Whitepaper

Anyperp

Perpetual futures on attested internet metrics, starting with artists' monthly listeners, on Solana: the mechanism, its formulas and parameters, the programs and accounts, every flow, and the services that run it.

Contents

1. Introduction

Abstract

What Anyperp is

Two questions every metric market answers

The design at a glance

Launch parameters

How this paper is organized

2. Markets on an attested metric

The metric

The index

A perpetual on the level

One market per artist

Relative value across markets

Rounds and prints

Market states

3. The cap: open interest below the cost of moving the number

The attack the cap prices

Break-even open interest

The short side

A quarter of break-even

Caps per side

Where the costs come from

A worked example

Caps and a moving level

Probation caps

Enforcement at fill time

4. The oracle: two-root rounds

Accounts
The round clock
Two attestation roots
Attestor sets and timelocked rotation
Observations
finalizeround
The circuit band and side pauses
Disputed and failed rounds
Guardian pauses, halts and settlement
The price history and what the perps program reads
Instruction reference
5. Trading: next-print execution
Placing an order with openorder
The pending order and the next print
The order queue and its barrier
Leverage and margin requirements
The fill
Rejection and refundable escrow
Adding to an open position
Closing at the next print
Moving margin with modifymargin
Cancelling with cancelorder
Equity, maintenance and liquidation
6. Funding and the borrow fee
Skew and the skew scale
Velocity funding
The cumulative funding index and position accrual
The borrow fee
Rounding
How funding and borrow reach the vault

7. The vault

One vault behind every market

The balance sheet

Shares and NAV per share

NAV generations

Deposits

Withdrawals

Distressed NAV

Fees and the fee split

The insurance fund

Queue limits

8. Halts, settlement and claims

Failed rounds and the halt threshold

The guardian's halt

Settlement at the 24-hour TWAP

Settling the market and each position

Protected claim balances

Claiming with claimbalances

9. Listing a market

The proposal

The bond: proposemarket

Shadow mode

Curator approval

The 48-hour timelock and activation

Probation

Promotion to live

Bond return

Slashing

The proposer's fee share

The launch cohort

Listing states and transitions

10. Programs and accounts

The two programs

How the perps program reads oracle state

Oracle accounts

Perps accounts

Account reference

The address lookup table

Oracle instructions

Perps instructions

11. Transaction flows

The common path

Sign in

Open an order and its fill

Add to a position

Modify margin

Close

Liquidation

Deposit and its fill at NAV

Withdrawal request and execution

Claiming balances

Proposing a market

A round from open to print

12. Security design

The authority model

Timelocked attestor rotation

Governed program upgrades

Permissionless work never depends on a user's token account

Rounding that favours the vault

The order queue barrier

Anti-option devices
Server-built transactions checked before signing
Sessions bound to a signed wallet message
Geo access
13. Off-chain infrastructure
13.1 Services and what each does
13.2 Coordination
13.3 Data model
13.4 Public API
13.5 Server-built transactions and the wallet
13.6 Sessions and eligibility
13.7 Hosting layout
14. Parameters
How parameters are set and changed
Rounds and oracle
Caps and listing
Trading and margin
Funding and borrow
Vault and withdrawals
Fees
Operations
Token
15. The ANYPERP token
Supply and launch
The fee flow
Where the token sits
16. Glossary and notation
Glossary
Notation

1. Introduction

Abstract

Anyperp is a perpetual futures service on Solana on artists' monthly listeners: the count of unique listeners that a streaming platform shows on each artist's public profile page. Each artist is one market. A trader goes long or short the artist's index with USDC margin at up to 3× leverage, and one USDC vault takes the other side of every market. Two independent attestation roots observe the count in every 15-minute round, and a round prints only when both roots' signed observations agree within 0.5 %; the print is their median. An order fills at the first print whose observations were all fetched after the order landed. Open interest on each side of a market is capped at a quarter of the open interest at which buying a move in the count breaks even, so an attack on the number loses money at every push size. Markets enter through a bonded proposal, 14 days of shadow rounds, approval by three of five curators, a 48-hour timelock and 30 days of probation at a quarter of their caps. This paper sets out the mechanism, the two on-chain programs that enforce it and the services that operate it.

What Anyperp is

Anyperp turns a public audience number into a continuous market. The number is monthly listeners, an integer the platform publishes on each artist's profile page and updates as listening changes. Anyperp reads it, attests it through two independent roots and prints it as an index every 15 minutes. A perpetual on that index lets a trader hold a view on an artist's audience for as long as the view lasts: positions have no expiry, so nothing has to be rolled and no single snapshot decides the payout. A long gains when the audience grows, a short gains when it shrinks, and both pay or receive funding while they stay open.

Traders open isolated positions with USDC margin. Liquidity providers (LPs) deposit USDC into the vault and hold its share token, a receipt on the vault's net asset value (NAV). Proposers post a USDC bond to bring a new artist to market and earn a share of its trading fees. Curators, five members of whom three must agree, approve listings and their caps. Reporters post the roots' signed observations, and keepers finalize rounds, fill orders and liquidate; both roles are permissionless, because the programs check what is submitted, not who submits it. A guardian can pause a market for a bounded time and halt it once rounds have stopped printing, and it cannot write a price or move funds.

Two Anchor programs hold all state. `anyperp_oracle` runs the rounds, verifies the roots' signatures and keeps each metric's price history. `anyperp_perps` holds the vault, the markets, the orders and the positions, and takes its prices only from finalized oracle rounds. Sentinel, an off-chain service, drafts market specs and the cost dossiers behind each cap and flags anomalies for curators. It has no on-chain authority.

Two questions every metric market answers

A market on a web metric has to settle two matters before it is safe to trade: who attests the number, and what stops someone from moving it.

Anyperp answers the first with two attestation roots that do not depend on each other. Root 1 is a set of Reclaim attestors identified by secp256k1 keys. Root 2 is an independent notary with an Ed25519 key. Both key sets are allowlisted on chain, and the oracle program verifies each signature through Solana's signature precompiles in the same transaction that submits the observation. A round needs one valid observation from each root, and the two values must agree within 0.5 % before the round prints. No account, the admin included, can write a price.

Anyperp answers the second with arithmetic. Buying a move in the count costs money, and the cost grows with the size of the move and with the artist's level. A position profits from the same move in proportion to its notional. Below a certain open interest, the cost exceeds the profit whatever the size of the move. Anyperp caps each side's open interest at a quarter of that break-even point, so an attacker holding the largest position the market allows still loses at least three quarters of what the push costs. Chapter 3 derives the cap in full.

The design at a glance

Two-root oracle

A round opens every 15 minutes. Each root signs an observation whose source and fetch times fall inside the round, bound to that metric, that round and that root. `finalize_round` requires both observations, checks that they agree within 50 bps, takes their median, normalizes it to a fixed9 index and checks it against a ±10 % circuit band around the close 24 hours earlier. A round that fails a check prints nothing; the next round starts clean.

Capped open interest

Each market has a long cap and a short cap in USDC, set at listing from its cost dossier: $cap_{long} = \kappa\, c_{up} L$ and $cap_{short} = \kappa\, c_{down} L$, with $\kappa = 0.25$. Pushing a count down costs more than pushing it up, so the short cap is the larger of the two. Every opening fill is checked against its side's cap, counting all accounts on that side at the index of the print where it fills.

Next-print execution

An order is a request. It escrows margin, names an acceptable price and waits. It fills at the first print whose observations from both roots were all fetched after the order landed. An order placed after the source changes fills at a print that already contains the change, so a stale index never sets a fill price.

One USDC vault

One vault takes the net skew of every market. Fill prices carry a skew premium, $I_t\left(1+\frac{s_{before}+s_{after}}{2S}\right)$, funding drifts toward the side that reduces the vault's exposure, and a borrow fee rises as a side fills toward its cap. Deposits are priced at the next complete NAV and withdrawals wait in a 72-hour queue, so LPs cannot enter or leave ahead of a print they can already see.

Curated, bonded listing

Anyone can propose a market by posting a bond of 5,000 to 25,000 USDC with a market spec and a cost dossier. The oracle runs the market in shadow mode for 14 days, and the roots must agree on at least 99 % of its rounds. Three of five curators approve its caps at or below the dossier's target, a 48-hour timelock follows, and the market trades for 30 days of probation at 25 % of the approved caps. A market that completes probation goes live at its full caps and the bond returns to the proposer. A market that fails during probation halts, and its bond goes to the insurance fund.

The ANYPERP token

ANYPERP has a fixed supply of 1,000,000,000, all of it released in one public launch on equal terms. The protocol's 20 % share of every trading, liquidation and borrow fee buys ANYPERP on the market and burns it, so the supply falls as the markets are used.

Launch parameters

The service runs with the launch parameters below, set in protocol configuration. The programs enforce production bounds on several of them and reject a configuration outside those bounds. Chapter 14 lists every parameter.

Parameter	Launch value
Round cadence	900 s (15 minutes)
Attestation roots per round	2
Root tolerance	50 bps (0.5 %)
Circuit band	1,000 bps (10 %) against the close at or before 24 h earlier
Circuit confirmations	2 agreeing rounds
Side pause	21,600 s (6 h)
Rounds without a print before a halt	4
Index divisor	1,000,000 listeners per index point

Parameter	Launch value
Maximum leverage	30,000 bps (3×)
Maintenance margin	2,000 bps of notional
Cap fraction κ	0.25 (`kappa_fixed9` = 250,000,000)
Trading fee	10 bps on opening and closing fills
Liquidation fee	100 bps
Fee split	60 % vault, 20 % protocol, 10 % insurance fund, 10 % proposer
ANYPERP	1,000,000,000, fixed; the protocol's share buys and burns it
LP withdrawal queue	259,200 s (72 h)
Proposal bond	5,000 to 25,000 USDC
Shadow mode	1,209,600 s (14 days), roots agreeing on at least 9,900 bps of rounds
Curator approval	3 of 5, then a 172,800 s (48 h) timelock
Probation	2,592,000 s (30 days) at 2,500 bps (25 %) of approved caps
Attestor rotation timelock	172,800 s (48 h)
Price history	168 hourly closes
Settlement price	TWAP of the 86,400 s (24 h) before the halt
Markets at launch	20 artists

How this paper is organized

Chapter 2 defines the metric, the index and the instrument: what a position is, how it is sized and valued, how markets relate to artists and to each other, and the states a market passes through. Chapter 3 derives the open-interest cap from the economics of an attack on the count and follows a worked example through to the checks that enforce it at fill time. Chapter 4 describes the oracle: the round, the two roots, what each root signs, the on-chain verification of every observation, the conditions under which a round prints, and the history and settlement prices the perps program reads.

Chapter 5 covers trading: next-print execution on the round clock, fill pricing, margin and liquidation. Chapter 6 covers funding and the borrow fee, chapter 7 the vault, its NAV, the withdrawal queue and the fee split, chapter 8 halts, settlement and claim balances, and chapter 9 the listing pipeline from proposal to live market. The chapters that follow describe the programs' accounts and the authority model, chapter 13 the off-chain infrastructure that keeps the system running, chapter 14 every

parameter with its launch value, and chapter 15 the ANYPERP token: its supply, its launch and the buyback the protocol's fee share funds.

2. Markets on an attested metric

The metric

Each market follows one artist's monthly listeners: the integer shown on the artist's public profile page on a streaming platform, read logged out. The count is a rolling tally of unique listeners, so it moves gradually as an audience grows or fades instead of jumping with each play. The platform removes artificial streams from the public count before it publishes it.

A market's `MetricSpec` account in `anyperp_oracle` fixes what is read and how. It holds a 32-byte `metric_id`, the SHA-256 of the metric's provider name and the artist's 22-character profile ID; a `spec_hash` that commits to the source, the extraction rule and the normalization; and the oracle parameters of the market: `index_divisor`, `cadence_seconds`, `tolerance_bps`, `band_bps`, `side_pause_seconds`, `confirmation_rounds` and `failed_rounds_before_halt`. No instruction changes a spec once it exists.

L denotes the level at the last print: the artist's monthly listeners, as an integer.

The index

The oracle prints a fixed9 index, an unsigned integer with 9 implied decimals. Each market sets an index divisor D in its spec. When the two roots observe raw counts v_1 and v_2 in round t, the round prints

$$m_t = \min(v_1, v_2) + \left\lfloor \frac{|v_1 - v_2|}{2} \right\rfloor, \qquad I_t = \left\lfloor \frac{10^9 \cdot m_t}{D} \right\rfloor$$

where m_t is the floored median of the two observations and I_t is the index in fixed9 units. One index point is D listeners. Launch markets use D = 1,000,000, one index point per million listeners.

For an artist at 60,000,000 listeners, the index reads 60.000000000 and is stored as 60,000,000,000. A move of 600,000 listeners is a move of 0.6 index points. The divisor sets the scale of the quote and nothing else: notional, PnL and caps are all computed from size × index, and size is quoted in the same index units.

A perpetual on the level

A position has a side, a size q in index units (fixed9), an entry price, isolated USDC margin and checkpoints into the market's cumulative funding and borrow indices. Its notional at a print is

$$N = q \cdot I_t$$

which the programs compute in USDC base units as $\lceil q \cdot I_t / 10^{12} \rceil$, both factors in fixed9. USDC uses 6 decimals, so 1 USDC is 1,000,000 base units. A position's PnL against a mark I_{mark} is

$$\mathrm{PnL}_{long} = q\,(I_{mark} - I_{entry}), \qquad \mathrm{PnL}_{short} = q\,(I_{entry} - I_{mark})$$

rounded down toward the vault. A size of one index unit gains or loses 1 USDC per index point. In the example above, a long of 5,000 index units has a notional of $300,000 at 60 points and gains $3,000 for every rise of 600,000 listeners.

Margin is isolated per position. An opening fill requires margin of at least N/λ after fees, with λ at most 3 (`max_leverage_bps` = 30,000). A position's equity is its margin plus PnL, minus accrued funding and borrow charges; a keeper can liquidate it at any print where equity falls below the market's maintenance margin, 2,000 bps of notional at launch.

Fills happen at the index of a print plus a skew premium. With $s = N_{long} - N_{short}$ the market's skew in USDC and S its skew scale, an order that moves the skew from s_{before} to s_{after} fills at

$$P = I_t \left(1 + \frac{s_{before} + s_{after}}{2S}\right)$$

with buys rounded up and sells rounded down. Funding has no spot price to anchor to, so it works as a balancing price: its rate f drifts at a velocity $\dot{f}$ in proportion to s/S, and the side that adds to the vault's net exposure pays the side that reduces it. A borrow fee on open interest rises with the market's utilization of its caps. Chapters 5 and 6 give the full pricing, margin and funding formulas.

A position has no expiry. It stays open until a reduce-only order closes it, a keeper liquidates it, or the market settles. Each owner holds at most one position per market, on one side; an order on the same side adds to it at the new print, and an order on the other side has to wait until the position is closed.

One market per artist

A market account is a PDA seeded by its metric (`["market", metric]`), so each `MetricSpec` has exactly one market, and each artist's monthly listeners is one metric. A position is a PDA per owner and market. The vault serves up to 24 markets; the service launches with 20 artists.

A market's caps are proportional to the reference level in its cost dossier (chapter 3). Every market keeps its own skew, funding rate, borrow index and caps, and all of them settle profit and loss against the same vault.

Relative value across markets

A view on one artist against another is two positions: long artist A and short artist B. With notional N in each leg at entry, the legs hold $q_A = N/I_A$ and $q_B = N/I_B$ index units. When the levels change by

fractions r_A and r_B, the pair returns

$$\mathrm{PnL} = q_A\, r_A I_A - q_B\, r_B I_B = N\,(r_A - r_B)$$

measured from the index at entry, before fees, funding and the skew premium. A rise in listening that lifts both artists alike cancels out; what remains is the difference between the two audiences. Each leg is margined and capped in its own market, and each fills at the next print of its own market.

Rounds and prints

A round is one 15-minute observation window of one metric. `open_round` opens it once `cadence_seconds` (900) have passed since the previous round opened, so windows never overlap and a metric runs 96 rounds a day. Within the window, reporters post one signed observation from each root; after both are in, a keeper calls `finalize_round`. A round that finalizes is a print. Its index, its finalization time and the earliest fetch time of its two observations are what every later step reads.

Prints drive the rest of the system. Pending orders fill at prints. Funding and borrow accrue over the time between consecutive prints. Liquidations, position valuations and NAV generations all reference a print. The oracle keeps the last valid print of each hour in a 168-entry history ring, which supplies the 24-hour reference for the circuit band and the settlement TWAP. When the source has not changed since the previous round, the new print repeats the level; the round still serves as the clock for fills, funding and NAV.

A round that does not print ends in one of three terminal states. `DISPUTED` means both roots reported but the observations did not agree or were out of order. `FAILED` means a root was missing when the window closed. `OUT_OF_BAND` means the roots agreed on a level beyond the circuit band, which waits for confirmation by later rounds. Chapter 4 describes each in full.

Market states

A market moves through a listing pipeline before it trades and through a halt and a settlement if its source stops printing. Two programs record the state: the `Proposal` and `Market` accounts in `anyperp_perps`, and the pause and halt fields of the `MetricSpec` in `anyperp_oracle`. Every opening order checks both at the moment it lands and again at the print where it fills.

State	Recorded as	Opening orders	Closing orders	Entered through	Left through
Proposed	`Proposal` with bond, cost inputs and target caps; `MetricSpec`, `PriceHistory` and `Market` created	No	No	`propose_market`	curator votes during shadow mode

State	Recorded as	Opening orders	Closing orders	Entered through	Left through
Shadow	`Market.status` = `SHADOW`; rounds run and `evaluated_rounds`, `agreed_rounds` accumulate	No	No	starts in the proposing transaction	three `approve_market` votes, or three `reject_market` votes (`REJECTED`, bond returned)
Approved	`Proposal.status` = `TIMELOCK`	No	No	third approval, after 14 days with at least 99 % agreement	`activate_market` after 48 h
Probation	`PROBATION`; caps at 25 % of the approved caps	Yes	Yes	`activate_market`	`promote_market` after 30 days with at least 99 % agreement, or `slash_bond`
Live	`LIVE`; full approved caps	Yes	Yes	`promote_market`	halt
Side paused	`pause_long_until` or `pause_short_until` in the future	Not on the paused side	Yes	an out-of-band round pauses one side; a disputed or failed round pauses both, for 6 h	the next print, or expiry
Guardian paused	`guardian_pause_until` in the future	No	Yes	`guardian_pause_metric`, at most 24 h per episode	expiry; curators extend or clear it
Halted	`MetricSpec.halted`; `Market.status` = `HALTED`	No	No	`halt_metric` after `failed_rounds_before_halt` consecutive rounds without a print, or `curator_halt_metric` from `slash_bond`; `halt_market` mirrors it	`settle_metric`, then `settle_market`
Settled	`SETTLED`; market index set to the settlement TWAP	No	Positions close through `settle_position`	`settle_market`	final

The shadow gate counts rounds, not days. Approval needs at least `shadow_seconds / cadence_seconds` evaluated rounds, 1,344 for 14 days of 15-minute rounds, and agreed rounds at or above 9,900 bps of them. Promotion applies the same rule to the rounds evaluated during probation, 2,880 over 30 days. A round counts as agreed when both roots reported, their values matched within tolerance and their times fell inside the window; a round with a missing root or a disagreement counts against the market.

Side pauses and guardian pauses stop new exposure and nothing else. Reporters keep reporting, keepers keep finalizing, closing orders fill, and a pending opening order on a paused side is rejected when a keeper reaches it at a print; `cancel_order` then returns its margin to the owner's claim balance. A halt stops the rounds themselves: `open_round` and the observation instructions refuse a halted metric, and the market waits for its settlement price. Pending orders in a halted market can be expired at once, and their margin returns to the owner's claim balance.

A proposal that curators reject returns its bond to the proposer, as does a market that completes probation. A market slashed during probation halts and settles like any other halted market, and its bond goes to the insurance fund.

3. The cap: open interest below the cost of moving the number

A listener count can be moved. Fake listeners can be bought, and real listening can be discouraged. A perpetual on the count pays whoever moves it, in proportion to the position they hold. Anyperp sizes each side of each market so that the payment never covers the cost. This chapter derives the cap from the attacker's profit, sets it per side, follows a worked example through the fixed-point arithmetic the programs run, and shows where the cap is enforced.

The attack the cap prices

An attacker opens a long position with notional N_{long} while the level is L and the index I_t. Next-print execution decides the order of events: the order fills at the first print whose observations were all fetched after it landed, so the position has to exist before the pushed number is observed. The attacker then buys a move of fraction x, raising the count from L to $L(1+x)$, and holds it until both roots observe it and it prints. The move costs

$$C(x) = c_{up}\, x\, L$$

where c_{up} is the net cost in USDC of adding one listener to the count and keeping it there through the print. The position holds $q = N_{long}/I_t$ index units, and the index rises by xI_t, so the position gains $q \cdot xI_t = N_{long}\, x$. The attack's profit is

$$\Pi_{up}(x) = N_{long}\, x - c_{up}\, L\, x = x\ (N_{long} - c_{up}L)$$

Trading fees are left out. The attacker pays them on the opening fill and again on the closing fill, so they only lower Π_{up}.

Break-even open interest

The profit is linear in x with slope $N_{long} - c_{up}L$, and its sign does not depend on x. If the slope is negative, every push loses money and a larger push loses more. If it is positive, every push makes money and a larger push makes more. There is no optimal push size to defend against, only a threshold on open interest:

$$N^*_{long} = c_{up}\, L$$

This is the break-even open interest of the long side. Below it no push of any size pays; above it every push does. The threshold is a property of the market, which is why Anyperp caps open interest rather than limiting how far the index may move.

