|----------------------------|-----------------------------------|--------------------------------------------------------------|------------------------------------|
| Minimum order notional     | 100 USDC (100,000,000)            | <code>init_config</code> ,<br><code>configure_auction</code> | MIN_ORDER_NOTIONAL_USDC_M ICROS    |
| Open orders per wallet     | 8                                 | <code>init_config</code> ,<br><code>configure_auction</code> | MAX_OPEN_ORDERS_PER_WALLET         |
| Failed-auction grace       | 3,600 s                           | <code>init_config</code> ,<br><code>configure_auction</code> | FAILED_AUCTION_GRACE_SECONDS       |
| Auction cutoff             | Thursday 16:00 UTC (57,600 s)     | <code>init_config</code>                                     | AUCTION_UTC_TIME                   |
| Reference spot ceiling     | 20 USDC per GPU-hour (20,000,000) | <code>init_config</code>                                     | REFERENCE_SPOT_MAX_USDC_M ICROS    |
| Manual certificate timeout | 86,400 s                          | <code>init_config</code>                                     | MANUAL_CERTIFICATE_TIMEOUT_SECONDS |
| Pledge advance cap         | 7,500 bps (75%)                   | <code>configure_credit</code>                                | PLEDGE_MAX_ADVANCE_BPS             |
| Inspector bond minimum     | 10,000 USDC (10,000,000,000)      | <code>init_config</code>                                     | INSPECTOR_BOND_USDC_BASE_UNITS     |
| Health-gating XIDs         | 48, 64, 74, 79, 95, 119, 120      | service agents                                               | HEALTH_GATING_XIDS                 |

| Parameter              | Value                                   | Set by         | Source                         |
|------------------------|-----------------------------------------|----------------|--------------------------------|
| Reference price source | committee median                        | service        | REFERENCE_PRICE_SOURCE         |
| Buyback start          | 100 USDC in the fee vault (100,000,000) | worker         | BUYBACK_MIN_USDC_MICROS        |
| Buyback batch          | up to 1,000 USDC (1,000,000,000)        | worker         | BUYBACK_MAX_USDC_MICROS        |
| Buyback slippage       | 100 bps (1%)                            | worker         | BUYBACK_SLIPPAGE_BPS           |
| Buyback interval       | 300 s                                   | worker         | BUYBACK_INTERVAL_SECONDS       |
| Buyback keeper         | the worker's key                        | set_token_mint | WORKER_KEYPAIR                 |
| Rent reclaim           | on, every 60 s                          | worker         | RENT_RECLAIM_ENABLED           |
| RPC poll interval      | 30 s (30 to 60)                         | worker         | RPC_POLL_INTERVAL_SECONDS      |
| RPC full reconcile     | 600 s (60 to 600)                       | worker         | RPC_RECONCILE_INTERVAL_SECONDS |

The health-gating XIDs are NVIDIA driver error codes. Each signed minute record from the delivery and provider agents counts the gating XIDs observed, and a minute with a nonzero count does not count toward a healthy hour (§9 Evidence and the committee). The reference price source and the XID list are service settings rather than `Config` fields; the certificate binds the reference price's source hash, and the epoch commits bind the resulting health.

## A.9 Token

| Parameter    | Value                                                                                                                                                                                                 |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ticker       | \$HOURLASS                                                                                                                                                                                            |
| Supply       | 1,000,000,000, fixed                                                                                                                                                                                  |
| Distribution | 100% to one public launch on equal terms; no presale, team, investor, airdrop or reserved allocation; no vesting                                                                                      |
| Fee use      | every fee the protocol keeps buys \$HOURLASS on the market and burns it, through <code>claim_buyback</code> , <code>buyback_swap</code> and <code>finish_buyback</code> in one transaction (§12 Fees) |
| Mint         | fixed supply: no mint or freeze authority; registered with <code>set_token_mint</code>                                                                                                                |

## Appendix B Glossary

Terms are listed alphabetically. Names in code type are the programs' own identifiers. `W` denotes the start of a series' delivery week.