Two features of the attacker's position leave the threshold unchanged. Leverage changes the margin the attacker posts, at least $N_{long}/3$, but not the notional that earns the move. Splitting the position across many accounts changes nothing either, because the cap counts every position on the side, whoever holds it.

The short side

An attacker who holds short notional N_{short} and pushes the count down by a fraction x pays c_{down} per listener removed:

$$\Pi_{down}(x) = N_{short}\, x - c_{down}\, L\, x = x\, \left(N_{short} - c_{down} L\right), \qquad N^*_{short} = c_{down}\, L$$

Removing listeners is harder than adding them. A fake listener can be bought, while lowering the count means keeping real people from listening to an artist they chose to hear. So c_{down} is larger than c_{up}, and the short side of a market carries more open interest than the long side.

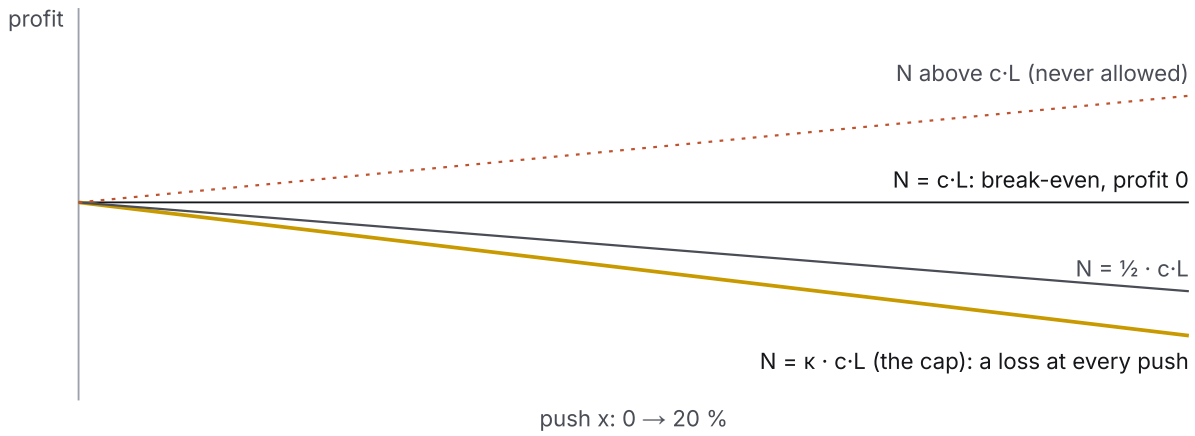
A quarter of break-even

Anyperp sets each side's cap at a fraction κ of break-even, with $\kappa = 0.25$, stored as `kappa_fixed9` = 250,000,000. With $N_{long} \le \kappa\, c_{up} L$, the profit of any upward push is bounded by

$$\Pi_{up}(x) \le x\, (\kappa - 1)\, c_{up} L = -(1-\kappa)\, C(x) = -0.75\, C(x)$$

The attack loses money at every push size $x > 0$, and the loss is at least three quarters of what the push costs. The same bound holds for the short side with c_{down} in place of c_{up}.

The fraction is also the margin for the cost figure. The cap sits at or below break-even whenever the true cost per listener is at least κ times the dossier's figure, so the cap holds even if moving the count costs only a quarter of the dossier's price.



Attack profit against push size x: below the cap the line falls with slope (κ − 1)·c_up·L, at break-even it lies flat at zero.

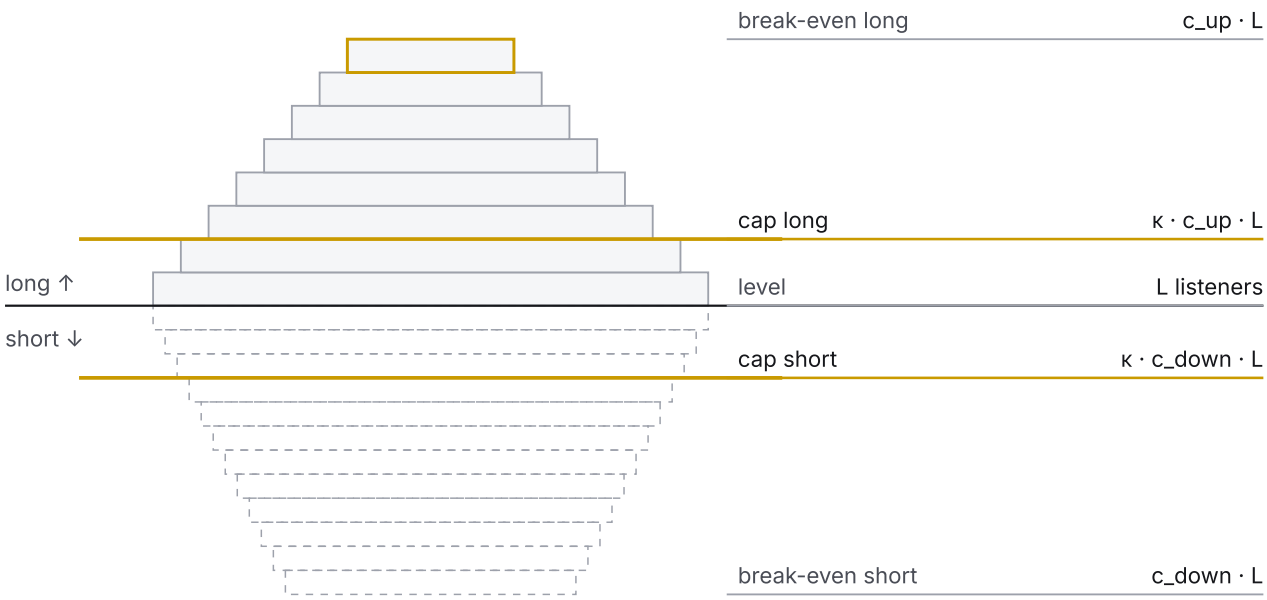
Caps per side

Each market carries two caps in USDC:

$$cap_{long} = \kappa\, c_{up}\, L, \qquad cap_{short} = \kappa\, c_{down}\, L$$

Caps are global per side. N_{long} is the notional of every long position in the market at the current index, and N_{short} the same for shorts. Every opening fill on a side is checked against that side's cap. The listing evaluates both caps at the reference level of the market's cost dossier and stores them in USDC, as the following sections show.

The level appears in both caps, so a market listed at a larger audience carries proportionally more open interest. The ratio between the two caps is the ratio of the two costs, $cap_{short}/cap_{long} = c_{down}/c_{up}$, and it does not depend on the level.



A market as a terraced hill: its height is the break-even open interest c_up·L, the cap is the κ contour at a quarter of that height, and the basin below is three times deeper for shorts.

Where the costs come from

A proposal carries three inputs to the cap, stored in its `Proposal` account: `cost_up_usdc_per_unit` (c_{up}, in USDC base units per listener), `cost_down_usdc_per_unit` (c_{down}) and `reference_level` (L_0, in listeners). A `dossier_hash` commits to the cost dossier that supports them. Sentinel drafts the dossier with cited evidence, the proposer posts it with the bond, and curators review it before they vote.

The dossier prices c_{up} as the net cost of one fake unique listener that survives the platform's filtering and stays in the count through the print. The platform removes artificial streams from the public count, so a bought listener only counts if it is indistinguishable from a real one. The dossier uses the cost to the subject, the artist or the label, because the subject gains visibility from a higher count and so pays the lowest net cost of anyone. c_{down} is the cost of removing one real listener from the count.

The program turns the inputs into caps itself. `propose_market` requires the proposal's target caps to equal the arithmetic exactly:

$$cap = \left\lfloor \frac{1}{10^4} \left\lfloor \frac{\kappa_9 \cdot c \cdot L_0}{10^9} \right\rfloor \cdot a \right\rfloor$$

where κ_9 is `kappa_fixed9`, c is the cost in USDC base units per listener, L_0 is the reference level and a is the active multiplier in bps: 10,000 for the target caps. The target is rounded down before the

multiplier is applied. The on-chain Rust function and the TypeScript function the services use are checked against the same test vectors.

Curators approve with `approve_market(cap_long_usdc, cap_short_usdc, configuration_hash)`. Each cap must be positive and at or below its target. The first approval fixes the caps, and the second and third must name the same caps and the same `configuration_hash`, a hash that binds the exchange policy, the full proposal, the market and metric addresses and both caps. Curators can approve smaller caps than the dossier's target, never larger ones.

A worked example

Take an artist at 60,000,000 listeners. The example uses c_{up} = $0.02 per listener and $c_{down} = 3 \times c_{up}$ = $0.06 per listener, with the launch value κ = 0.25.

Break-even open interest is

$$N^*_{long} = 0.02 \times 60{,}000{,}000 = 1{,}200{,}000 \text{ USDC}, \qquad N^*_{short} = 0.06 \times 60{,}000{,}000 = 3{,}600{,}000 \text{ USDC}$$

and the caps are a quarter of each:

$$cap_{long} = 0.25 \times 1{,}200{,}000 = 300{,}000 \text{ USDC}, \qquad cap_{short} = 0.25 \times 3{,}600{,}000 = 900{,}000 \text{ USDC}$$

In the program's units, c_{up} is 20,000 base units and c_{down} 60,000 base units. The long target is $\lfloor 250{,}000{,}000 \times 20{,}000 \times 60{,}000{,}000/10^9 \rfloor$ = 300,000,000,000 base units, which is 300,000 USDC, and the short target is 900,000,000,000 base units, 900,000 USDC.

The table carries the long side through four push sizes, for an attacker holding the whole long cap and for one holding the break-even amount.

Push x	Listeners added	Cost $c_{up}xL$	Gain at the cap	Net at the cap	Net at break-even
1 %	600,000	$12,000	$3,000	−$9,000	$0
5 %	3,000,000	$60,000	$15,000	−$45,000	$0
10 %	6,000,000	$120,000	$30,000	−$90,000	$0
25 %	15,000,000	$300,000	$75,000	−$225,000	$0

At every size the attacker at the cap loses three quarters of the cost. The short side follows the same pattern at three times the scale.

Push x	Listeners removed	Cost $c_{down}xL$	Gain at the cap	Net at the cap
1 %	600,000	$36,000	$9,000	−$27,000
10 %	6,000,000	$360,000	$90,000	−$270,000

In index terms, with the launch index divisor of 1,000,000 the index stands at 60 points. A long at the cap holds 300,000 ÷ 60 = 5,000 index units. A 10 % push lifts the index to 66 points and the position gains 5,000 × 6 = $30,000, the figure in the table.

The circuit band adds a time cost on top. A round whose index lies more than 10 % from the close at or before 24 hours earlier does not print until later rounds confirm the new level, and new positions on the side the move favors are paused meanwhile. A push larger than the band has to be held across the confirming rounds before it prints, and that side takes no new positions until it does.

The launch markets are proposed with lower cost figures than the example: c_{up} = 1,000 base units (0.001 USDC per listener) and c_{down} = 3,000 base units (0.003 USDC), at a reference level of 10,000,000 listeners. Each launch market's target caps are therefore 2,500 USDC long and 7,500 USDC short, and 625 USDC and 1,875 USDC during probation.

Caps and a moving level

The caps are fixed in USDC at listing, from the reference level L_0 in the dossier. Open interest is revalued at every print: the programs compute each side's notional as its total size times the current index. As the level moves to L_t, the cap's share of break-even becomes

$$\frac{cap_{long}}{c_{up}\,L_t} = \kappa\,\frac{L_0}{L_t}$$

A rising level lowers the share below κ. The same cap then covers fewer index units, cap_{long}/I_t, and the market holds a smaller fraction of what a push would cost. A falling level raises the share, and the cap stays below break-even at every level above κL_0. In the example, that is every level above 15,000,000 listeners.

Revaluation also means a side can reach its cap without any new order. When the index rises, the long side's notional rises with it; once it reaches cap_{long}, opening longs are rejected while closes and shorts continue. The borrow fee uses the same revalued open interest. Its rate is

$$r_{borrow} = r_{base} + r_{slope} \cdot \max\left(\frac{N_{long}}{cap_{long}}, \frac{N_{short}}{cap_{short}}\right)$$

with utilization measured in bps, so the cost of holding a position rises as either side nears its cap.

Caps change in two steps of the listing lifecycle, and only there. `activate_market` sets them to 25 % of the approved caps, and `promote_market` sets them to the approved caps in full.

Probation caps

A market trades its first 30 days in probation. `activate_market` applies the active multiplier `probation_multiplier_bps`, which the exchange configuration requires to be exactly 2,500:

$$cap_{side}^{prob} = \left\lfloor \frac{2{,}500 \cdot cap_{side}}{10^4} \right\rfloor$$

During probation a market carries at most $\kappa/4$ of break-even, 6.25 %. In the example the probation caps are 75,000 USDC long and 225,000 USDC short.

Probation tests the source while little is at stake. `promote_market` succeeds only after `probation_ends_at`, with the metric not halted, the market solvent, at least 2,880 rounds evaluated over the 30 days, and the roots agreeing on at least 99 % of them. It then sets the full approved caps and marks the market `LIVE`, and `return_bond` credits the bond to the proposer's claim balance. If the metric halts, the market becomes insolvent or probation ends without the required agreement, `slash_bond` sends the bond to the insurance fund and halts the market.

Enforcement at fill time

The cap is checked when an order fills, not when it lands, because both the open interest and the index it is measured at belong to the print. `execute_order` first runs `sync`, which moves the market to the print's index and accrues funding and borrow. For an opening order on a side with total size q_{side}, the program then requires

$$\left\lceil \frac{(q_{side} + q_{order}) \cdot I_t}{10^{12}} \right\rceil < cap_{side}$$

with both sizes and the index in fixed9 and the result in USDC base units. The inequality is strict.

The check runs twice. `preflight_order` evaluates it along with the market state, the side pauses, the acceptable price and the vault's solvency. If any of these fails, the order is marked rejected at that print: it never fills in part, the queue moves on to the next order, and `cancel_order` returns its margin to the owner's claim balance. If preflight passes, the program checks the cap again before it writes the new position. Closing orders skip the check, since a close only lowers the side's open interest.

Orders fill in the order they landed, one market queue at a time, so headroom under the cap goes to the earliest order. The app's quote shows each side's headroom, the cap minus the current open interest, and the largest order size that fits under it.

4. The oracle: two-root rounds

`anyperp_oracle` turns two independent observations of a public count into a price the perps program can trade on. It holds the configuration of the two attestation roots, one spec and one price history per metric, and one account per round. Every check in this chapter runs inside the program. The services that open rounds, report observations and finalize them pay for transactions and hold no authority over what prints.

Accounts

Account	Seeds	Holds
`OracleConfig`	`["oracle_config"]`	admin, `curator_authority` (the perps curator PDA), guardian, attestor rotation timelock, guardian pause limit, history bucket, reference and settlement windows
`AttestorSet`	`["attestors", root]`	up to 8 keys of one root and their validity window, the previous epoch's keys and window, a pending rotation, a generation counter
`MetricSpec`	`["metric", metric_id]`	the immutable spec, the round sequence, the last print, side and guardian pauses, circuit-band confirmation state, agreement counters, halt and settlement
`OracleRound`	`["round", metric, sequence]`	the round's window and status, both roots' observations with signer and signature, the final index, the earliest fetch time of this print and of the previous one
`PriceHistory`	`["history", metric]`	a ring of 168 hourly closes of 16 bytes each, a timestamp and an index

The round clock

`open_round(sequence)` opens the next round of a metric. Anyone can call it and pays for the new account. It requires a metric that is not halted, a `sequence` equal to the metric's `next_sequence`, and at least `cadence_seconds` since the previous round opened. The round's window is half-open, from `opened_at`, the time of the call, to `closes_at` = `opened_at` + `cadence_seconds`. In production the cadence is 900 seconds; the perps listing policy accepts no other value. Sequences start at 1, each round is its own PDA, and windows never overlap.

A round starts `OPEN` and becomes `AWAITING_ROOTS` with its first observation. It ends in one of four terminal states: `FINALIZED` (a print), `DISPUTED`, `OUT_OF_BAND` or `FAILED`. Only a `FINALIZED` round carries an index, and no round leaves a terminal state.

Two attestation roots

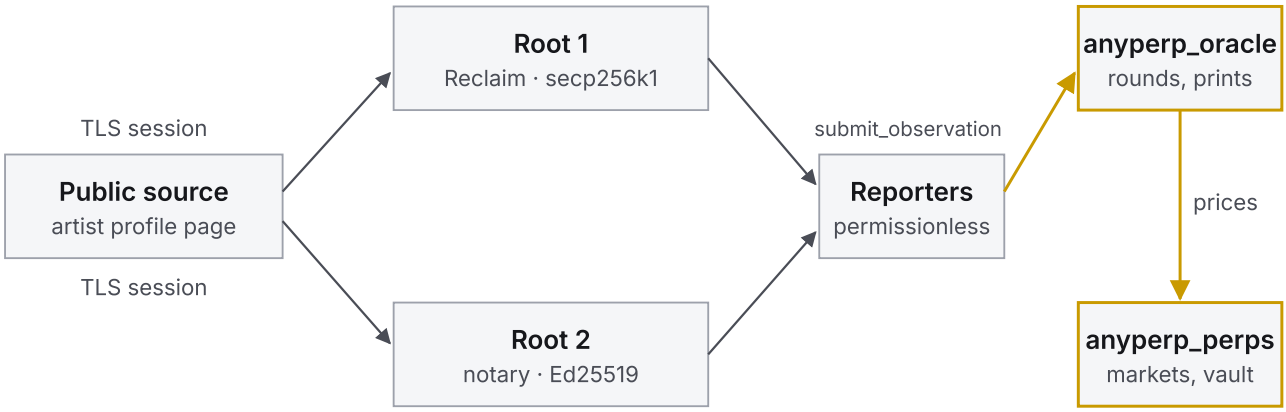
Root 1: Reclaim attestors

Root 1 is a set of Reclaim attestors. An attestor witnesses the TLS session in which the artist's public profile page is fetched and signs the value it returned. Each attestor is identified by the Ethereum-style address of its secp256k1 key; root 1's `AttestorSet` stores these 20-byte addresses, zero-padded to 32 bytes, and the program verifies their signatures through Solana's secp256k1 precompile. The program trusts exactly the addresses in its allowlist and does not call Reclaim's own on-chain verifier, so Anyperp's governance decides which attestors count.

Root 2: an independent notary

Root 2 is a notary service run by an operator independent of root 1. It fetches the same public page itself and signs the value it read with its own Ed25519 key. Its `AttestorSet` stores 32-byte Ed25519 public keys, and the program verifies its signatures through Solana's Ed25519 precompile.

Neither root prints alone. `finalize_round` requires an observation from each, and the two values have to agree within tolerance. A root that reports a different number either stays within the 0.5 % tolerance, which moves the median by at most half of it, or turns the round into a dispute with no price.



The public source is read by root 1 (Reclaim attestors) and root 2 (an independent notary); reporters submit both signed observations to anyperp_oracle, whose prints anyperp_perps reads.

Reporters carry the observations on chain. Reporting is permissionless: a reporter's key pays for and signs the transaction, and the program checks the attestation inside it, not the reporter.

Attestor sets and timelocked rotation

`initialize_oracle_config` creates the `OracleConfig` and both `AttestorSet` accounts once, signed by the oracle program's upgrade authority as the loader's `ProgramData` account records it. It names the perps curator PDA (`["curator"]`) as `curator_authority`, sets the guardian key and enforces the production policy: an attestor timelock of at least 172,800 s, a guardian pause limit of 86,400 s, hourly history buckets and 86,400 s reference and settlement windows. Each set takes 1 to 8 distinct, nonzero keys; a root-1 key is an address padded with 12 zero bytes.

A set admits a key for an observation according to the observation's fetch time t_{fetch}. The key must be in the current set with t_{fetch} in [`valid_from`, `valid_until`), or in the previous set with t_{fetch} in [`previous_valid_from`, `previous_valid_until`).

Rotation is a two-step change with a public waiting period.

1. `schedule_attestor_rotation(root, keys, valid_from, valid_until)` queues new keys. Its signer is `curator_authority`, which the perps program signs for only after three of five curators approve the same action through `approve_oracle_policy`. Activation is due `attestor_timelock_seconds` later, 48 hours in production, and a set holds one pending rotation at a time, so a queued change cannot be silently replaced.
2. `activate_attestor_rotation(root)`, open to anyone once the timelock has passed, makes the new keys current and turns the old keys into the previous epoch, whose window closes at the activation time. A claim fetched before activation stays verifiable under the keys valid when it was fetched, and the set's `generation` increases by one.

`cancel_attestor_rotation(root)` clears a queued rotation, again through a curator quorum. Scheduling emits `AttestorRotationScheduled` and activation `AttestorRotationActivated`, so every key change is public for the full timelock.

Observations

An observation is one root's signed reading of the count in one round. Each root has its own message format and its own instruction, `submit_reclaim_observation` for root 1 and `submit_notary_observation` for root 2; both run the same checks and record the observation the same way.

Time bounds

Each observation carries a source timestamp, when the source produced the value, and a fetch timestamp, when the root read it. Both must fall inside the round's window:

$$t_{open} \leq t_{source} \leq t_{fetch} \leq t_{now}, \qquad t_{fetch} < t_{close}, \qquad t_{source} < t_{close}$$

where t_{open} and t_{close} are the round's `opened_at` and `closes_at` and t_{now} is the chain clock at submission. In both root formats one attested time serves as both timestamps: a Reclaim claim's `timestampS`, and a notary message whose two timestamps must be equal. An observation read inside the window can still be submitted after `closes_at`, until the round reaches a terminal state.

Root 1: the Reclaim claim and its identifier

A Reclaim claim attests to an HTTPS request and what its response contained. It names a provider, the request parameters and a context, and its claim identifier is the Keccak-256 hash of the three. The oracle program builds every part of the claim itself:

Part	Content
Provider	`http`
Parameters	the request as JSON: method `GET`, an empty body, the artist's public profile URL, one response match, a regular expression that captures the integer after `"monthlyListeners"` as `monthly_listeners`, and no redactions
Provider hash	Keccak-256 of the parameters
Context	JSON with `contextAddress`, the metric account; `contextMessage`, `anyperp.reclaim.v1:` followed by the hex SHA-256 of the oracle program ID, the metric account, its spec hash, the round account and the round sequence; `extractedParameters`, the value as `monthly_listeners`; and `providerHash`
Claim identifier	Keccak-256 of the provider, the parameters and the context, joined by newlines
Signed claim	the identifier, the claim owner's address, `timestampS` and the attestor epoch, one per line

The attestor signs the claim as an Ethereum signed message, the prefix `\x19Ethereum Signed Message:\n` followed by the claim's length and the claim. The secp256k1 precompile hashes that message with Keccak-256 and checks the signature against the attestor's Ethereum address; the 65-byte signature is r, s and a recovery ID of 0 or 1.

A reporter submits only `artist_id`, `value`, `timestamp_s`, `owner` and `epoch`. The artist ID must be 22 ASCII letters or digits, and the SHA-256 of the metric's provider name and that ID in a fixed JSON form must equal the metric's `metric_id`, so a claim about one artist's page cannot be filed in another artist's market. A change to the metric, spec, round, value or timestamp changes the identifier, and a signature over one identifier verifies for no other. The program emits `AttestationVerified` with the claim identifier, owner and epoch.

Root 2: the notary message

The notary signs a fixed 178-byte message directly with Ed25519, a 64-byte signature:

Offset	Field	Bytes
0	Domain tag `ANYPERP_NOTARY_V1`	17
17	Oracle program ID	32
49	Metric account	32
81	Spec hash	32
113	Round account	32
145	Round sequence, little-endian u64	8
153	Root selector, 1	1
154	Raw source value, little-endian u64	8
162	Fetch timestamp, little-endian i64 Unix seconds	8
170	Source timestamp, little-endian i64 Unix seconds	8

The program rebuilds the message from its own accounts and the submitted value and timestamps, and accepts a signature only over exactly these bytes, so a notary signature cannot be replayed into another deployment, market or round.

Submitting an observation

An observation arrives as a transaction of two instructions: the precompile instruction that verifies the root's signature, followed by `submit_reclaim_observation(artist_id, value, timestamp_s, owner, epoch, verification_instruction_index)` or `submit_notary_observation(value, fetched_at, source_timestamp, verification_instruction_index)`. Both take the same accounts: the reporter (signer), the metric's `OracleConfig`, the metric, the round (a PDA of that metric), the root's `AttestorSet` (bound to the same config) and the instructions sysvar, checked by address.

The program requires, in order:

1. an `AttestorSet` of the instruction's root: `root` 0 (root 1) for a Reclaim claim, `root` 1 (root 2) for a notary message (`UnknownRoot` otherwise);
2. a metric that is not halted, and a round in `OPEN` or `AWAITING_ROOTS`;
3. no observation yet from that root in this round (`ObservationReplay` otherwise);
4. a positive value that normalizes to a positive index;
5. both timestamps inside the window and in order, as above (`StaleObservation`, `ObservationOutsideRound`, `FutureObservation`);
6. a verification instruction placed before this one in the transaction (`VerificationMustPrecede`);
7. a precompile instruction of the exact expected layout, over the exact message the program built;
8. a signer that the root's `AttestorSet` permits at the observation's fetch time.

The layout check leaves the precompile no room to verify anything else. The instruction must belong to the expected precompile, carry no accounts and exactly one signature, have an exact length and fixed offsets, and refer only to its own data. For secp256k1 that is the address at offset 12, the signature at 32 and the message at 97, with all three instruction indices pointing at the instruction itself; for Ed25519, 290 bytes with the public key at 16, the signature at 48 and the message at 112, and all three indices set to 65535. The runtime verifies the cryptography; the oracle verifies that what was verified is this message, signed by this key.

The round stores the observation with its value, timestamps, submission time, signer and signature, moves to `AWAITING_ROOTS` and emits `ObservationSubmitted`. Once both roots are present, it records `min_fetched_at`, the earlier of the two fetch times.

A third instruction, `submit_observation`, takes a generic 183-byte claim for either root under the domain tag `ANYPERP_LOCAL_CLAIM_V1`. The program accepts it only from a configuration created for local testing.

`finalize_round`

Anyone can call `finalize_round` with the metric, the round and the metric's `PriceHistory`; the keeper does so once both observations are in. It requires a metric that is not halted, a round in `OPEN` or `AWAITING_ROOTS`, and both observations present (`MissingRoot` otherwise). With v_1, v_2 the two raw values and τ the metric's `tolerance_bps` (50), the roots agree when

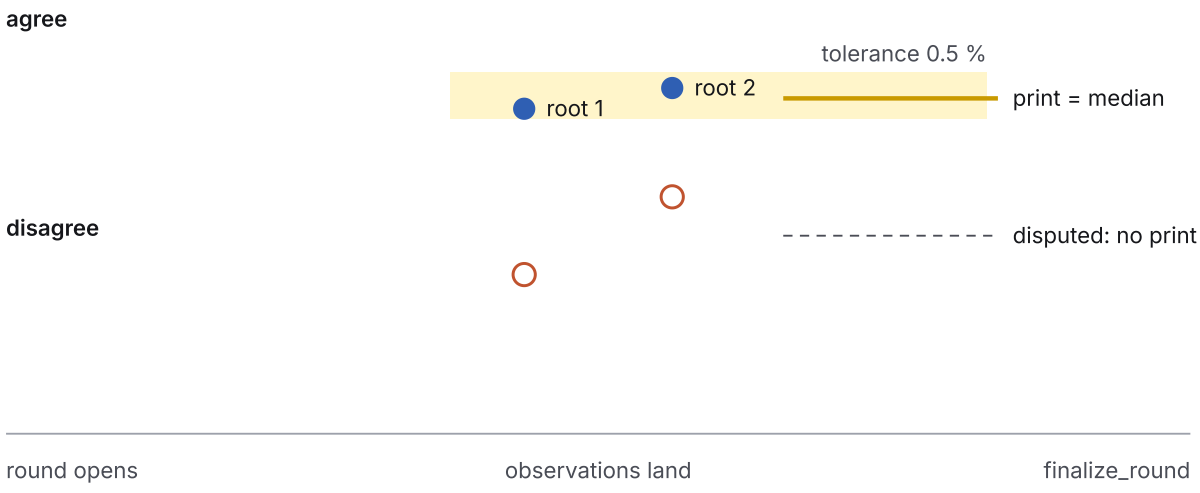
$$|v_1 - v_2| \cdot 10^4 \leq \min(v_1, v_2) \cdot \tau$$

that is, when they differ by at most 0.5 % of the smaller value, equality included. The round is `DISPUTED` if the roots disagree, if either observation's times fall outside the window, if a newer round of the metric has already been evaluated, or if its earliest fetch time is not later than the last print's. Otherwise the program takes the floored median and normalizes it:

$$I_t = \left\lfloor \frac{10^9}{D} \left(\min(v_1, v_2) + \left\lfloor \frac{|v_1 - v_2|}{2} \right\rfloor \right) \right\rfloor$$

A round whose index passes the circuit band, or confirms a level beyond it, finalizes. It stores `final_index`, `finalized_at`, `min_fetched_at` and the previous print's earliest fetch time, and the metric records it as its latest print, resets its count of consecutive rounds without a print, clears both side pauses and writes the close into the price history. Every evaluation emits `RoundEvaluated` with its outcome.

Every evaluation also updates the metric's agreement counters exactly once: `evaluated_rounds` always, `agreed_rounds` when the roots agreed within tolerance and inside the window, `failed_rounds` otherwise. A round whose roots agree on a level beyond the band counts as agreed. These counters are the shadow-mode and probation statistics of chapter 2.



One round: it opens, two roots post signed observations, the tolerance check compares them, and the round prints at their median or ends in a dispute.

The circuit band and side pauses

The band compares the new index with the history's most recent close at or before 24 hours earlier, R, the close at `now` − `reference_window_seconds` (86,400 s). With β the metric's `band_bps` (1,000), the index is in band when

$$|I_t - R| \cdot 10^4 \leq R \cdot \beta$$

that is, within 10 % of the close a day earlier. Shadow mode runs for 14 days before a market trades, so its history holds this reference long before the first order arrives.

A round beyond the band stays pending until later rounds confirm the level. A beyond-band round in the same direction as the pending one, whose index agrees with the pending index within τ, adds one to the metric's confirmation count; any other beyond-band round starts a new count at one. While the count is below the metric's `confirmation_rounds`, at least 2 and 2 at launch, the round ends `OUT_OF_BAND`. It writes no price and no history, adds to the count of consecutive rounds without a print, and pauses new positions on the side the move favors for `side_pause_seconds`, 21,600 s: an upward move sets `pause_long_until`, a downward move `pause_short_until`. The round that brings the count to `confirmation_rounds` finalizes at the new level. Later prints that agree with the confirmed level finalize directly while the 24-hour reference catches up, and the pending state clears once a print falls back inside the band.

A side pause stops opening orders on that side, both when they land and when they would fill. Closing orders, the other side, reporting and finalization continue, and any print clears both pause fields.

Disputed and failed rounds

A `DISPUTED` round has both observations but no price: the roots disagreed, a time fell outside the window, or the round arrived out of order. A `FAILED` round is missing an observation at the end of its window; anyone can mark it with `mark_failed_round` once `closes_at` has passed, while a round with both roots present has to be evaluated instead.

Both outcomes are rounds without a print. If the round is newer than the last evaluated one, the metric's `consecutive_failed_rounds` increases by one and opening orders on both sides pause for 21,600 s. A late older round still reaches a terminal state, but it cannot rewind or inflate the failure count. No terminal round blocks the next: the next round opens on schedule, and the first print after a dispute or failure resets the failure count and clears the pauses.

Guardian pauses, halts and settlement

`guardian_pause_metric(until)` lets the guardian pause opening orders in a market until at most `guardian_pause_seconds`, 86,400 s, after the start of the pause episode; a later call can extend the pause within that limit but never shorten it. `curator_pause_metric(until)` lets a curator quorum, through `approve_oracle_policy`, set a longer pause, lift one, or close an episode so the guardian can start another. Reporting, finalization, closing orders and cancellations continue through every pause.

`halt_metric` stops a metric for good. The guardian or the curator authority can call it once `consecutive_failed_rounds` has reached the metric's `failed_rounds_before_halt`, 4 at launch, one hour of rounds; below that count the program rejects the halt. `curator_halt_metric` is reserved for the curator PDA, which signs it when `slash_bond` halts a market that failed probation. A halt records `halted_at`. After it, no round opens and no observation lands.

A halted metric settles at a time-weighted average of its valid closes over the 24 hours before the halt. With h = `halted_at`, T = `settlement_window_seconds` = 86,400 s, and I_k the closes in effect over intervals Δt_k that tile $[h - T, h)$,

$$I_{settle} = \left\lfloor \frac{1}{T} \sum_k I_k \, \Delta t_k \right\rfloor$$

The average needs a close at or before $h - T$ to cover the start of the window. `settlement_twap` returns it; `settle_metric`, callable by anyone, stores it once in `settlement_index` with `settled_at`. No caller chooses the value, and a second settlement is rejected.

The price history and what the perps program reads

`PriceHistory` keeps one close per hour. Each print records its `finalized_at` and index: a later print in the same hour replaces that hour's close, and a print in a new hour takes the next slot of the 168-entry ring, overwriting the close from seven days earlier. Only `FINALIZED` rounds enter, in time order. The history supplies the band's 24-hour reference and the settlement TWAP's closes, and the production policy keeps both windows within the ring's 167-hour horizon.

The perps program never writes a price into the oracle. It reads every price from an `OracleRound`: `sync` accepts only a `FINALIZED` round of the market's own metric with a positive index and a sequence no older than the market's last, moves the market to that index and accrues funding and borrow since the previous print. Next-print execution uses the two fetch times each print stores. A print r fills an order that landed at time ℓ only when

$$t_{fetch}^{min}(r) > \ell \geq t_{fetch}^{min}(r_{prev})$$

where t_{fetch}^{min} is a print's `min_fetched_at` and r_{prev} the print before r. Every fetch in r happened after the order landed, and at least one fetch of the previous print did not, so r is the first eligible print. The perps program proves this from one round account without reading any other.

From the `MetricSpec`, every opening order reads `halted` and the three pause fields, the shadow and probation gates read `evaluated_rounds` and `agreed_rounds`, liquidation and settlement check `latest_finalized_sequence`, and settlement reads `settlement_index`. The perps program writes to the oracle only as the curator PDA: `initialize_metric` from `propose_market`, rotations and `curator_pause_metric` from `approve_oracle_policy`, and `curator_halt_metric` from `slash_bond`.

Instruction reference

Instruction	Signer	Effect
`initialize_oracle_config`	the oracle program's upgrade authority	creates `OracleConfig` and both `AttestorSet` accounts and enforces the production policy
`initialize_metric`	admin or `curator_authority` (through `propose_market`)	creates a `MetricSpec` and its `PriceHistory`; the spec cannot change afterwards
`schedule_attestor_rotation`	`curator_authority` (3 of 5 curators)	queues new keys for one root behind a timelock of at least 48 h
`activate_attestor_rotation`	anyone	activates a due rotation; the previous keys stay valid for claims fetched before activation
`cancel_attestor_rotation`	`curator_authority` (3 of 5 curators)	clears a queued rotation

Instruction	Signer	Effect
`open_round`	anyone, as payer	opens the next round once the cadence has elapsed
`submit_reclaim_observation`	any reporter, with a preceding secp256k1 signature from a permitted root-1 attestor	rebuilds the Reclaim claim and its identifier and records root 1's observation
`submit_notary_observation`	any reporter, with a preceding Ed25519 signature from a permitted root-2 key	rebuilds the notary message and records root 2's observation
`submit_observation`	any reporter, in a local test configuration only	records an observation signed over the generic 183-byte claim
`finalize_round`	anyone	evaluates a round with both observations: `FINALIZED`, `DISPUTED` or `OUT_OF_BAND`
`mark_failed_round`	anyone	marks a round missing a root after `closes_at` as `FAILED`
`guardian_pause_metric`	guardian	pauses opening orders for at most 24 h per episode
`curator_pause_metric`	`curator_authority` (3 of 5 curators)	sets, extends or lifts a pause and closes the guardian's episode
`halt_metric`	guardian or `curator_authority`	halts a metric after `failed_rounds_before_halt` consecutive rounds without a print
`curator_halt_metric`	`curator_authority` (from `slash_bond`)	halts a metric whose market failed probation
`settlement_twap`	anyone	returns the 24-hour TWAP ending at `halted_at`
`settle_metric`	anyone	stores the settlement TWAP once

5. Trading: next-print execution

A trader goes long or short a market's index with isolated USDC margin, and the vault takes the other side. A wallet holds at most one position per market, in a `Position` account derived from the market and the owner (`["position", market, owner]`), and that position carries at most one pending order at a time. Orders do not fill when they land; they wait for the market's next print. This chapter follows an order from the instruction that places it through the fill, the close and, when margin runs out, the liquidation.

The programs compute every amount in integers. Sizes, prices and indices are fixed9, so 1.0 is stored as 1,000,000,000. USDC amounts are base units with 6 decimals. A position of size q index units has a notional at price P of

$$N(q, P) = \left\lceil \frac{q \cdot P}{10^{12}} \right\rceil$$

USDC base units, the divisor 10^{12} bringing the product of two fixed9 numbers back to 6 decimals. The worked examples in this chapter use one example market: an artist at 60 million listeners with an index divisor of 1,000,000, so the index reads 60.000000000 and a position of 500 units has a notional of $30,000.

Placing an order with `open_order`

`open_order` takes six arguments.

Argument	Meaning
`nonce`	The order's client order id on chain. It seeds the order's address (`["order", owner, nonce]`) and must exceed the `last_nonce` stored in the wallet's `OwnerState`, so a signed order is never replayed. Orders prepared through the API carry a `clientOrderId`, and the API reserves the wallet's next nonce for it.
`long`	The side.
`reduce_only`	`false` opens a position or adds to one; `true` reduces or closes the position on that side.
`size`	The quantity q in index units, fixed9.
`margin`	USDC moved from the wallet into the vault as escrow. It funds the position's margin, the trading fee and the keeper fee. Zero for a reduce-only order.
`acceptable_price`	The worst fill the trader accepts, fixed9: a ceiling for a buy, a floor for a sell.

The instruction moves the margin into the vault and adds it to `reserved_margin_usdc`, which keeps it out of NAV from the moment it arrives. It writes a `PendingOrder` account with the order's terms and

its arrival: `landed_at`, the Solana clock time of the transaction, `landed_slot`, and `landed_round_sequence`, the latest round the oracle had opened for the metric.

An opening order lands only when the market is in probation or live, the metric is not halted, neither a guardian pause nor a pause on the chosen side is in force, the market is not marked insolvent, the wallet holds no open position on the other side, the margin is positive and the position has no other pending order. A reduce-only order needs an open position on the same side at least as large as the order, zero margin, a metric that is not halted and a market that is not settled. Side pauses and guardian pauses do not apply to it, so a trader can always reduce exposure while the market prints.

The pending order and the next print

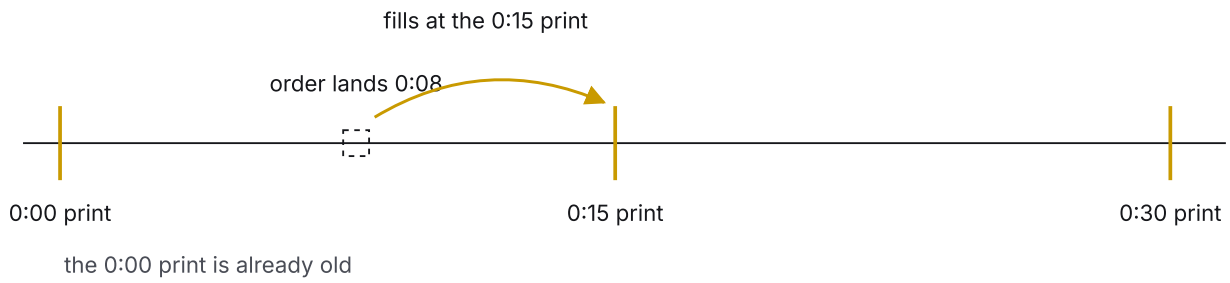
A landed order is a request with fixed terms. It fills at exactly one print: the first finalized round whose two observations were both fetched after the order landed. On chain, a round r is eligible for an order when

$$t_{min}^{fetch}(r) > t_{land} \geq t_{min}^{fetch}(r_{prev})$$

Here $t_{min}^{fetch}(r)$ is the round's `min_fetched_at`, the earlier of its two observations' fetch times, and $t_{min}^{fetch}(r_{prev})$ is the same time for the finalized round before it, which the oracle copies into each print as `previous_finalized_min_fetched_at`. The left inequality says both roots fetched the source after the order landed. The right one makes r the first such print. The oracle finalizes rounds in strictly increasing fetch time, and a round fetched no later than the previous print ends `DISPUTED`, so every order has exactly one eligible print. `execute_order` refuses every other round with `IneligibleRound`.

Execution waits for that print because the source is public. Monthly listeners appear on a public profile page, and anyone who reads it sees a new count before the oracle prints it. If orders filled at the latest print, a trader who saw the source change could trade at an index already known to be stale. Under next-print execution the fill uses observations fetched after the trader committed size, margin and acceptable price, so the price a trader receives is never older than what the trader could see when the order landed.

The wait is about one round. Rounds open every 15 minutes (`cadence_seconds` = 900), and each round accepts only observations fetched inside its own window. An order that lands before either root has fetched the current round can fill at that round's print. An order that lands after one root has already fetched it waits for the next round.



Rounds open every 15 minutes; an order that lands mid-round fills at the first print whose two observations were both fetched after it landed.

The order queue and its barrier

Each market keeps its pending orders in a first-in, first-out queue: a linked list of `PendingOrder` accounts whose ends sit on the `Market` as `queue_head` and `queue_tail`, with `pending_count` beside them. `open_order` appends at the tail. `execute_order` accepts only the head and refuses any other order with `NotQueueHead`. Orders therefore fill in landing order, and each one sees the skew left by the orders before it.

The queue also holds the market at the print its orders are owed. When a keeper synchronizes a market to a round that is eligible for the head order, the program records that round as the market's barrier (`barrier_sequence`, `barrier_min_fetched_at`). While the barrier stands, the market cannot move to any other print: synchronizing to a later round fails with `PendingEligibleOrder`, and a margin withdrawal that would read a later print fails the same way. When the head leaves the queue, the barrier stays if the new head landed before the barrier print was fetched, because that print is the new head's eligible print as well. It clears when the queue empties or the next head is owed a later print. Every order eligible at a print is processed at that print, in order, before the market advances to the next one.

Leverage and margin requirements

Leverage is at most 3×: `max_leverage_bps` is 30,000, and `initialize_exchange` accepts no value outside 10,000 to 30,000. A position with notional N needs initial margin

$$m_{init}(N) = \left\lceil \frac{N \cdot 10^4}{\lambda_{bps}} \right\rceil$$

with λ_{bps} = `max_leverage_bps`, which is $N/3$ at launch. A trader chooses lower leverage by escrowing more margin; the app's quote sizes the escrow for the chosen leverage and allows for the price of a print that has not happened yet.