| Term             | Meaning                                                                                                                                                                                                                                    |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \$HOURLASS       | The protocol's token: a fixed supply of 1,000,000,000, launched in full to the public on equal terms. Every fee the protocol keeps buys it on the market and burns it.                                                                     |
| Acceptance       | An independent inspector's test of an assigned node, recorded with <code>record_acceptance</code> from <code>W - 24 h</code> until <code>W</code> plus the cure window. A pass opens a two-hour window for the buyer to confirm or object. |
| Access hash      | SHA-256 of the buyer's SSH public key, committed with <code>set_access_hash</code> by <code>W - 60 h</code> and pinned by the seller when assigning the node.                                                                              |
| Access relay     | The service's store for the buyer's SSH public key, encrypted to the seller's X25519 key. It holds ciphertext only.                                                                                                                        |
| Accreditation    | An inspector-signed <code>NodeRecord</code> naming one node's eight GPU UUID hashes, valid for 90 days. Required through delivery end to reserve capacity or assign.                                                                       |
| Arbitration      | Resolution of a dispute still open after 72 hours, recorded by <code>resolve_arbitration</code> with two committee signatures over the external award. It may excuse whole hours inside the filed interval.                                |
| Auction          | The weekly uniform-price call auction of a series, held on Thursdays from eight weeks to one week before delivery. It clears at maximum executable volume, then the lowest tick.                                                           |
| Buyback          | The one transaction that spends the fee vault: <code>claim_buyback</code> moves USDC to program custody, <code>buyback_swap</code> buys \$HOURLASS through Jupiter and <code>finish_buyback</code> burns all of it (\$12 Fees).            |
| Buyer default    | The disposition of a ticket whose buyer committed no access hash by <code>W - 60 h</code> . The seller receives the full escrow, with no delivery fee or damages.                                                                          |
| Certificate      | The immutable <code>DeliveryCertificate</code> of one ticket, posted after the challenge window with two committee signatures. It fixes healthy time, reference spot, disposition and excused hours.                                       |
| Challenge window | The 24 hours after delivery end during which faults can still be filed. Certificates are posted only after it.                                                                                                                             |
| Claim            | A zero-decimal Token-2022 token of one series; one token is one node-week. It moves only between eligible wallets, and only before notice.                                                                                                 |
| Collateral       | USDC a seller locks for each fill: $\text{notional} \times (20\% + 1.5 \text{ percentage points per remaining week})$ , rounded up. It caps the ticket's damages.                                                                          |
| Committee        | The three keys in <code>Config</code> . Two distinct members must sign every epoch, certificate and arbitration award.                                                                                                                     |
| Compliance hold  | A record by the KYC authority, during the delivery week, that physical access is suspended because a counterparty lost eligibility. Buyer-caused seconds are stored per hour.                                                              |
| Crank            | Any caller of permissionless instructions, such as clearing, materialization, waiver and settlement. The service's worker is one.                                                                                                          |