Each market sets its maintenance requirement in `maintenance_margin_bps` (μ_{bps}), at or above the exchange's `min_maintenance_margin_bps`. The program accepts only values below the initial requirement, `maintenance_margin_bps` × `max_leverage_bps` < 100,000,000, and applies the requirement to the notional at the index:

$$m_{maint} = \left\lceil \frac{N(q, I_t) \cdot \mu_{bps}}{10^4} \right\rceil$$

The fill

A keeper fills the head order with `execute_order`, passing the order's eligible round. The instruction synchronizes the market to that round, which accrues funding and borrow up to the print (chapter 6) and moves the market's index to the print's `final_index`. For an opening order it then requires every open position in the vault to be valued at its market's current print (`unvalued_positions` = 0). It runs the whole fill once as a preflight, and only a fill that passes every check is applied.

Fill price

The fill price is the print's index adjusted for skew. With s the market's skew, long minus short open interest in USDC at the index, and S the market's skew scale `skew_scale_usdc`:

$$P = I_t \cdot \frac{2S + s_{before} + s_{after}}{2S}$$

The order moves the skew by its notional at the index, $N(q, I_t)$: up for a long open or a short close, which are buys and round up, and down for a short open or a long close, which are sells and round down. The adjustment is the average skew over the trade divided by S, so an order that adds to the skew trades on the far side of the index and an order that reduces it trades on the near side. A buy fills only if P is at or below `acceptable_price`, a sell only if P is at or above it.

In the example market, with an example skew scale of $10,000,000 and no skew, a long of 500 units moves the skew from 0 to $30,000 and fills at

$$P = 60 \cdot \frac{20{,}000{,}000 + 0 + 30{,}000}{20{,}000{,}000} = 60.090000000$$

Fees

Each fill pays a trading fee on its notional at the fill price and a flat keeper fee:

$$\phi = \left\lceil \frac{N(q, P) \cdot b_{trade}}{10^4} \right\rceil, \qquad b_{trade} = \texttt{trading_fee_bps} = 10$$

The keeper fee `keeper_fee_usdc` goes to the USDC account of the keeper that signed the fill. The trading fee is split 60 % to the vault, 20 % to the protocol, 10 % to the insurance fund and 10 % to

the market's proposer (chapter 7). In the example the fill has a notional of 500 × 60.09 = $30,045 and pays a trading fee of $30.045.

Margin check at the fill price

Fees come out of the escrow, and what remains is the position's margin, $m = m_{escrow} - \phi - k$, with k the keeper fee. A new position fills only if $m \geq m_{init}(N(q, P))$, the initial requirement at the fill price, and its equity at the print, $m + \mathrm{PnL}(q, P, I_t)$, covers m_{maint} at the index. In the example, the initial requirement is $10,015. An escrow of $10,100 with an example keeper fee of $0.01 leaves $10,069.945 of margin; the position opens $45 below water at the index (500 × (60 − 60.09)), and its equity of $10,024.945 clears an example maintenance requirement of 10 %, $3,000.

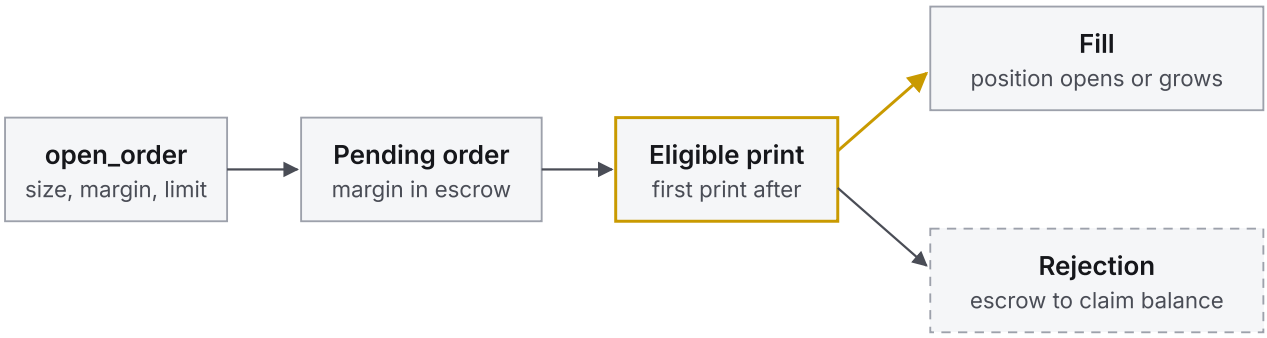
The fill also has to fit the market's cap on that side: the side's open interest including this order, valued at the index, must stay strictly below `cap_long_usdc` or `cap_short_usdc` (chapter 3). After the fill, the vault's cash must still cover every reserve it protects, and its NAV must stay above zero (chapter 7).

A fill that passes writes the position with entry price $P_{entry} = P$ and checkpoints at the market's current funding and borrow indices, grows the market's side size and entry basis, distributes the fee, pays the keeper and marks the order filled with its `fill_price` and `fill_sequence`, emitting `OrderFilled`.

Rejection and refundable escrow

An order that fails any check at its eligible print is rejected there. It is never carried to a later print. The program sets the order's state to rejected, records the print in `eligible_sequence`, removes the order from the queue, clears the position's pending flag and emits `OrderRejected`. The checks it can fail are the market's state and pauses at the print, an opposite-side position, the acceptable price, the side cap, the margin requirements at the fill price, the vault's reserves and, for a reduce-only order, equity too small to pay its fees.

The escrow stays in the vault, reserved for the owner, until `cancel_order` or `expire_order` returns it to the owner's claim balance (chapter 8) under the rules below. The order account then closes with `close_order`, which returns its rent to the owner.



A trader can cancel only once the eligible round has passed without a fill.

open_order escrows margin and records a pending order; at its eligible print the order fills into a position or is rejected, and a rejected order's escrow returns to the claim balance.

Adding to an open position

An opening order on the side of an existing position adds to it. The fill updates the same `Position` account. The new entry price is the size-weighted average of the old entry and the fill, rounded against the trader (up for a long, down for a short):

$$P'_{entry} = \frac{q_{old}\, P_{entry} + q_{add}\, P}{q_{old} + q_{add}}$$

Charges the position has accrued stay owed. Only the added quantity starts at the current indices: the program moves the funding checkpoint so that the merged size still owes the old quantity's funding,

$$F'_{cp} = F - \frac{q_{old}\,(F - F_{cp})}{q_{old} + q_{add}}$$

with the borrow checkpoint moved the same way and every division rounded so that no accrued charge is forgiven. The margin requirement applies to the whole position. The combined margin after fees must cover the initial requirement at the new entry, and the equity at the print must cover the initial requirement at the index, a stricter test than the maintenance requirement a new position meets. The cap check counts the whole side, existing position included.

If the example position adds 250 units at a later fill of 63.000000000, it becomes 750 units at an entry of (500 × 60.09 + 250 × 63) / 750 = 61.060000000.

Closing at the next print

A close is a reduce-only order. The app's close action prepares an `open_order` with `reduce_only` set, zero margin, a size up to the position's size and an acceptable price, and it fills at its eligible

print like any other order: a long closes with a sell, a short with a buy, at the skew-adjusted price. A close can take part of a position or all of it.

The closed quantity q_c takes its share of the margin and its share of the accrued charges c:

$$e_{close} = \left\lfloor \frac{m \cdot q_c}{q} \right\rfloor + \mathrm{PnL}(q_c, P_{entry}, P) - c(q_c)$$

The close fills only if e_{close} covers the trading fee and the keeper fee, and the rest, $e_{close} - \phi - k$, is credited to the owner's claim balance. The remaining position keeps its entry price, its checkpoints and the rest of its margin. A position closed in full has state closed, and `close_position` then closes the empty account and returns its rent to the owner.

Closing the example position of 500 units in full, at a later print of 66.000000000 with the position as the market's only open interest, moves the skew from $33,000 to zero and fills at 66 × (20,000,000 + 33,000) / 20,000,000 = 66.108900000, above the index, because the close reduces the skew. The close earns 500 × (66.1089 − 60.09) = $3,009.45 and pays a fee of $33.05445; before funding and borrow, $13,046.33055 reaches the claim balance.

Moving margin with `modify_margin`

`modify_margin` adds margin to an open position or takes margin out of it. Adding moves USDC from the wallet into the vault and raises the position's margin and the vault's reserved margin; it works at any time the position is open.

Removing margin requires a market in probation or live and a metric that is not halted, and it reads the metric's latest print. If the market has not synchronized to that print yet, the program projects funding and borrow forward to it without writing anything; a print beyond the barrier of queued orders is refused. The equity left must cover m_{init} at the print, and the equity at the market's recorded state must stay non-negative. The USDC goes straight to the owner's token account.

Cancelling with `cancel_order`

The owner of a pending or rejected order can cancel it with `cancel_order` when any of these holds:

Condition	Applies to
The metric is halted.	pending and rejected orders
`cancel_timeout_seconds` have passed since `landed_at`.	pending and rejected orders

Condition	Applies to
No eligible print exists yet: the oracle has evaluated rounds through `landed_round_sequence` + `cancel_after_rounds` (2), the latest print was fetched at or before landing, no later round is open and collecting observations, and the metric is not partway through confirming an out-of-band level.	pending orders
The market has synchronized past the print that rejected the order, to a round at least 2 after `landed_round_sequence`.	rejected orders

A pending order leaves the queue from its head. Cancelling returns the escrow to the owner's claim balance and emits `OrderCancelled`. While a round is open whose observations could still become the order's fill, the order stays in force, so a trader cannot watch a print form and withdraw ahead of it. `expire_order` does the same for anyone, without the owner's signature, once the timeout has passed or the metric is halted; keepers use it to clear stale orders.

Equity, maintenance and liquidation

A position's PnL at the index is

$$\mathrm{PnL}(q, P_{entry}, I_t) = \left\lfloor \frac{\sigma\, q\, (I_t - P_{entry})}{10^{12}} \right\rfloor$$

with $\sigma = +1$ for a long and -1 for a short, rounded down. Its equity is margin plus PnL minus the funding charge c_F and the borrow charge c_B accrued since its checkpoints (chapter 6):

$$e = m + \mathrm{PnL} - c_F - c_B$$

A position is liquidatable at a print where $e < m_{maint}$.

Liquidation is permissionless: any keeper signs `liquidate` with the position and a round. The instruction synchronizes the market to the round, which must be the metric's latest print or the barrier print of queued orders, and proceeds only if equity is below maintenance at the market's index (`NotLiquidatable` otherwise). The whole position closes at the index itself, with no skew adjustment, and pays a liquidation fee on its notional at the index:

$$\phi_{liq} = \left\lceil \frac{N(q, I_t) \cdot b_{liq}}{10^4} \right\rceil, \qquad b_{liq} = \texttt{liquidation_fee_bps} = 100$$

which is 1 % of the notional.

What is left of the position's equity pays out in a fixed order. The keeper fee comes first and goes to the liquidating keeper. The liquidation fee comes next, as far as equity reaches, and is split like a trading fee: 60 % to the vault, 20 % to the protocol, 10 % to the insurance fund and 10 % to the proposer. The remaining margin, $\max(e, 0) - k - \phi_{liq}$ where it is positive, is credited to the owner's claim balance. When equity is negative, the insurance fund pays the shortfall into the vault; a

shortfall beyond the fund's balance is recorded in `bad_debt_usdc`, and the market is marked insolvent and takes no new opening orders. The position ends in state liquidated and emits `PositionLiquidated`.

The liquidation price

Setting equity equal to the maintenance requirement and solving for the index gives the liquidation price, with $\mu = \mu_{bps}/10^4$ and $c = c_F + c_B$:

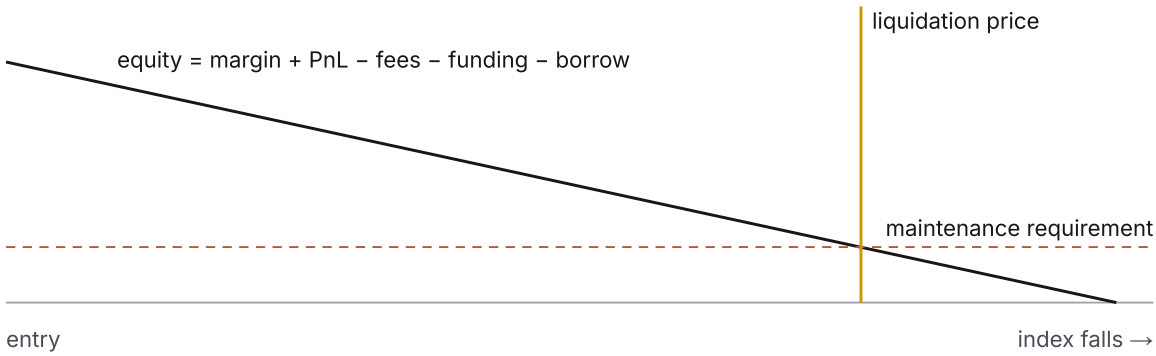
$$P_{liq,long} = \frac{P_{entry} - (m - c)/q}{1 - \mu}, \qquad P_{liq,short} = \frac{P_{entry} + (m - c)/q}{1 + \mu}$$

The program computes it from the exact integers, rounding up for a long and down for a short, so a long's reported price is never below the exact threshold and a short's never above it. A long whose margin net of charges exceeds its entry value has no liquidation price. The price moves as funding and borrow accrue, and liquidation itself always checks exact equity at the print.

In the example, the 500-unit long with $10,069.945 of margin, no charges yet and an example maintenance requirement of 10 % has

$$P_{liq} = \frac{60.09 - 10{,}069.945/500}{0.9} = 44.389011112$$

At a print of 44.389011112 its equity is $2,219.450556, exactly its maintenance requirement; at any print below it, a keeper can liquidate.



Equity falls with the index toward the maintenance requirement; where the two lines cross is the liquidation price, and a print below it lets any keeper liquidate.

6. Funding and the borrow fee

Two charges accrue on every open position between prints. Funding is signed: it moves value between the side that adds to the vault's net exposure and the side that reduces it. The borrow fee is unsigned: both sides pay it, at a rate that rises as the market fills its caps. Each charge accrues into a cumulative index on the `Market` account, advanced once per print the market consumes, and each position settles its charges against those indices when its quantity closes. Monthly listeners have no spot market, so funding cannot pull a perpetual's price toward a spot price the way it does for a traded asset. It works as the price of the vault's exposure instead, and it keeps moving until traders take that exposure off the vault.

Skew and the skew scale

A market's skew is its long open interest minus its short open interest, both valued at the index:

$$s = N(q_{long}, I_t) - N(q_{short}, I_t)$$

where q_{long} and q_{short} are the market's `long_size` and `short_size`. The vault holds the opposite of every trade, so its net exposure in the market is $-s$. A positive skew means traders are net long and the vault is net short.

The skew scale S is the market's `skew_scale_usdc`, in USDC. It sets how strongly the skew acts on prices. In the fill price of chapter 5 the skew enters as $(s_{before} + s_{after})/2S$, and in funding it enters as s/S. A market with a larger S moves its fill prices and its funding rate more slowly for the same imbalance.

The parameters of this chapter are set per market in `MarketParams`. The proposer submits them with the market, and the curators' approval hash binds them, so a market trades with exactly the parameters the curators approved.

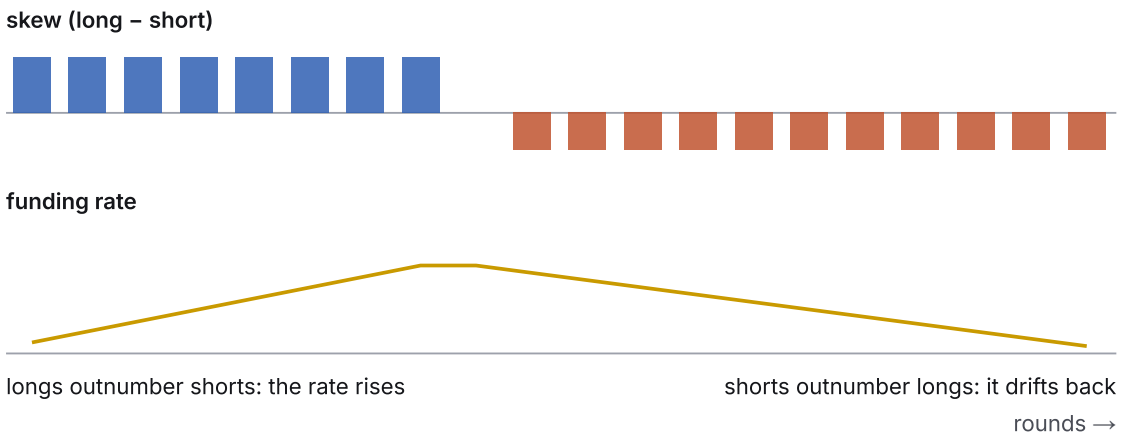
Parameter	Symbol	Field	Unit
Skew scale	S	`skew_scale_usdc`	USDC base units
Funding velocity	v	`funding_velocity`	fixed9 per hour, per hour, at $s = S$
Borrow base rate	b_0	`borrow_base_rate`	fixed9 per hour
Borrow slope	b_1	`borrow_slope_rate`	fixed9 per hour at full utilization
Side caps	cap_{long}, cap_{short}	`cap_long_usdc`, `cap_short_usdc`	USDC base units

Velocity funding

The funding rate f is stored on the market as `funding_rate`, a signed fixed9 fraction of the index per hour. A positive rate means longs pay and shorts receive; a negative rate means the reverse. The skew does not set the rate directly. It sets the rate's velocity:

$$\dot{f} = v \cdot \frac{s}{S}$$

While traders are net long, the rate rises, so longs pay more and shorts are paid more, and the incentive to go short or close a long grows for as long as the skew lasts. While traders are net short, the rate falls. When the skew is zero, the rate holds where it is. The rate therefore drifts toward paying whichever side reduces the skew, and its drift changes direction only when the skew changes sign. Under a persistent imbalance the rate keeps moving; in a balanced market it stays at the level it reached.



The skew sets the velocity of the funding rate: under a long skew the rate climbs, under a short skew it falls, and it drifts toward paying the side that reduces the skew.

The discrete update at each print

The program advances funding each time a market synchronizes to a new print, in `sync_market` and in the synchronization at the start of `execute_order` and `liquidate`. The interval is the time between the two prints' finalization, Δt = `finalized_at` of the new round minus the market's `last_accrued_at`, in seconds. The step uses the state the market held during the interval: the previous print's index I_{t-1} and the skew s_{t-1} at that index. `funding_step` computes:

$$\Delta f_t = \left\lfloor \frac{v \cdot s_{t-1} \cdot \Delta t}{S \cdot 3600} \right\rfloor$$

$$f_t = f_{t-1} + \Delta f_t$$

$$\bar{f}_t = \left\lfloor \frac{(f_{t-1} + f_t) \cdot \Delta t}{2 \cdot 3600} \right\rfloor$$

$$\Delta F_t = \left\lfloor \frac{I_{t-1} \cdot \bar{f}_t}{10^9} \right\rfloor, \qquad F_t = F_{t-1} + \Delta F_t$$

The first line is the velocity applied over the interval. The rate changes linearly between prints, so the third line integrates it by the trapezoid rule: $\bar{f}_t$ is the fraction of the index that accrues as funding over the interval. The fourth line turns that fraction into index units and adds it to the cumulative funding index F, the market's `funding_index`. Floors are signed and round toward negative infinity. The program also adds $|\Delta F_t|$ to a second index, `funding_absolute_index`, which the vault uses to reserve cash for funding credits.

An example step, with example parameters: index 60.000000000, skew scale $10,000,000, a long skew of $150,000, a funding velocity of 0.001 per hour per hour (`funding_velocity` = 1,000,000), a rate of zero before the step and one 15-minute interval of 900 seconds.

Quantity	Computation	Result (fixed9)
Δf_t	1,000,000 × 150,000,000,000 × 900 / (10,000,000,000,000 × 3,600)	3,750
f_t	0 + 3,750	3,750 (0.00000375 per hour)
$\bar{f}_t$	(0 + 3,750) × 900 / 7,200, floored	468
ΔF_t	60,000,000,000 × 468 / 10^9	28,080

If the skew and the index hold, the rate climbs by the same 3,750 at every 15-minute print and reaches 360,000, a rate of 0.00036 per hour, after 96 prints, one day.

The cumulative funding index and position accrual

A position records the funding index when it opens, as `funding_checkpoint` F_{cp}. From then on its accrued funding is

$$c_F = \left\lceil \frac{\sigma \cdot q \cdot (F - F_{cp})}{10^{12}} \right\rceil$$

in USDC base units, with $\sigma = +1$ for a long and -1 for a short. A positive c_F is a charge to the trader and a negative one is a credit. Because the index accumulates $I \cdot \bar{f}$, a position pays its notional times the integrated rate, whatever its entry price.

Nothing moves between accounts at a print. The charge accrues in the index and enters the position's equity at once: every liquidation check, every valuation and every margin withdrawal reads c_F at the current index. It is paid when quantity closes. A close settles the funding of the closed quantity out of that quantity's margin and PnL, a liquidation settles all of it, and a settlement

at a halt settles all of it at the last valid print. A position that adds quantity keeps its old funding owed through the merged checkpoint described in chapter 5, so adding never resets what a position owes or is owed.

The borrow fee

The borrow fee charges open interest for the cap capacity it occupies. Its rate follows the market's utilization u, in basis points, which is the higher of the two sides' open interest as a share of their caps:

$$u = \max\left(\left\lceil \frac{10^4 \cdot N_{long}}{cap_{long}} \right\rceil, \left\lceil \frac{10^4 \cdot N_{short}}{cap_{short}} \right\rceil\right)$$

The caps are the market's active caps, so during probation utilization is measured against 25 % of the approved caps. The hourly borrow rate is a base plus a slope in utilization:

$$b = b_0 + \left\lceil \frac{b_1 \cdot u}{10^4} \right\rceil$$

At each print the program advances the borrow index B, the market's `borrow_index`, with the same interval and previous index as funding:

$$\Delta B_t = \left\lceil \frac{I_{t-1} \cdot b \cdot \Delta t}{10^9 \cdot 3600} \right\rceil, \qquad B_t = B_{t-1} + \Delta B_t$$

B only increases. A position records it at opening as `borrow_checkpoint` B_{cp}, and its accrued borrow is

$$c_B = \left\lceil \frac{q \cdot (B - B_{cp})}{10^{12}} \right\rceil$$

Both sides pay it, longs and shorts alike. Like funding, it enters equity at once and is collected when quantity closes. At collection the borrow fee is split like every fee the protocol takes: 60 % to the vault, 20 % to the protocol, 10 % to the insurance fund and 10 % to the market's proposer.

Rounding

Each step rounds in a fixed direction. The funding step uses signed floors throughout; every amount charged to a position and every figure in the vault's books rounds in the vault's favor.

Quantity	Rounding
Rate step, integrated rate, funding index step	signed floor

Quantity	Rounding
Position funding c_F	ceiling: a charge is never understated, and a credit never overstated
Utilization, borrow rate, borrow index step, position borrow c_B	ceiling
Funding and borrow checkpoints after an increase	rounded so the merged position owes at least the old quantity's charges
Funding and borrow receivables in NAV	floor, so NAV never counts rounding income before it is collected
Funding credit reserve	ceiling

How funding and borrow reach the vault

The vault is the counterparty to both charges. Each print adds to three books on the market. The funding receivable basis grows by $(q_{long} - q_{short}) \cdot \Delta F_t$: what longs owe minus what shorts are owed, which is the vault's net funding income. The borrow receivable basis grows by $(q_{long} + q_{short}) \cdot \Delta B_t$, since both sides pay. The funding credit bound grows by $(q_{long} + q_{short}) \cdot |\Delta F_t|$. Position changes remove the bases of the quantity that closes, and a position that adds replaces its bases whole, so the books always describe the open positions exactly.

From the first two books the program computes the outstanding charges it counts in NAV:

$$\Phi = \left\lfloor \frac{\Sigma_F}{10^{12}} \right\rfloor + \left\lfloor \frac{6000}{10^4} \left\lfloor \frac{\Sigma_B}{10^{12}} \right\rfloor \right\rfloor$$

Σ_F and Σ_B are the funding and borrow receivable bases. The vault counts its net funding receivable in full and only its 60 % share of borrow that has not been collected yet, because 40 % of every borrow fee leaves the vault at collection. The sum across markets is the vault's `total_charges_usdc`, an asset in its balance sheet (chapter 7).

From the third book the program computes the funding credit reserve $\lceil \Sigma_{|F|}/10^{12} \rceil$, the vault's `reserved_funding_usdc`. It bounds every funding credit not yet paid, and the vault keeps at least that much cash in reserve, alongside trader margin and claim balances, before it takes on new exposure or pays out a withdrawal.

Cash moves when quantity closes. A long that paid funding receives a smaller payout, and the difference stays in the vault. A short that earned funding receives a larger payout from vault cash. The vault's net over a skewed market is $(q_{long} - q_{short}) \cdot \Delta F$: positive when the rate charges the side the skew is on, which is where the velocity drives it. The borrow fee collected from the closing quantity leaves the payout, and the 40 % that belongs to the protocol, the insurance fund and the proposer moves to their accounts in the same instruction.

7. The vault

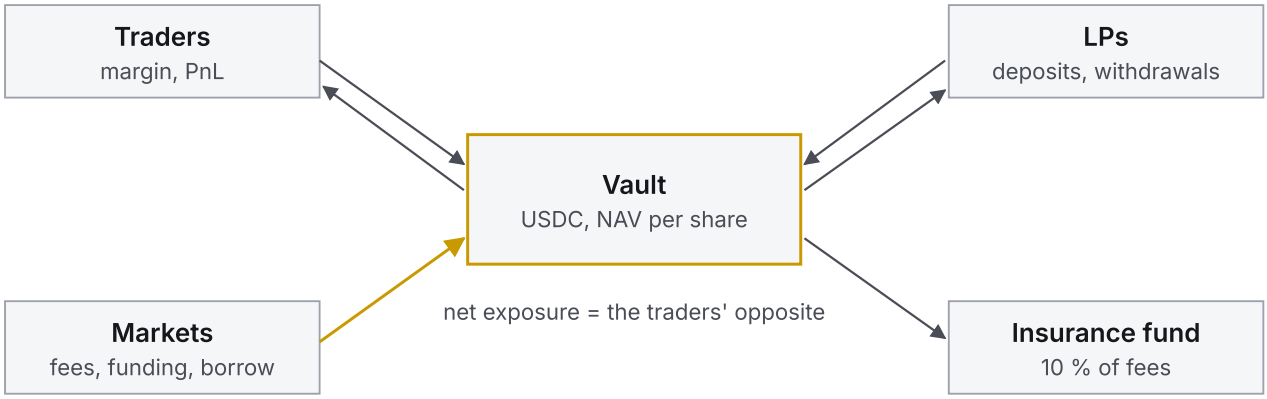
One USDC vault takes the other side of every trade in every market. Liquidity providers deposit USDC into it and receive vault shares, a classic SPL token whose only claim is a pro-rata part of the vault's net asset value. The vault earns what traders lose, the funding the skewed side pays and 60 % of every fee. It pays what traders win and the funding credits of the side that reduces the skew. This chapter describes the vault's accounts and balance sheet, how its NAV is computed and frozen, how deposits and withdrawals are priced against it, and where fees and the insurance fund sit.

One vault behind every market

The `Vault` account is a single PDA (`["vault"]`) that serves up to 24 markets. It controls these accounts:

Account	Seeds	Holds
Vault USDC account	`["usdc"]`	the vault's cash: LP capital, trader margin and escrow, pending deposits, priced withdrawals and claim balances
Share mint	`["shares"]`	the vault share, 9 decimals, minted and burned only by the vault PDA
Share escrow	`["share_escrow"]`	shares queued for withdrawal and shares credited to claim balances
Dead shares	`["dead_shares"]`	the shares minted against the seed deposit, which no instruction moves
Insurance fund	set in `Exchange.insurance_token`	the insurance fund's USDC, owned by the vault PDA and kept apart from vault cash
Proposer fee pool	set in `Exchange.proposer_token`	proposers' fee shares until each proposer claims them

Every market's PnL, funding and fees settle against the same cash. A market's long and short open interest are separate positions to traders, but to the vault they net: in each market the vault holds $-s$, the opposite of that market's skew, and its exposure across the service is the set of those 24 or fewer net positions.



The vault as counterparty: margin, fees, funding and PnL move between traders and the vault; LPs deposit USDC for shares and withdraw through the queue.

The balance sheet

The vault's NAV is its cash minus everything in that cash that belongs to someone else, minus what it owes traders on open positions, plus what traders owe it in charges:

$$\mathrm{NAV} = C - (M + D + W + K) - \Pi + \Phi$$

Term	Field	Meaning
C	balance of the vault USDC account	all USDC the vault holds
M	`reserved_margin_usdc`	margin of open positions and escrow of pending orders
D	`pending_deposits_usdc`	deposits waiting for their NAV
W	`reserved_withdrawals_usdc`	withdrawals already priced, owed to their LPs
K	`reserved_claims_usdc`	claim balances owed to wallets (chapter 8)
Π	`total_pnl_usdc`	traders' unrealized PnL across all markets, at each market's index
Φ	`total_charges_usdc`	funding and borrow traders owe the vault, net (chapter 6)

Each market keeps the inputs to its share of Π: its long and short sizes and its long and short entry bases, the sums $V = \sum q_i P_{entry,i}$ over its positions. Its unrealized PnL at the index is

$$\Pi_{market} = \left\lfloor \frac{I_t\,(q_{long} - q_{short}) - (V_{long} - V_{short})}{10^{12}} \right\rfloor$$

and the program refreshes it at every print and every position change. When traders are up, Π is positive and NAV is lower by that amount; when they are down, NAV is higher.

Beside NAV, the vault holds two cash reserves that NAV does not subtract. The funding credit reserve R_F (`reserved_funding_usdc`) bounds the funding credits not yet paid (chapter 6). The exposure reserve R_X (`reserved_exposure_usdc`) bounds what traders could be owed in gross profit. A long's profit never exceeds its value at the index and a short's profit never exceeds its entry value, so per market

$$R_X = \max\left(\left\lceil \frac{|N_{long} - N_{short}| \cdot r}{10^4} \right\rceil,\ N_{long} + \left\lceil \frac{V_{short}}{10^{12}} \right\rceil\right)$$

with r the exchange's `vault_reserve_bps`. The protected reserves are $M + D + W + K + R_F + R_X$. An opening fill goes through only if the vault's cash still covers them afterward and NAV stays above zero; a withdrawal executes only if cash covers the withdrawal on top of them; a claim pays out only if cash covers trader margin, pending deposits and all claim balances, the claim included.

For example, a vault with cash of $1,250,000 holds $180,000 of margin and escrow, $50,000 of pending deposits, $20,000 of priced withdrawals and $5,000 of claim balances. Its traders are up $12,000 and owe $1,500 in net charges. Its NAV is

$$1{,}250{,}000 - 255{,}000 - 12{,}000 + 1{,}500 = 984{,}500$$

Shares and NAV per share

The share mint has 9 decimals. `initialize_vault` takes the vault's seed deposit from the admin and mints `initial_shares`, 1,000 share base units per USDC base unit, so the first share is worth 1 USDC. Those shares go to the dead shares account. No instruction transfers or burns them, so the share supply never returns to zero, and every later deposit and withdrawal is priced against an existing supply.

With Z the share supply in base units, the price of one share in fixed9 USDC is

$$p = \left\lfloor \frac{\mathrm{NAV} \cdot 10^{12}}{Z} \right\rfloor$$

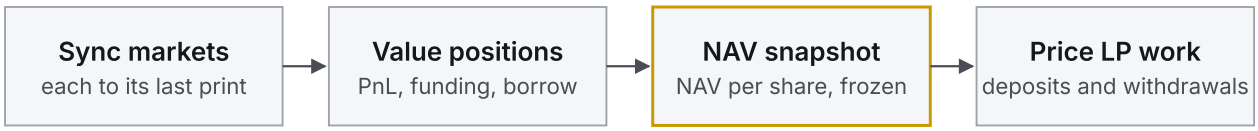
In the example, with 980,000 shares outstanding, p = 1.004591836.

NAV generations

Deposits and withdrawals are priced at a NAV generation: a cohort in which every market has been valued at a fresh print. The vault numbers generations with `valuation_sequence`, and every instruction that changes valuation names the generation it works on as its `epoch` argument. A generation completes in three steps, and LP intents are priced at it in a fourth.

1. **Sync markets.** `sync_market` moves a market to its latest print, accruing funding and borrow and refreshing PnL; `execute_order` and `liquidate` synchronize the same way. The first synchronization after a completed generation opens the next one. Synchronizing marks every open position in the market as unvalued.
2. **Value positions.** `value_position` checks each open position at its market's print and requires its equity to be zero or more. A position with negative equity fails with `NegativePositionEquity` and has to be liquidated first, so NAV never counts a loss that a trader's margin cannot pay. When a market's last position is valued, the market joins the generation and contributes its print's `min_fetched_at`; a market without positions joins when it synchronizes. If a market moves to a newer print before the generation completes, its earlier entry is withdrawn and it is valued again at the newer print.
3. **Snapshot.** The instruction in which the last registered market joins completes the generation's `NavSnapshot` account (`["nav", epoch]`). It records `nav_usdc`, `share_supply`, the generation's `min_fetched_at` (the earliest print fetch time among its markets), `previous_min_fetched_at` (the same time for the previous generation), `completed_at` and a `distressed` flag. A completed snapshot never changes. A `NavCompleted` event announces it.
4. **Price LP intents.** Keepers price the deposits and withdrawals that are eligible at the snapshot, at its frozen NAV and share supply, in queue order. A new generation cannot open while the completed one has an eligible deposit or withdrawal at the head of its queue still unpriced (`PendingLpIntent`).

Opening fills wait for valuation as well: an opening order executes only when no position in the vault is unvalued. A newly activated market joins at a completed generation boundary and starts a fresh generation that must include it. Keepers close snapshot accounts more than 24 generations old and return their rent to the keepers that created them.



A deposit or withdrawal fills only at a generation completed after it became eligible.

A NAV generation: every market syncs to a fresh print, every position is valued, the snapshot freezes NAV and share supply, and deposits and withdrawals are priced at it.

Deposits

`deposit` moves USDC from the wallet into the vault and writes a `PendingDeposit` (`["deposit", owner, nonce]`) at the tail of the vault's deposit queue. The amount joins `pending_deposits_usdc`,

so it does not count in NAV while it waits. A keeper fills it with `execute_deposit` at the first complete generation fetched after it landed:

$$t_{min}^{fetch}(g) > t_{land} \geq t_{min}^{fetch}(g_{prev})$$

The deposit receives

$$z = \left\lfloor \frac{a \cdot Z_g}{\mathrm{NAV}_g} \right\rfloor$$

shares, with a the deposit, and NAV_g and Z_g taken from the snapshot. The shares are minted to the share escrow and credited to the owner's claim balance, from which the owner can claim them to a wallet or queue them straight into a withdrawal. Each generation's `NavMintAllocation` account counts the shares minted against it, so the frozen quote prices every deposit in its cohort the same way.

The rule mirrors next-print execution. Every market in the generation has a print fetched after the deposit landed, so the NAV that prices the deposit already contains every count the depositor could have seen. A deposit placed after the source moved buys shares at a NAV that includes the move.

Continuing the example, the pending $50,000 deposit fills at a snapshot with NAV $984,500 and 980,000 shares and receives 49,771.457592686 shares. Its USDC leaves D and enters NAV, which rises to $1,034,500, and the price per share stays 1.004591836.

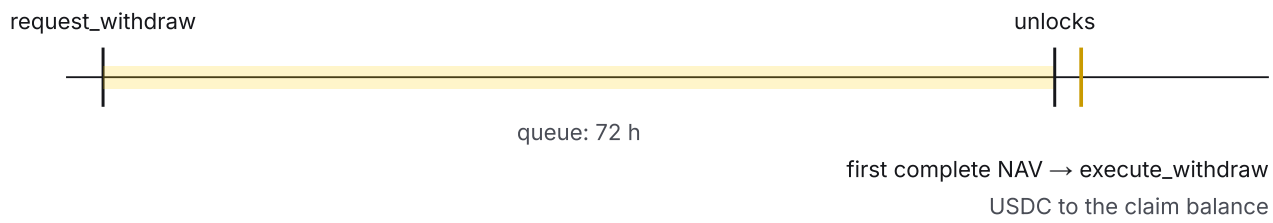
A deposit that no generation has priced can be recovered. `cancel_deposit` returns it to the owner's claim balance once `withdrawal_queue_seconds` have passed since it landed, provided no complete generation eligible for it exists.

Withdrawals

A withdrawal runs in three instructions.

1. **Request.** `request_withdraw` takes a number of shares, first from the owner's claimable shares and the rest from a share account in the wallet, and moves them into the share escrow. It writes a `Withdrawal` (`["withdraw", owner, nonce]`) at the tail of the withdrawal queue with `unlocks_at` = `requested_at` + `withdrawal_queue_seconds`. The queue is 259,200 seconds, 72 hours, and outside local test configuration `initialize_exchange` refuses anything shorter. Queued shares stay in the supply and keep their share of the vault's gains and losses while they wait.
2. **Price.** After the unlock, a keeper calls `price_withdraw` with the first complete generation fetched after `unlocks_at`, by the same rule as deposits. The withdrawal is worth $\lfloor z \cdot \mathrm{NAV}_g / Z_g \rfloor$ USDC. Its shares are burned, the amount moves into `reserved_withdrawals_usdc`, and from then on it is a fixed claim outside NAV.
3. **Execute.** `execute_withdraw` credits the amount to the owner's claim balance once the vault's current NAV is non-negative and its cash covers the amount on top of the protected reserves.

The queue keeps LPs from leaving ahead of a print they can already see. An LP who reads a move at the source cannot exit at a NAV that has not printed it: the withdrawal is priced only after 72 hours, at a generation every market of which was fetched after the unlock.



A withdrawal request waits 72 hours, is priced at the first complete NAV after it unlocks, executes when cash covers it, and lands in the claim balance.

Distressed NAV

A snapshot whose raw NAV is zero or below is marked `distressed` and records a NAV of zero. Nothing is priced against it. A deposit eligible at a distressed snapshot is returned in USDC to the owner's claim balance. A withdrawal eligible at a distressed snapshot is not burned: `price_withdraw` returns its shares to the owner's claimable shares, closes the request and emits `WithdrawalReturned`, and the LP holds the shares again and can request a new withdrawal. While NAV is not positive the vault opens no new positions. `recapitalize_vault` lets any signer add USDC to the vault without receiving shares.

Fees and the fee split

Fee	Charged on	Rate
Trading fee	every opening, adding and closing fill, on notional at the fill price	`trading_fee_bps` = 10
Liquidation fee	every liquidation, on notional at the index	`liquidation_fee_bps` = 100
Borrow fee	collected from each closing quantity, as accrued (chapter 6)	per market

Every one of them is split the same way. `initialize_exchange` accepts only `vault_fee_bps` = 6,000, `protocol_fee_bps` = 2,000, `insurance_fee_bps` = 1,000 and `proposer_fee_bps` = 1,000: 60 % to the vault, 20 % to the protocol, 10 % to the insurance fund and 10 % to the market's proposer. The three external shares round down and the vault keeps the remainder:

$$\phi_{vault} = \phi - \left\lfloor \frac{2000\,\phi}{10^4} \right\rfloor - \left\lfloor \frac{1000\,\phi}{10^4} \right\rfloor - \left\lfloor \frac{1000\,\phi}{10^4} \right\rfloor$$

The vault's share stays in vault cash and raises NAV; `fees_earned_usdc` records it. The protocol's share moves to the protocol treasury's USDC account and the insurance share to the insurance fund; it buys ANYPERP on the market, which is burned (chapter 15). The proposer's share moves to the proposer fee pool and is booked to the market in `claimable_proposer_usdc`, and the market's proposer withdraws it with `claim_market_fees`. A `FeesPaid` event records each split. Keeper fees sit outside the split: `keeper_fee_usdc` is paid whole to the keeper that runs a fill, liquidation or settlement. A settlement at a halt charges no trading fee.

Each trading, liquidation and borrow fee splits 60 % to the vault, 20 % to the protocol, 10 % to the insurance fund and 10 % to the market's proposer.

The insurance fund

The insurance fund is a USDC account owned by the vault PDA and named in `Exchange.insurance_token`. It is not vault cash and not part of NAV. Two flows fill it: 10 % of every trading, liquidation and borrow fee, and the bonds of proposers whose markets fail during probation, which `slash_bond` transfers in whole. One flow draws on it. When a close, a liquidation or a settlement finds a position's equity below zero, the program moves the shortfall from the insurance fund into vault cash, up to the fund's balance, so the vault's LPs do not carry it. Any part beyond the fund's balance is recorded in the vault's `bad_debt_usdc`, and the market is marked insolvent, which stops new opening orders there.

Queue limits

Queue	Limit
Pending deposits in the vault	`max_pending_deposits`, from 1 to 64
Pending withdrawals in the vault	`max_pending_withdrawals`, from 1 to 64
Pending deposits per wallet	1 (`OwnerState.pending_deposit`)
Pending withdrawals per wallet	1 (`OwnerState.pending_withdrawal`)
Pending orders per position	1 (`Position.has_pending_order`)
Markets per vault	24

The two vault queue sizes are set in protocol configuration, and `initialize_exchange` accepts only values in that range. A deposit or withdrawal beyond a limit fails with `LpQueueFull`, and the wallet places it again once its earlier intent is priced or the queue has room. Both queues are first-in, first-out, priced from the head, and each entry is priced at the one generation its rule selects.

8. Halts, settlement and claims

A market stops when its source stops producing prints that the two roots agree on. The oracle counts the rounds that end without a print, the guardian halts the metric once the count reaches the market's threshold, and the market settles at the time-weighted average of its last 24 hours of valid prints. Each position then closes at that price, and what it is owed lands in the owner's claim balance, the protected internal balance that receives the system's payouts and that the owner withdraws with `claim_balances`.

Failed rounds and the halt threshold

A round that does not print ends in one of three states (chapter 4). It is `FAILED` when `mark_failed_round` finds a root's observation missing after the round's `closes_at`. It is `DISPUTED` when both roots reported but their values differ by more than the tolerance of 50 bps, a timestamp falls outside the round's window, or the round was fetched no later than the previous print. It is `OUT_OF_BAND` when the roots agree on a level beyond the circuit band and the level waits for confirmation by later rounds.

Each of these adds one to the metric's `consecutive_failed_rounds`; a failed round older than the last one evaluated reaches its terminal state without adding to the count. A failed or disputed round also pauses opening orders on both sides for `side_pause_seconds`, 21,600 seconds (6 hours); an out-of-band round pauses the side in the direction of the move. The next print resets the count to zero and lifts the side pauses.