| Term               | Meaning                                                                                                                                                                                                                                                                   |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Credit (cash)      | A durable record of USDC owed to a beneficiary whose canonical account could not receive it. Anyone may collect it to that account later.                                                                                                                                 |
| Credit (finance)   | An advance from an allowlisted lender against a seller's escrowed receivable, recorded as a pledge.                                                                                                                                                                       |
| Cure               | Replacement of a faulted node by another inspector-accepted node within four hours of the fault's filing. It stops further missed hours; earlier ones stay missed.                                                                                                        |
| Delivery Agent     | The buyer's program on the delivered node. It signs a health record every minute and answers inspector probes.                                                                                                                                                            |
| Delivery fee       | 2.5% of delivered value, paid by the seller at settlement.                                                                                                                                                                                                                |
| Disposition        | A certificate's classification: ordinary, buyer default, or buyer-caused non-access.                                                                                                                                                                                      |
| Dispute            | A bonded fault or access objection on one ticket, with an immutable alleged hour interval.                                                                                                                                                                                |
| Eligibility        | A wallet's on-chain record of roles and KYC, KYB, sanctions and allowlist flags. It is read at every physical action and claim transfer.                                                                                                                                  |
| Epoch              | One delivery hour of one series, committed with <code>commit_epoch</code> as a Merkle root and a healthy-ticket bitmap.                                                                                                                                                   |
| Escrow             | The buyer's notional, held in the series escrow vault from materialization until settlement or waiver.                                                                                                                                                                    |
| Fault bond         | 5% of the ticket's notional, rounded up, posted by a buyer who files a fault or objection.                                                                                                                                                                                |
| GPU-hour           | One H100 GPU for one hour; the unit of price. A node-week is 1,344 GPU-hours.                                                                                                                                                                                             |
| Healthy node-hour  | An hour in which all eight accredited GPUs are visible, no gating XID event occurs and the node is reachable for at least 55 minutes.                                                                                                                                     |
| Inspector bond     | USDC an inspector holds in program custody to accredit, accept and probe. A withdrawal request deactivates it at once, and it is paid back in full at the later of 72 hours after the request and 72 hours after the last challenge window its signed work is exposed to. |
| KYC authority      | The key in <code>Config</code> that writes eligibility, thaws and freezes claim accounts, and records compliance holds.                                                                                                                                                   |
| Liquidated damages | $\max(0.25 \times P, S - P) \times 8 \times H$ , capped by the ticket's collateral, where P is the contract price, S the reference spot and H the non-excused missed node-hours beyond tolerance.                                                                         |
| Lookup table       | The protocol's address lookup table, against which the service compiles version-0 transactions.                                                                                                                                                                           |
| Materialization    | Applying a cleared order to balances with <code>claim_fill</code> or <code>materialize_order</code> : escrow, collateral, claims, refunds and fees.                                                                                                                       |
| Node-week          | One HG-H100x8 node for one 168-hour delivery week, Monday 00:00 UTC to the next Monday.                                                                                                                                                                                   |
| Notice             | W - 72 h: the last instant to convert claims into delivery requests. Claims stop transferring.                                                                                                                                                                            |
| Notional           | Price per GPU-hour $\times$ 1,344 $\times$ node-weeks, in USDC base units.                                                                                                                                                                                                |
| Obligation         | A seller's short position in one series: quantity, escrow notional, locked collateral and pledge pointer.                                                                                                                                                                 |
| Open interest      | Outstanding original escrowed notional, capped by each series' reserved share (at most 250,000 USDC) and by 1,000,000 USDC in total.                                                                                                                                      |

| Term           | Meaning                                                                                                                                                                                           |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Pledge         | The credit account recording one lender's advance against one obligation. Repaid from seller escrow releases before the seller.                                                                   |
| Provider Agent | The seller's program on its management plane. It signs BMC and DCGM telemetry every minute.                                                                                                       |
| Reference spot | The market price per GPU-hour recorded in a certificate: the median of three named providers' prices, bounded by <code>Config.reference_spot_max</code> .                                         |
| Retirement     | The permissionless closing of a finished series' operator-funded accounts after every order, ticket and issuance unit has been audited. Each account's rent returns to the payer recorded for it. |
| Series         | One region and one delivery week of HG-H100x8, with its own auction book, claim mint, vaults and cap.                                                                                             |
| Settlement     | <code>settle_ticket</code> : the permissionless, one-time payout of a certified ticket's refund, damages, fees, lender repayment and seller remainder.                                            |
| Signing domain | SHA-256 of the V7 prefix, the cluster's genesis hash, the market program id and the <code>Config</code> address. It binds every committee message to one deployment.                              |
| Tick           | 0.01 USDC per GPU-hour, the price increment.                                                                                                                                                      |
| Ticket         | A <code>DeliveryTicket</code> : one requested node-week bound to a buyer, a seller's obligation, an original price and allocated collateral.                                                      |
| Tier A         | A recorded accreditation flag that the node's GPU identity was verified by NVIDIA attestation. It is recorded, not required.                                                                      |
| Tolerance      | 99% healthy delivery: an allowance of 1.68 node-hours per node-week before damages accrue. Every missed hour is still refunded.                                                                   |
| Trade fee      | 0.5% of filled notional, paid by the buyer, rounded up.                                                                                                                                           |
| Waiver         | Extinction of an unrequested unit after notice. Its notional goes to the seller, lender first, without a delivery fee, and its collateral unlocks.                                                |