The threshold is the metric's `failed_rounds_before_halt`, set in its `MetricSpec` when the market is proposed. It is part of the configuration the curators' approval hash binds, so every market trades under the halt threshold its curators approved.

The guardian's halt

The guardian is a separate signer with two powers over a metric, both in `anyperp_oracle`. `guardian_pause_metric` pauses opening orders until a chosen time, within `guardian_pause_seconds` of the episode's start; only a curator decision, through `curator_pause_metric`, clears an episode or starts another. `halt_metric` halts the metric, and the oracle accepts it, from the guardian or the curator authority, only when `consecutive_failed_rounds` has reached `failed_rounds_before_halt`; below the threshold it fails with `HaltThresholdNotMet`. The guardian writes no price and moves no funds.

The guardian worker checks every metric on each pass and signs `halt_metric` as soon as a metric reaches its threshold. The halt sets `halted` and records `halted_at`, the clock time of the halt, and emits `MetricHalted`. A market can also halt during probation: when `slash_bond` finds root agreement below 99 % at the end of probation, the market insolvent or the metric already halted, the

curator PDA halts the metric through `curator_halt_metric` if it is not halted yet, and the proposer's bond goes to the insurance fund.

A halted metric takes no new rounds and no observations. The market accepts no opening or reduce-only orders and no margin withdrawals. Pending orders can be expired at once with `expire_order`, and their escrow returns to the owners' claim balances. The market then waits for its settlement price.

Settlement at the 24-hour TWAP

`settle_metric` sets the settlement price. Anyone can call it once the metric is halted. It reads the metric's `PriceHistory`, the ring of 168 hourly closes, and computes the time-weighted average of the index over the `settlement_window_seconds` before the halt, 86,400 seconds (24 hours). With T_h the halt time, W the window, $\tau_1 < \cdots < \tau_n$ the timestamps of the hourly closes inside the window, $\tau_0 = T_h - W$ and $\tau_{n+1} = T_h$:

$$I_{settle} = \left\lfloor \frac{1}{W} \sum_{k=0}^{n} I_k \left(\tau_{k+1} - \tau_k\right) \right\rfloor$$

I_0 is the latest close at or before the start of the window and I_k the close at τ_k, so the index is a step function that holds each close until the next one. Each hourly close is the last print of its 3,600-second bucket, timed at that print's finalization. Only finalized rounds write history, so failed, disputed and out-of-band rounds contribute nothing, and the last valid close holds through the rounds that failed before the halt. The program requires a close at or before the start of the window and refuses to settle without one (`InsufficientHistory`). The result is stored as the metric's `settlement_index` with `settled_at`, and `MetricSettled` records it.

For example, if the hourly closes held 60.000000000 for the first 18 hours of the window and 57.000000000 for the last 6, the settlement index is (60 × 18 + 57 × 6) / 24 = 59.250000000.

Settling the market and each position

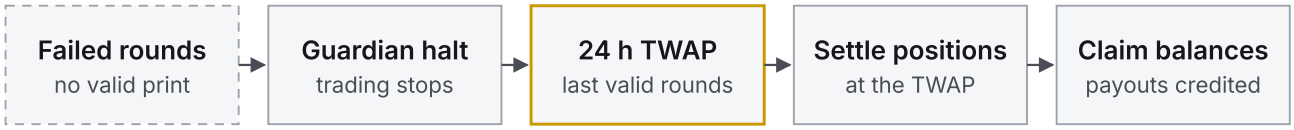
Three permissionless instructions take a halted market to its end, and the keeper runs them in order.

1. `halt_market` sets the market's status to `HALTED` once the metric is halted.
2. `settle_market` requires a settlement index, an empty order queue and a market synchronized to the metric's last print. It sets the market's index to `settlement_index`, sets its status to `SETTLED`, marks every open position unvalued and refreshes the market's PnL at the settlement index. Funding and borrow stop at the last valid print; nothing accrues after the halt.
3. `settle_position` closes one position in full at the settlement index. It charges no trading fee. It pays the keeper fee, collects the accrued borrow fee and splits it like every fee, and credits the owner's claim balance with

$$\text{payout} = \max\left(0,\ m + \text{PnL}(q, P_{entry}, I_{settle}) - c_F - c_B - k\right)$$

where c_F and c_B are the funding and borrow accrued to the last valid print and k the keeper fee. A position whose equity is negative receives nothing, and the insurance fund covers its shortfall into the vault as in a liquidation.

The vault completes no NAV generation while a settled market still holds open positions. Once the last one has settled, the market joins every later generation with no exposure.



Rounds fail until the threshold, the guardian halts the metric, the 24-hour TWAP sets the settlement index, every position settles at it, and payouts land in claim balances.

Protected claim balances

Payouts from closes, liquidations, settlements, refunds, withdrawals and bond returns go first to the receiving wallet's `OwnerState`, as `claimable_usdc` for USDC and `claimable_shares` for vault shares. The keepers that run fills, liquidations, settlements and withdrawals never transfer to an owner's token account, so no keeper depends on that account existing or accepting a transfer, and the owner chooses where the funds go.

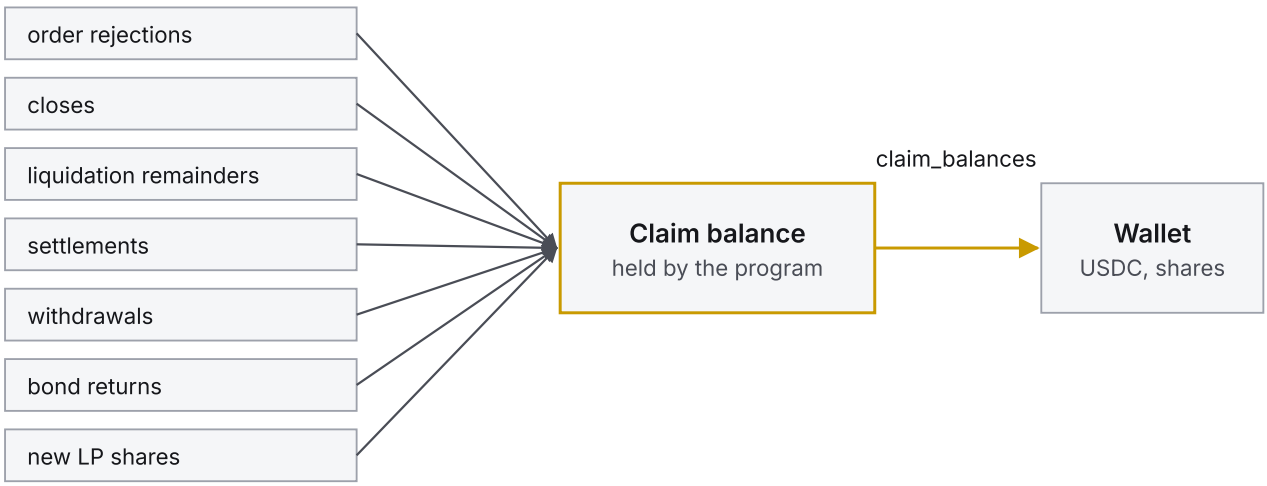
Credit	Instruction	Balance
Escrow of a cancelled, rejected or expired order	`cancel_order`, `expire_order`	USDC
Proceeds of a close	`execute_order`, reduce-only	USDC
Remaining margin after a liquidation	`liquidate`	USDC
Settlement payout	`settle_position`	USDC
Shares from a deposit	`execute_deposit`	shares
Refund of an unpriced or distressed deposit	`cancel_deposit`, `execute_deposit`	USDC
Proceeds of a withdrawal	`execute_withdraw`	USDC
Shares of a withdrawal at a distressed NAV	`price_withdraw`	shares

Credit	Instruction	Balance
Returned proposal bond	`return_bond`	USDC

The USDC behind claim balances stays in the vault's USDC account, and its total is the vault's `reserved_claims_usdc`. It is excluded from NAV, so LP shares never count it as theirs, and it is part of every reserve check. A close, a liquidation or a settlement credits a payout only if vault cash covers trader margin, pending deposits, the funding credit reserve and all existing claims with the new payout on top. Claimable shares sit in the share escrow; a wallet can claim them or queue them straight into `request_withdraw` without taking them out.

Claiming with `claim_balances`

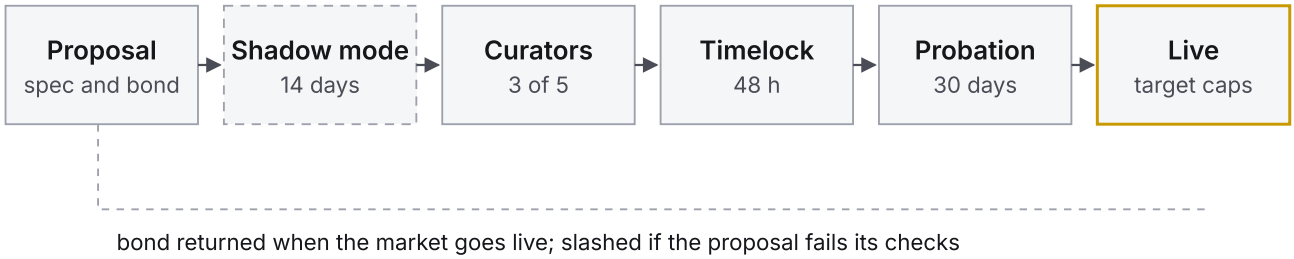
`claim_balances` takes a USDC amount and a share amount, at least one of them positive and neither above the owner's balance. The owner signs, and the instruction transfers the USDC from the vault's USDC account and the shares from the share escrow to token accounts the owner holds for those mints; any correctly owned account is accepted, not only the wallet's associated token account. Before it pays USDC, the instruction checks that the vault's cash covers trader margin, pending deposits and all claim balances, the claim included. The balances fall by the amounts paid, `reserved_claims_usdc` falls with them, and `BalancesClaimed` records the claim.



Order refunds, closes, liquidations, settlements, withdrawals and bond returns credit protected claim balances; claim_balances moves them to any token account the owner holds.

9. Listing a market

Every market enters Anyperp through one pipeline, and `anyperp_perps` enforces each step of it. A proposer describes a metric and posts a USDC bond. The metric runs in shadow mode for 14 days while both attestation roots report on it. Three of the five curators approve caps bound to the full configuration. A 48-hour timelock follows, then 30 days of probation at a quarter of the approved caps. A market that completes probation goes live and its bond returns to the proposer. A market that fails probation halts, settles, and its bond goes to the insurance fund. The program offers no other entry: `create_market` exists only to return `ListingRequired`.



A proposal and its bond enter shadow mode for 14 days; three curators approve, a 48-hour timelock runs, and the market trades 30 days of probation at 25 % of its caps before it goes live. The bond returns at live or rejection and is slashed to the insurance fund when probation fails.

The proposal

A proposal starts as a plain-language description of the market: which artist, and the artist's ID on the source. Sentinel turns the description into a draft with two parts, a metric spec and a cost dossier, built from research inputs the proposer supplies.

Research input	What it holds	Recorded on chain as
Source endpoint	the artist's public profile page, addressed by the opaque 22-character artist ID	part of `spec_hash`
Extraction rule	the exact integer monthly-listener count in the page's embedded artist data	part of `spec_hash`; the spec also carries the rule's own hash
Index divisor	listeners per index point: 1,000,000 at launch, one index point per million listeners	`index_divisor`
Reference level L	the artist's monthly listeners when the proposal is made	`reference_level`

Research input	What it holds	Recorded on chain as
c_{up}	USDC to add one listener that survives the source's filtering and holds through settlement	`cost_up_usdc_per_unit`
c_{down}	USDC to remove one real listener	`cost_down_usdc_per_unit`
Cited evidence	a title, a link and a note for each source behind the two costs	part of `dossier_hash`

Both costs are in USDC base units per listener. The spec fixes how the number is read and printed, from endpoint and extraction rule to the 900-second cadence, the 50 bps tolerance and each root's claim encoding. The dossier fixes what moving the number costs, with the target caps of chapter 3:

$$cap_{long} = \lfloor \kappa\, c_{up}\, L \rfloor, \qquad cap_{short} = \lfloor \kappa\, c_{down}\, L \rfloor$$

For an artist at 60,000,000 listeners with c_{up} = $0.02 and c_{down} = $0.06 per listener, an example, the target caps are $300,000 long and $900,000 short.

Sentinel computes every financial and spec field of the draft deterministically from these inputs; its model writes only the dossier's advisory limitations. A draft that lacks an input names it in `missingConfiguration`, and the API builds no `propose_market` transaction until that list is empty. Each draft suggests the 5,000 USDC minimum bond, is stored with the SHA-256 of its canonical spec and dossier, and is public.

Sentinel is advisory. It holds no key, signs no price, vote or transfer, and nothing it writes enters a round or a settlement. It also publishes an assessment for every market: flags such as a halted metric or recent failed rounds, and the external markets on the same number.

The bond: `propose_market`

The proposer signs one transaction carrying the metric arguments, the market parameters, c_{up}, c_{down}, L, the dossier hash, the SHA-256 of the description and the bond. The API fills every market parameter except the caps from protocol configuration.

The program checks the proposal against policy before it creates anything:

- the bond lies between `min_bond_usdc` and `max_bond_usdc`, 5,000 and 25,000 USDC at launch;
- the metric uses 900-second rounds, a root tolerance of at most 50 bps, a circuit band of at most 1,000 bps, a side pause of at least 21,600 seconds and at least two confirmation rounds;
- the maintenance margin is at least the exchange minimum and below the initial margin at maximum leverage;
- each target cap equals $\lfloor \kappa c L \rfloor$ exactly, recomputed on chain with `kappa_fixed9` = 250,000,000; any other value fails with `CapExceeded`;

- the root keys in the spec are the first keys of the two active attestor sets.

The same transaction then creates the listing:

1. The curator PDA signs a cross-program call to `initialize_metric` in `anyperp_oracle`, which creates the `MetricSpec` at `["metric", metric_id]` and its `PriceHistory`. The metric ID is the SHA-256 of the provider and the subject ID, so each artist has one metric; the `Market` derives from the metric, so each metric has one market.
2. The bond moves from the proposer's USDC account into a token account at `["bond", proposal]` whose authority is the proposal PDA. Only the program's return and slash paths sign for it.
3. The `Proposal` at `["proposal", proposer, nonce]` records the hashes, costs, level, target caps, bond and `shadow_started_at`.
4. The `Market` at `["market", metric]` starts in `SHADOW` with `market_index` = 255, outside the vault's valuation.

Shadow mode

From that transaction on, the metric runs like any listed metric. Reporters open a round every 900 seconds, both roots observe the source, keepers finalize, and prints enter the price history. The market takes no orders, since `open_order` accepts only `PROBATION` and `LIVE` markets.

The oracle keeps the score. Every round that reaches a terminal state adds one to `evaluated_rounds`, exactly once. It adds one to `agreed_rounds` when both roots reported values within tolerance, fetched inside the round's window. Failed and disputed rounds count against the market. Shadow mode is complete when three conditions hold together:

- 1,209,600 seconds (14 days) have passed since `shadow_started_at`;
- at least `shadow_seconds / cadence_seconds` = 1,344 rounds are evaluated;
- $agreed \cdot 10{,}000 \geq evaluated \cdot 9{,}900$, so the roots agree on at least 99 % of rounds.

The gate counts rounds as well as time, so a source that missed rounds takes longer to qualify, and a halted metric never qualifies. The listings page in the app shows the running agreement as shadow rounds agreeing.

Curator approval

Five member keys are registered in `CuratorConfig`, and any three decide. Each curator votes from their own wallet with `approve_market(cap_long, cap_short, configuration_hash)` or `reject_market(reason_hash)`. A vote sets the member's bit in the proposal's `approve_mask` or `reject_mask`, and each member votes once across both.

An approval is bound to the full configuration. The program rebuilds the proposal's arguments from the immutable fields of the proposal, the market and the metric, and hashes them with the policy in

force:

$$h = \mathrm{SHA256}\left(D \parallel P \parallel A \parallel \mathrm{ExchangeParams} \parallel \mathrm{ProposeMarketArgs} \parallel cap_{long} \parallel cap_{short}\right)$$

Here D is the domain `ANYPERP_LISTING_APPROVAL_V1`, P the two program IDs, A the exchange, proposal, proposer, market and metric addresses, and both structs are Borsh-encoded. A vote with any other hash fails with `InconsistentVote`, so a vote approves exactly one bond, spec, pair of root keys, set of market parameters, dossier, description and exchange policy. The API computes the hash from confirmed accounts and places it in the curator's transaction.

At every vote the program re-checks the policy bounds and the shadow gate. The caps must be positive and at most the target caps. The first approval fixes them, and the second and third must name the same pair, so curators can approve the targets or any lower pair. At the third approval the proposal moves to `TIMELOCK` with `approved_at` and `timelock_ends_at` = `approved_at` + 172,800 seconds. Three rejections naming the same reason hash move it to `REJECTED`.

The 48-hour timelock and activation

For 48 hours the approved configuration stands in public and nothing trades. Then anyone can call `activate_market`, and the keeper does so as soon as the vault allows. Activation requires a metric that is not halted, fewer than 24 registered markets, a complete current NAV generation, and no deposit or withdrawal waiting on that generation.

Activation registers the market in the vault. The market takes the next `market_index`, `market_count` rises, and a fresh valuation generation opens that must include the new market before the next NAV exists (chapter 7). The program sets each cap to `probation_multiplier_bps` = 2,500 of the approved cap, rounded down, marks the market and the proposal `PROBATION`, starts the 2,592,000-second (30-day) probation, and snapshots the metric's evaluated and agreed counts so probation is scored on its own rounds.

Probation

During probation the market trades at its reduced caps, $75,000 long and $225,000 short in the example. Fees, funding, borrow and the proposer's fee share all run. Objective on-chain state decides the outcome, and any keeper executes it:

- promotion, once probation has ended with full coverage and 99 % agreement;
- slashing, once the metric has halted, the market has become insolvent, or probation has ended with full coverage and agreement below 99 %;
- continued probation, while probation has ended but fewer than 2,880 rounds are evaluated; the market keeps trading at probation caps until coverage is complete.

Promotion to live

`promote_market` requires `PROBATION`, the end of probation, a metric that is not halted, a solvent market, and at least 2,880 evaluated probation rounds with the roots agreeing on at least 99 % of them. It sets the caps to the approved caps and marks the market and the proposal `LIVE`.

Bond return

`return_bond` runs for a proposal that is `LIVE` or `REJECTED` and still holds its bond. The proposal PDA signs the transfer from the bond account into the vault's USDC account, and the program credits the full amount to the proposer's claim balance, `claimable_usdc` in `OwnerState`, reserved in the vault's `reserved_claims_usdc` and outside LP NAV. The proposer moves it to a wallet with `claim_balances`. The instruction has no signer requirement, and the keeper sends it.

Slashing

`slash_bond` runs in probation only, and only when one of three conditions holds: the oracle metric is halted; the market is insolvent, its bad debt having exceeded the insurance fund; or probation ended with full coverage and agreement below 99 %. On a healthy probation it fails with `ProbationHealthy`. If the metric is still running, the slash halts it through `curator_halt_metric`, signed by the curator PDA. The whole bond goes to the insurance fund's token account, the proposal becomes `SLASHED`, and the market becomes `HALTED` and settles at the TWAP of the 24 hours before the halt (chapter 8). The insurance fund holds the slashed bond with the rest of its USDC and pays it out against negative equity when positions settle.

Once a proposal is `LIVE`, `REJECTED` or `SLASHED` with no bond left, the proposer can close it and its empty bond account with `close_proposal` and recover the rent. The indexer keeps the closed proposal as a read-only receipt.

The proposer's fee share

The proposer earns `proposer_fee_bps` = 1,000, or 10 %, of every fee the market collects: trading fees on opening and closing fills, liquidation fees and the collected borrow fee. At each fee the program moves the proposer's share into the proposer fee account, a USDC token account owned by the vault PDA, and adds it to `claimable_proposer_usdc` on the market. For a 10 bps fee on a 10,000 USDC fill, an example, the fee is 10 USDC and 1 USDC of it is the proposer's. The proposer collects with `claim_market_fees(amount)`, signed by the market's recorded proposer, into a USDC account the proposer owns. The share accrues from the first fill in probation for the life of the market, and it is independent of the bond and of the proposal account.

The launch cohort

The 20 launch markets enter through `bootstrap_launch_market`, a genesis path that closes when the programs pass to governance. Its signer must be the exchange admin and also the current upgrade authority of `anyperp_perps`, and it runs only before the vault's first NAV generation, while the vault holds no margin, deposits or withdrawals. Each launch market passes the checks of `propose_market` with a zero bond, is registered in the vault at once and starts `LIVE` at its target caps.

Protocol configuration sets the launch cohort at c_{up} = 0.001 USDC and c_{down} = 0.003 USDC per listener with a reference level of 10,000,000 listeners per artist, which gives caps of 2,500 USDC long and 7,500 USDC short. Once the upgrade authority passes to the curator PDA (chapter 12), no signer meets the entry check, and every later market takes the full pipeline.

Listing states and transitions

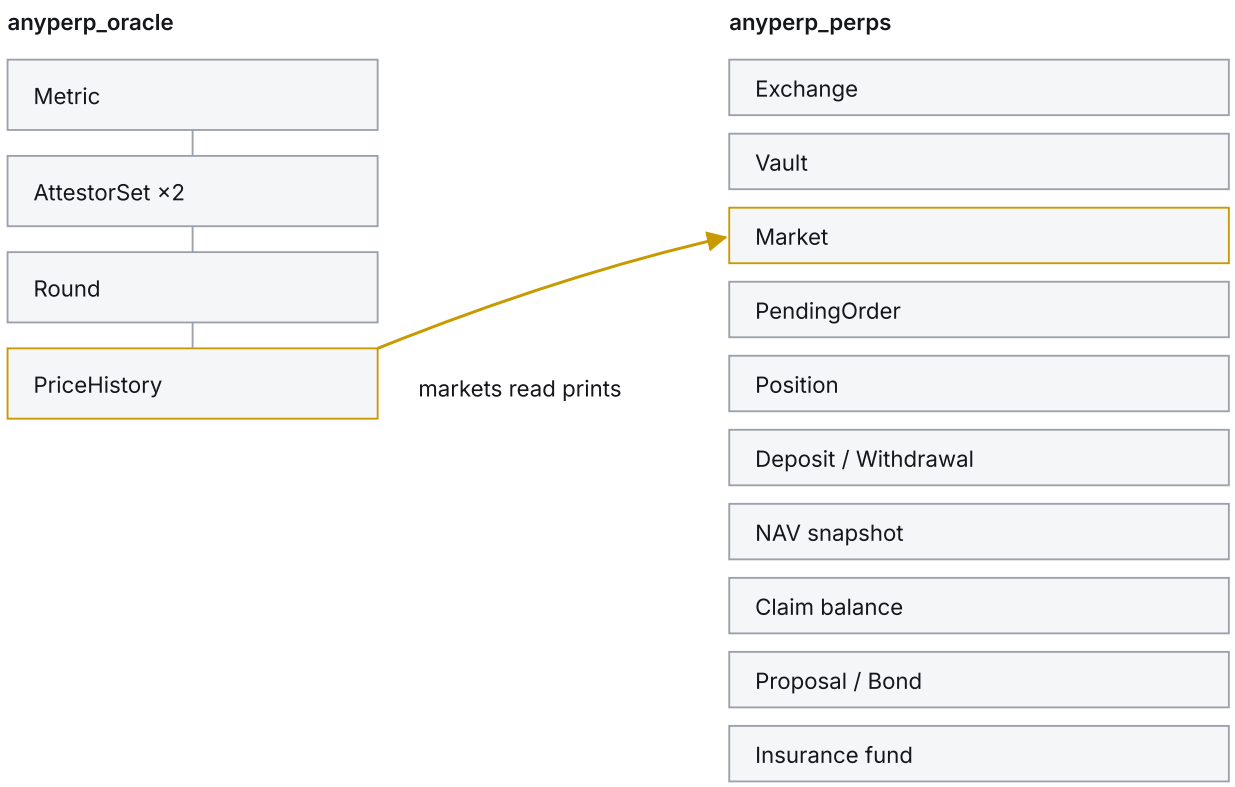
`Proposal.status` and `Market.status` share one set of codes. The market keeps `SHADOW` while its proposal is in `TIMELOCK` or `REJECTED`; activation and promotion set both; a slash sets the proposal to `SLASHED` and the market to `HALTED`.

State (code)	Entered by	Signer	Orders	Left by
`SHADOW` (0)	`propose_market`	proposer	none; rounds run and are scored	three `approve_market` votes to `TIMELOCK`, or three `reject_market` votes to `REJECTED`
`TIMELOCK` (1)	the third `approve_market`	curator	none	`activate_market` once 172,800 s have passed
`PROBATION` (2)	`activate_market`	any	opens up to 25 % of approved caps; closes	`promote_market` to `LIVE`, or `slash_bond` to `SLASHED`
`LIVE` (3)	`promote_market`, or `bootstrap_launch_market` for the launch cohort	any; admin for the launch cohort	opens up to approved caps; closes	a halt; `return_bond` returns the bond without changing state
`REJECTED` (4)	the third `reject_market`	curator	none	`return_bond`, then `close_proposal`
`SLASHED` (5)	`slash_bond`	any	none; the market is halted	`close_proposal`
`HALTED` (6)	`halt_market` after an oracle halt, or `slash_bond`	any	none	`settle_market` once the metric's settlement price is recorded

State (code)	Entered by	Signer	Orders	Left by
`SETTLED` (7)	`settle_market`	any	none; positions close through `settle_position`	final

10. Programs and accounts

Anyperp runs as two Anchor programs on Solana. `anyperp_oracle` holds the metrics and their prints. `anyperp_perps` holds the money: the exchange, the vault, markets, orders, positions, the LP queues, NAV snapshots, claim balances, listings and governance. Both build on `anyperp_domain`, a shared crate of exact integer arithmetic with no floating point, which the off-chain services check against the same test vectors. Every account the two programs own is a PDA with fixed seeds, so any client derives its address from public inputs. Account sizes below are in bytes and include Anchor's 8-byte discriminator.



The two programs and their accounts. The perps program reads MetricSpec and OracleRound as typed oracle accounts and writes to the oracle only through calls signed by its curator PDA.

The two programs

`anyperp_oracle` owns five account types: one `OracleConfig`, two `AttestorSet`s, and for every metric a `MetricSpec`, a `PriceHistory` and one `OracleRound` per round. It holds no tokens. It verifies each observation against a signature precompile in the same transaction, secp256k1 for root 1 and Ed25519 for root 2, and writes prices only from rounds in which both roots agree.

`anyperp_perps` owns every account that moves value. Its token accounts hold USDC and LP shares under the authority of the vault PDA or of a proposal PDA, so only program code signs a transfer out

of them.

The oracle names the perps program in one place: `initialize_oracle_config` requires `curator_authority` to be the PDA `["curator"]` of `anyperp_perps`. Oracle governance is therefore the perps program's curator quorum, and both programs answer to one set of five curators.

How the perps program reads oracle state

The perps program links the oracle crate and takes `MetricSpec` and `OracleRound` as typed Anchor accounts. Anchor checks that each is owned by `anyperp_oracle` and carries the right discriminator, so an account written by any other program fails to load. Constraints then tie both to the market: the `Market` names its metric (`has_one = metric`), and every round must name the same metric or the instruction fails with `WrongOracleMetric`.

From the round, the perps program reads the print. `sync` accepts a round only if it is `FINALIZED` with `final_index` > 0 and a sequence at or above the market's `last_sequence`, and copies `final_index`, `finalized_at` and `min_fetched_at` into the market. The round's two fetch-time bounds decide which orders it serves: an order that landed at time t fills only at the round with `previous_finalized_min_fetched_at` $\leq t <$ `min_fetched_at`, the first print whose observations were all fetched after the order landed (chapter 5).

From the metric, it reads the gates: the halt and pause fields for every opening order, `latest_finalized_sequence` for liquidations and margin removal, the sequence counters for cancellation, `evaluated_rounds` and `agreed_rounds` for listing, and `settled_at` and `settlement_index` for settlement. A listing vote counts only if the metric's `authority` is the exchange's curator PDA.

The perps program writes to the oracle only through cross-program calls signed by the curator PDA:

Call	From	Purpose
`initialize_metric`	`propose_market`, `bootstrap_launch_market`	create the metric and its price history
`curator_halt_metric`	`slash_bond`	halt the metric of a failed probation
`schedule_attestor_rotation`, `cancel_attestor_rotation`	`approve_oracle_policy`	queue or cancel a root's key rotation
`curator_pause_metric`	`approve_oracle_policy`	set or clear a pause and close a guardian episode

No perps instruction writes a price.

Oracle accounts

`OracleConfig` holds the roles and the windows every metric inherits: the admin, the curator authority, the guardian, the attestor timelock (172,800 seconds), the guardian pause limit (86,400 seconds), the hourly history bucket and the 86,400-second reference and settlement windows. Outside local test configuration, `initialize_oracle_config` requires an attestor timelock of at least 172,800 seconds and exactly these pause, bucket and window lengths.

Each `AttestorSet` lists the keys of one root: up to 8 Ethereum addresses of Reclaim attestors for root 1, padded with 12 zero bytes, and up to 8 Ed25519 keys of the independent notary for root 2. It keeps three generations side by side. The current keys verify observations fetched inside their window; the previous keys verify only observations fetched before the last activation; the pending keys wait for their activation time. `generation` counts activations.

A `MetricSpec` holds one artist's metric: the immutable spec fixed at listing, the live state of its rounds, pauses and band confirmation, the shadow and probation score, and the halt and settlement fields. An `OracleRound` exists per metric and sequence, paid for by whoever opens it, with two observation slots holding each root's value, times, signer and 65-byte signature. `PriceHistory` is a fixed ring of 168 hourly closes of 16 bytes each; the last print in an hour replaces that hour's close, the circuit band reads its reference from it and settlement its 24-hour TWAP. Its length never changes, so the account never reallocates.

Perps accounts

`Exchange` holds the admin, the curator PDA, the guardian, the protocol treasury, the USDC mint (6 decimals, checked), the three fee accounts and `ExchangeParams`, the protocol policy listed in chapter 14. `initialize_exchange` fixes the fee split at 6,000/2,000/1,000/1,000 bps, bounds leverage between 1× and 3×, requires at least two rounds before cancellation, bounds each LP queue at 1 to 64 entries, enforces the production minimums for the withdrawal queue, shadow mode, timelock, probation and bonds, and requires the guardian and curator authority to match the oracle's configuration.

`CuratorConfig` at `["curator"]` lists the five curator members. Its PDA is the governance signer of the system: the oracle's curator authority, the authority of every `MetricSpec`, and, after handover, the upgrade authority of both programs.

`Vault` at `["vault"]` backs every market. It carries the reserves of the balance sheet (trader margin, pending deposits, priced withdrawals, claim balances, funding credits and exposure), the trader PnL and charge books, bad debt and fees earned, and the heads and tails of the deposit and withdrawal queues and the valuation state of the current NAV generation, with one fetch-time slot for each of up to 24 markets. Four token accounts sit under its authority: USDC at `["usdc"]`, the share mint at `["shares"]` with 9 decimals, the share escrow at `["share_escrow"]`, which holds shares owed to claim balances and shares in the withdrawal queue, and `["dead_shares"]`.

`initialize_vault` mints the seed's shares, 1,000 share units per USDC base unit, into the dead-share account, where they stay. At the launch seed of 1,000 USDC, one share is worth 1 USDC.

The three fee accounts are USDC token accounts whose addresses `Exchange` records: the protocol fee account, owned by the protocol treasury, and the insurance fund and the proposer fee account, both owned by the vault PDA. Every fee transfer is constrained to these addresses. The insurance fund receives 10 % of fees and every slashed bond, and covers negative equity when a position settles; what it cannot cover is booked as bad debt and marks the market insolvent.

A `Market` at `["market", metric]` holds one metric's book and its order queue. Each `PendingOrder` at `["order", owner, nonce]` links to the next order in that queue and records its landing time, round and outcome. A `Position` at `["position", market, owner]` is one wallet's isolated position in one market, locked while an order on it is pending.

`OwnerState` at `["owner", owner]` is each wallet's ledger with the program. It keeps the last nonce, the claim balances in USDC and in shares, and pointers to the wallet's one pending deposit and one pending withdrawal. Every account a wallet creates with a nonce takes one strictly above `last_nonce`, so the address of a closed order, deposit, withdrawal or proposal is never created again (`NonceReplay`). Claim balances are backed one to one: USDC by `reserved_claims_usdc` in the vault, shares by the share escrow.

A `PendingDeposit` at `["deposit", owner, nonce]` and a `Withdrawal` at `["withdraw", owner, nonce]` each link to the next entry of their queue; a withdrawal keeps its fixed USDC amount and NAV epoch once priced.

A `NavSnapshot` at `["nav", epoch]` freezes one NAV generation: NAV, share supply, the earliest fetch time across its prints, the previous generation's bound and a distressed flag. Once `completed_at` is set, nothing changes it. The `NavMintAllocation` at `["nav_mint", epoch]` counts shares already issued against that snapshot, so deposits priced at one epoch take their shares from a fixed quote in queue order, independent of later supply. The keeper whose transaction created either account pays its rent and receives it back when the account is closed.

A `Proposal` at `["proposal", proposer, nonce]` records a listing from bond to outcome, with its bond held at `["bond", proposal]` under the proposal PDA's authority (chapter 9). An `OraclePolicyApproval` at `["oracle_action", action_hash]` collects curator votes on one rotation or pause. An `UpgradeProposal` at `["upgrade", proposal_hash]` collects votes on one program upgrade and holds its timelock (chapter 12).

Account reference

Account	Seeds	Program	Purpose	Key fields	Bytes
`OracleConfig`	`"oracle_config"`	oracle	roles and windows	`admin`, `curator_authority`, `guardian`, `attestor_timelock_seconds`, `guardian_pause_seconds`, `history_bucket_seconds`, `reference_window_seconds`, `settlement_window_seconds`	126
`AttestorSet`	`"attestors"`, root (0 or 1)	oracle	keys of one root	`keys`, `valid_from`, `valid_until`, `previous_keys` and window, `pending_keys`, `pending_activation_at`, `generation`	886
`MetricSpec`	`"metric"`, metric ID	oracle	one artist's metric	`spec_hash`, `index_divisor`, `cadence_seconds`, `tolerance_bps`, `band_bps`, `root1_eth_address`, `root2_ed25519`, `next_sequence`, `latest_finalized_sequence`, `latest_finalized_index`, pauses, `evaluated_rounds`, `agreed_rounds`, `consecutive_failed_rounds`, `halted`, `settlement_index`	383
`OracleRound`	`"round"`, metric, sequence (u64 LE)	oracle	one round	`opened_at`, `closes_at`, `observations[2]`, `status`, `final_index`, `finalized_at`, `min_fetched_at`, `previous_finalized_min_fetched_at`	359
`PriceHistory`	`"history"`, metric	oracle	168 hourly closes	`entries`, `head`, `len`, `bucket_seconds`	2,741
`Exchange`	`"exchange"`	perps	roles and policy	`admin`, `curator`, `guardian`, `protocol_treasury`, `usdc_mint`, `protocol_token`, `insurance_token`, `proposer_token`, `params`	376
`CuratorConfig`	`"curator"`	perps	curator registry and governance signer	`exchange`, `members[5]`	201
`Vault`	`"vault"`	perps	shared counterparty and LP book	reserves, `total_pnl_usdc`, `total_charges_usdc`, `bad_debt_usdc`, `fees_earned_usdc`, queue heads and tails, `market_count`, `valuation_sequence`, `valuation_count`, `market_fetch_times[24]`	617
vault USDC	`"usdc"`	token program	vault cash	authority: vault PDA	165
share mint	`"shares"`	token program	LP shares, 9 decimals	mint authority: vault PDA	—

Account	Seeds	Program	Purpose	Key fields	Bytes
share escrow	`"share_escrow"`	token program	claimable and queued shares	authority: vault PDA	165
dead shares	`"dead_shares"`	token program	the seed's locked shares	authority: vault PDA	165
fee accounts	recorded in `Exchange`	token program	protocol fees, insurance fund, proposer fees	owners: treasury, vault PDA, vault PDA	165
`Market`	`"market"`, metric	perps	one metric's book	`status`, `index`, `long_size`, `short_size`, `params`, `funding_rate`, `funding_index`, `borrow_index`, `queue_head`, `queue_tail`, `barrier_sequence`, `insolvent`, `claimable_proposer_usdc`, `market_index`	586
`PendingOrder`	`"order"`, owner, nonce	perps	one order	`long`, `reduce_only`, `size`, `margin_usdc`, `acceptable_price`, `landed_at`, `state`, `fill_price`, `next`	197
`Position`	`"position"`, market, owner	perps	one isolated position	`long`, `size`, `entry_price`, `margin_usdc`, `funding_checkpoint`, `borrow_checkpoint`, `state`, `valued_sequence`, `has_pending_order`	196
`OwnerState`	`"owner"`, owner	perps	nonces and claim balances	`last_nonce`, `claimable_usdc`, `claimable_shares`, `pending_deposit`, `pending_withdrawal`	129
`PendingDeposit`	`"deposit"`, owner, nonce	perps	one deposit	`amount_usdc`, `landed_at`, `state`, `shares`, `nav_epoch`, `next`	154
`Withdrawal`	`"withdraw"`, owner, nonce	perps	one withdrawal	`shares`, `requested_at`, `unlocks_at`, `amount_usdc`, `nav_epoch`, `state`, `next`	162
`NavSnapshot`	`"nav"`, epoch	perps	one frozen NAV	`nav_usdc`, `share_supply`, `min_fetched_at`, `previous_min_fetched_at`, `completed_at`, `distressed`, `rent_payer`	122
`NavMintAllocation`	`"nav_mint"`, epoch	perps	shares issued at one NAV	`allocated_shares`, `rent_payer`	90
`Proposal`	`"proposal"`, proposer, nonce	perps	one listing	`bond_remaining_usdc`, `spec_hash`, `dossier_hash`, `cost_up_usdc_per_unit`, `cost_down_usdc_per_unit`, `reference_level`, target and approved caps, `status`, `approve_mask`, timelock and probation times	372
bond account	`"bond"`, proposal	token program	the proposal's bond	authority: proposal PDA	165

Account	Seeds	Program	Purpose	Key fields	Bytes
`OraclePolicyApproval`	`"oracle_action"`, action hash	perps	one oracle policy vote	`action_hash`, `nonce`, `approval_mask`, `executed`	99
`UpgradeProposal`	`"upgrade"`, proposal hash	perps	one upgrade vote	`program`, `buffer`, `spill`, `elf_hash`, `approval_mask`, `execute_after`, `executed`	211

The address lookup table

Transactions that touch many accounts compile against one address lookup table, whose authority is the keeper key. At initialization it receives the fixed addresses: `Exchange`, `Vault`, the vault's USDC account, share mint and share escrow, the USDC mint, the three fee accounts, the curator PDA, `OracleConfig`, both `AttestorSet`s, both program IDs, the token program, the system program and the rent sysvar. For every market it adds the `Market`, `MetricSpec` and `PriceHistory`, and the keeper extends it with each new market's three addresses, 20 per extension. The API builds a legacy message when it fits Solana's 1,232-byte packet and a v0 message with the table when it does not, or when the action is known to be large, such as `propose_market` and `settle_position`. No prepared transaction exceeds 1,232 bytes.

Oracle instructions

Instruction	Signer	Effect
`initialize_oracle_config`	admin, the program's upgrade authority	creates `OracleConfig` and both `AttestorSet`s; rejects production values below the minimums
`schedule_attestor_rotation`	curator PDA	queues a root's next keys, active no earlier than 172,800 s later; one pending rotation per root
`cancel_attestor_rotation`	curator PDA	clears a queued rotation
`activate_attestor_rotation`	any	after the activation time, makes the pending keys current and bounds the previous keys' window at activation
`initialize_metric`	curator PDA or admin, with a payer	creates `MetricSpec` and `PriceHistory`; checks cadence, tolerance, band, pause, confirmation and root keys
`open_round`	payer, any	creates the next `OracleRound` once a cadence has passed since the last opening

Instruction	Signer	Effect
`submit_reclaim_observation`	reporter, any	rebuilds the Reclaim claim for the artist, round and value, verifies the attestor's secp256k1 signature through the preceding precompile instruction and the root-1 set, and records the observation
`submit_notary_observation`	reporter, any	rebuilds the notary's claim, verifies its Ed25519 signature through the preceding precompile instruction and the root-2 set, and records the observation
`submit_observation`	reporter, any	the same for locally generated test attestors; accepted only by a local test configuration
`finalize_round`	any	prints the median when both roots agree, the round is in order and the band holds; otherwise disputes or holds the round out of band
`mark_failed_round`	any	after a round closes with a root missing, marks it failed and pauses both sides for 21,600 s
`guardian_pause_metric`	guardian	pauses opens up to 86,400 s from the start of the episode
`curator_pause_metric`	curator PDA	sets or clears a pause and closes the guardian episode
`halt_metric`	guardian or curator PDA	halts after `failed_rounds_before_halt` consecutive rounds without a print
`curator_halt_metric`	curator PDA	halts the metric of a slashed probation
`settle_metric`	any	records the settlement index, the TWAP of the 86,400 s before the halt, once
`settlement_twap`	any	returns that TWAP without writing

Perps instructions

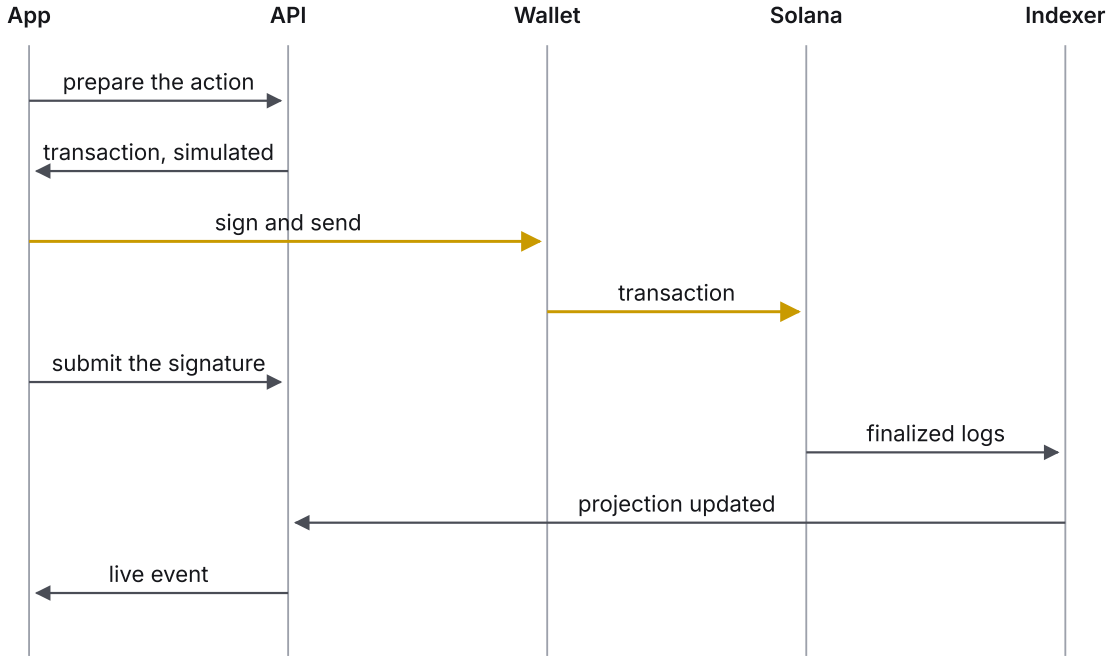
Instruction	Signer	Effect
`initialize_exchange`	admin, the program's upgrade authority	creates `Exchange` and `CuratorConfig`; validates policy, fee accounts and the oracle link
`initialize_vault`	admin	creates the vault and its token accounts; locks the seed's shares
`set_local_withdrawal_queue`	admin	changes the withdrawal queue on local test exchanges only
`create_market`	admin	always fails with `ListingRequired`

Instruction	Signer	Effect
`tag_transaction`	owner	binds a transaction to one API intent; changes nothing
`propose_market`	proposer	creates proposal, market, metric and history; takes the bond
`bootstrap_launch_market`	admin, the program's upgrade authority	lists a launch market live with no bond, before the first NAV
`approve_market`	curator member	votes caps under the configuration hash; the third vote starts the timelock
`reject_market`	curator member	votes a reason hash; the third vote rejects
`activate_market`	any	registers the market in the vault at probation caps
`promote_market`	any	sets approved caps and `LIVE` after a passing probation
`return_bond`	any	moves the bond to the proposer's claim balance
`slash_bond`	any	moves the bond to the insurance fund and halts the market
`claim_market_fees`	market proposer	pays accrued proposer fees to the proposer's USDC account
`close_proposal`	proposer	closes a finished proposal and its bond account
`approve_oracle_policy`	curator member	votes a rotation or pause; the third vote executes it
`approve_program_upgrade`	curator member	votes an upgrade; the third vote starts the 259,200 s timelock
`execute_program_upgrade`	any	upgrades a program from the voted buffer after the timelock
`open_order`	owner	escrows margin and queues an order
`cancel_order`	owner	returns the margin of an order that can no longer fill to the claim balance
`expire_order`	keeper	does the same after 86,400 s or a halt
`execute_order`	keeper	fills or rejects the queue head at its first eligible print
`modify_margin`	owner	adds margin, or removes it within the initial margin at the latest print

Instruction	Signer	Effect
`liquidate`	keeper	closes a position below maintenance at the index
`sync_market`	keeper	moves a market to a finalized print and accrues funding and borrow
`value_position`	keeper	confirms a position's equity at the market's print for the NAV generation
`deposit`	owner	queues USDC for the next eligible NAV
`cancel_deposit`	owner	refunds a deposit to the claim balance after 259,200 s with no eligible NAV
`execute_deposit`	keeper	issues shares at the deposit's NAV into the claim balance
`request_withdraw`	owner	queues shares for 259,200 s
`price_withdraw`	any	fixes the USDC amount at the first NAV after unlock and burns the shares
`execute_withdraw`	any	credits a priced withdrawal to the claim balance
`claim_balances`	owner	pays claim balances to token accounts the wallet owns
`recapitalize_vault`	donor	adds USDC to the vault
`halt_market`	any	marks the market halted after its metric halts
`settle_market`	any	sets the index to the settlement price and marks the market settled
`settle_position`	keeper	closes a position at the settlement index into the claim balance
`close_order`, `close_position`, `close_deposit`, `close_withdrawal`	owner	return the rent of a finished account
`close_nav_snapshot`, `close_nav_mint_allocation`	any	return the rent of a past NAV account to its payer

11. Transaction flows

This chapter follows each user flow through its instructions: who signs, which instruction runs, which accounts change and what the app shows. Every flow a user starts takes the same path through the API, the wallet and the indexer. Keepers and reporters sign their own transactions against the same programs and the same checks.



The app asks the API to prepare a transaction; the wallet signs and sends it; the API confirms that the included message is the one it prepared; the indexer projects the finalized result back to the app.

The common path

1. The app calls `prepareTransaction` with one action and an idempotency key. The API checks the action against indexed state, reserves a nonce when the action creates an owner-keyed account, and assembles a compute limit of 1,000,000 units, `tag_transaction` with the SHA-256 of the owner and the idempotency key, any token-account creation the action needs, and the action itself. The wallet is the fee payer and the only signer.
2. The wallet shows the transaction, signs it and sends it.
3. The app reports the signature with `submitTransaction`, and the API accepts it only if the included message hashes to the prepared one.
4. The indexer projects the finalized accounts and events, and live events refresh the app: `transaction.updated`, `order.filled`, `round.finalized`, `vault.updated` and `market.state_changed`.

Chapter 13 describes the API side of each step.

Sign in

The app reads the current terms version from `eligibility` and calls `authChallenge` with the wallet and its origin. The wallet signs the challenge message, which states that the signature authorizes no transaction. `authVerify` checks the Ed25519 signature, the origin, the expiry and the eligibility attestation, consumes the challenge once and sets the `anyperp_session` cookie. Nothing changes on chain. The app shows the connected wallet and opens the portfolio.

Open an order and its fill

Quote. `orderQuote` returns the indicative fill price, skew impact, fees, required margin, indicative liquidation price and cap headroom. The ticket shows them next to the rule that the order fills at the first print fetched after it lands.

Landing. The owner signs `open_order(nonce, long, reduce_only = false, size, margin, acceptable_price)`.

- `OwnerState` is created on first use, and `last_nonce` advances.
- The `PendingOrder` at `["order", owner, nonce]` is created with its landing time, slot and round, and joins the end of the market's queue: the previous tail's `next`, or the market's `queue_head`, points at it, and `queue_tail` and `pending_count` update.
- The `Position` at `["position", market, owner]` is created on first use and locked by `has_pending_order`.
- The margin moves from the owner's USDC account into the vault, and `Vault.reserved_margin_usdc` rises by it.

The program refuses the order if the market is not in probation or live, is insolvent, or is halted or paused on that side, or if the wallet holds the other side. The app lists the order with the print it will fill at.

Print, sync and valuation. The first finalized round whose observations were all fetched after `landed_at` is the order's print. A keeper signs `sync_market(epoch)` with it: the market moves to the print's index, funding and borrow accrue, the queue records its barrier, and the vault's PnL, charge book and reserves update. The keeper then signs `liquidate` for any position below maintenance and `value_position` for every other open position, so the whole book is valued at the print before new exposure enters.

Fill. The keeper signs `execute_order(epoch)` for the queue head with the same round. The program prices the fill against skew and checks the acceptable price, the side cap and the margin. It writes the `Position` (side, size, entry price, margin net of the trading and keeper fees, checkpoints, `ACTIVE`), the `Market` (side size and entry basis, charge bases, `total_positions`, `claimable_proposer_usdc`, queue head and barrier), the `Vault` (reserved margin, fees earned, PnL, reserves) and the `PendingOrder` (`FILLED`, `fill_price`, `fill_sequence`). Of the fee, 20 % goes to

the protocol fee account, 10 % to the insurance fund and 10 % to the proposer fee account, and the remainder stays in the vault; the keeper fee goes to the keeper's USDC account. The app receives `order.filled` and shows the position with size, notional, entry, mark, funding, borrow, margin, equity and liquidation price.

Rejection and cancellation. When the fill breaks a rule at its print, `execute_order` marks the order rejected at that sequence, pops it from the queue and unlocks the position. The owner signs `cancel_order` once the market has moved past that print, or a keeper signs `expire_order` after 86,400 seconds, and the margin moves from reserved margin to the owner's claim balance. `cancel_order` also releases a pending order that no print can fill: two rounds after the one open at landing have ended with no finalized print fetched after `landed_at`, and no round is open. A halted metric releases it at once.

Add to a position

The owner signs `open_order` again on the same side with the added size and new margin, and it fills at its own print like any order. The fill merges it into the position: size grows, the entry price becomes the size-weighted entry rounded against the trader, the checkpoints are rebased so accrued charges stay owed, and the margin becomes the old margin plus the new margin less fees. The combined position must meet the initial margin at both its entry price and the current index, and the side cap counts the whole position. The quote prices fees on the increase and states margin and liquidation price for the combined position, which the app updates in place.

Modify margin

Margin changes take effect at once. The owner signs `modify_margin(add, amount, epoch)`.

- **Add.** USDC moves from the owner's account into the vault, and `Position.margin_usdc` and `Vault.reserved_margin_usdc` rise.
- **Remove.** The transaction names the metric's latest finalized round. The program projects equity at that print with funding and borrow accrued, without opening a NAV generation, and requires the remaining margin to cover the initial margin at that mark. It refuses on a halted or settled market and while an earlier print still holds queued orders. USDC moves from the vault to the owner's token account, and both margin fields fall.

Close

A close is an order. The app's close action prepares `open_order` with `reduce_only = true`, the position's side, a size up to the position's size and no margin, and it fills at its print. The closing equity is the position's margin share for the closed size plus PnL at the fill price, less funding and borrow on that size. The trading and keeper fees come out of it, the collected borrow fee is split like

any fee, and the rest is credited to `OwnerState.claimable_usdc` and reserved in `Vault.reserved_claims_usdc`. The market's side size, entry basis and charge bases fall; at zero size the position is closed and `total_positions` falls. The app shows the payout as ready to claim, and the owner can recover the rent of the closed position and the filled order with `close_position` and `close_order`.

Liquidation

A keeper computes each position's equity at its market's mark and signs `liquidate(epoch)` for one below maintenance, with the metric's latest finalized round or the barrier round while orders wait on it. The program syncs the market and checks equity < maintenance margin, failing with `NotLiquidatable` otherwise. The position closes at the index. From its equity, floored at zero, the keeper fee is paid first, then the 100 bps liquidation fee, split like any fee, and the rest goes to the owner's claim balance. Negative equity is covered from the insurance fund, and any shortfall beyond it is booked as bad debt and marks the market insolvent. The position becomes liquidated and leaves the app's positions list; the history records it.

Deposit and its fill at NAV

1. The LP signs `deposit(nonce, amount)`. The USDC moves into the vault and is held outside NAV in `pending_deposits_usdc`, and a `PendingDeposit` joins the end of the deposit queue. Each wallet has one pending deposit, and the queue holds up to 32. The app states that the deposit fills at the next round's NAV.
2. Keepers build the next NAV generation with `sync_market` for each market and `value_position` for each open position. When every registered market has contributed, the vault writes the generation's `NavSnapshot`: NAV, share supply and the earliest fetch time across its prints.
3. A keeper signs `execute_deposit(epoch)` with the first completed snapshot whose prints were all fetched after the deposit landed. The deposit receives $\lfloor amount \cdot supply / NAV \rfloor$ shares, minted into the share escrow and credited to `OwnerState.claimable_shares`; `NavMintAllocation` records them and `pending_deposits_usdc` falls. A distressed snapshot or a zero share count refunds the deposit to the claim balance instead.

The shares appear as shares ready to claim. A deposit that no NAV has covered for 259,200 seconds can be refunded with `cancel_deposit`.

Withdrawal request and execution

1. The LP signs `request_withdraw(nonce, shares)`. Shares come from the claim balance first, which the share escrow already holds, and the rest move from the wallet's share account into the escrow. A `Withdrawal` joins the queue with `unlocks_at` = now + 259,200 seconds, which the app shows.

2. After `unlocks_at`, a keeper signs `price_withdraw(epoch)` with the first completed NAV whose prints were all fetched after the unlock. The amount is $\lfloor shares \cdot NAV / supply \rfloor$; the shares are burned and the amount is reserved in `reserved_withdrawals_usdc`, outside NAV. A distressed snapshot returns the shares to the claim balance instead.
3. The app then offers `execute_withdraw`, which any signer can send. The program checks that NAV is not negative and that cash covers the amount on top of every protected reserve, then moves the amount to `OwnerState.claimable_usdc`.

Claiming balances

Close fills, liquidation remainders, settlements, refunds, executed withdrawals, deposit shares and bond returns all credit `OwnerState`, never a wallet token account. The owner signs `claim_balances(amount_usdc, shares)` for any amounts up to the balances. USDC moves from the vault and shares from the share escrow to token accounts the wallet owns, its associated accounts by default, and `claimable_usdc`, `claimable_shares` and `Vault.reserved_claims_usdc` fall. A USDC claim requires vault cash to cover reserved margin, pending deposits and reserved claims. The app shows the balances as ready to claim.

Proposing a market

1. On the listings page the proposer describes the market and the artist ID, and Sentinel returns a draft. The proposer completes the research inputs until nothing is missing.
2. The proposer signs `propose_market` with the bond, creating `Proposal`, `Market`, `MetricSpec`, `PriceHistory` and the bond account (chapter 9). The app lists the proposal with its shadow agreement, curator votes, timelock end and probation end.
3. Each curator signs `approve_market` with the caps and the configuration hash, or `reject_market`.
4. Keepers send `activate_market` after the timelock, `promote_market` after probation and `return_bond` once the market is live or rejected, or `slash_bond` on a failed probation.
5. The proposer moves the bond with `claim_balances` and the fee share with `claim_market_fees`.

A round from open to print

1. **Open.** A reporter signs `open_round(sequence)` once a cadence has passed. The `OracleRound` at `["round", metric, sequence]` is created with `closes_at` = `opened_at` + 900 seconds, and the metric's `next_sequence` advances.
2. **Root 1.** The reporter obtains a Reclaim proof of the artist's public page, whose claim binds the request, the extracted count and a hash of the oracle program, metric, spec hash, round and sequence. It sends a secp256k1 precompile instruction with the attestor's signature, then `submit_reclaim_observation`. The program rebuilds the claim from the artist ID, the value and its

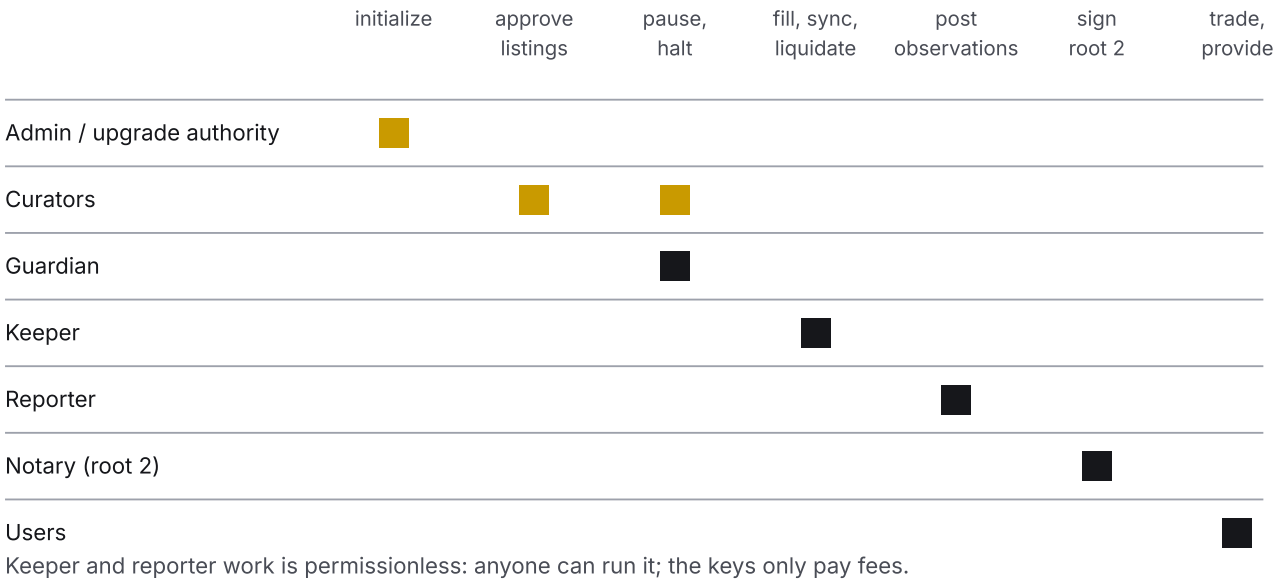
own accounts, checks the artist ID against the metric's ID and the attestor against the root-1 set, and records observation 0.

3. **Root 2.** The reporter requests the notary's signed observation of the same page and sends an Ed25519 precompile instruction, then `submit_notary_observation`. The program rebuilds the notary's claim, checks the key against the root-2 set and records observation 1. Both fetch times must lie inside the round, and the earlier one becomes `min_fetched_at`.
4. **Print.** A keeper signs `finalize_round`. When the roots agree within 50 bps, the round is in order and the index is within the circuit band, the round prints the median: the `OracleRound` records `final_index` and `finalized_at`, the `MetricSpec` records the latest print, resets the failure counter and clears side pauses, and the `PriceHistory` records the hour's close. A disagreement ends disputed, and a move beyond the band waits out of band for confirmation (chapter 4).
5. **No print.** A round that closes with a root missing is marked failed by `mark_failed_round`, from any signer; the failure counter rises and both sides pause for 21,600 seconds.

The market page lists recent rounds with both roots' values, their distance, the print and the signatures. `round.finalized` updates the index and the time of the next print.

12. Security design

Anyperp's security is a set of authorities, each limited to named instructions, and a set of checks that every instruction runs on chain. This chapter lists who can sign what and what each mechanism enforces.



The roles and what each can sign: the deploy-time admin until handover, the curator PDA with its three-of-five quorum, the guardian, permissionless keepers and reporters, the notary as root-2 attestor, and users.

The authority model

Role	Key	Can sign	Bounded by
Deploy-time admin	the key that deploys both programs	`initialize_oracle_config`, `initialize_exchange`, `initialize_vault`, `bootstrap_launch_market`	each requires the signer to be the program's current upgrade authority or the exchange admin; the launch path also requires an untouched vault
Curator PDA	`["curator"]` in `anyperp_perps`	metric creation, slash halts, attestor rotations, curator pauses, program upgrades	signs only inside perps instructions, after three member votes on one payload hash, or on objective listing conditions
Curators	five members of `CuratorConfig`	`approve_market`, `reject_market`, `approve_oracle_policy`, `approve_program_upgrade`	three distinct members per action; one vote each
Guardian	one key in `OracleConfig` and `Exchange`	`guardian_pause_metric`, `halt_metric`	pauses of at most 86,400 s per episode; halts only after `failed_rounds_before_halt` failed rounds

Role	Key	Can sign	Bounded by
Keeper	any key	rounds, syncs, valuations, fills, liquidations, LP execution, listing transitions, settlement	every precondition is on-chain state; paid a fixed keeper fee
Reporter	any key	`open_round` and the observation instructions	an observation counts only with a valid attestor signature
Notary	an Ed25519 key in the root-2 attestor set	observation claims, not transactions	its key enters and leaves only through timelocked rotation
User	the wallet	its own orders, margin, deposits, withdrawals, claims and proposals	the program's checks on each instruction

Handover. At deploy, the admin holds the upgrade authority of both programs, and the two initializers verify it by reading each program's `ProgramData`. The launch markets are listed in that state. The operator then transfers both upgrade authorities to the curator PDA, and each program's `ProgramData` shows the handover to anyone who reads it. From then on the admin key holds no upgrade power, `bootstrap_launch_market` rejects every signer, and `set_local_withdrawal_queue`, the admin's one remaining instruction, works only on local test exchanges.

Guardian. A guardian pause stops new opening orders for at most 86,400 seconds from the start of its episode. The guardian can lengthen an active pause within that limit but never shorten it, and a new episode needs a curator decision through `curator_pause_metric`. A guardian halt needs 4 consecutive failed rounds at launch. The guardian writes no price, changes no cap and moves no funds, and closes, margin additions and claims continue through a pause.

Keepers and reporters. Every keeper instruction checks objective state, such as a finalized round, a queue head or an elapsed timelock, so a keeper chooses only when to send. Its reward is `keeper_fee_usdc`, 0.01 USDC per fill, liquidation or settlement, paid to a USDC account the keeper owns. A reporter carries signatures it cannot produce.

Notary. The notary is the independent root-2 service. It fetches the public source and signs one claim per observation, binding the oracle program, the metric, its spec hash, the round, its sequence, the value and the fetch time. It answers only authenticated reporter requests over HTTPS, signs no transaction and holds no funds.

Timelocked attestor rotation

Root keys change only through an oracle policy action. Three curators vote on the same payload with `approve_oracle_policy`, whose hash is the SHA-256 of `ANYPERP_ORACLE_POLICY_V1` and the Borsh encoding of a nonce and the action, so votes on different keys never combine. The third vote calls `schedule_attestor_rotation` under the curator PDA. The new keys activate no earlier than `attestor_timelock_seconds` after scheduling, 172,800 seconds, a minimum the program enforces in

production. Each root holds at most one pending rotation, so a queued, publicly visible rotation cannot be swapped out; it can only be cancelled by another three-vote action. After activation, which anyone can send, the previous keys verify only observations fetched before activation. Every key set has 1 to 8 distinct non-zero keys.

Governed program upgrades

A program upgrade is a vote on an exact binary. `approve_program_upgrade` binds a nonce, the target program (only `anyperp_oracle` or `anyperp_perps`), the loader buffer, the rent spill account, which must be the protocol treasury, and the SHA-256 of the buffer's ELF bytes. The vote account sits at `["upgrade", hash]` of that payload. The third distinct vote sets `execute_after` = now + 259,200 seconds, once. `execute_program_upgrade` then checks three votes and an elapsed timelock, an upgrade authority equal to the curator PDA, and the voted buffer: owned by the loader, under the curator PDA's authority, starting with an ELF header and hashing to the voted SHA-256. Only then does the curator PDA sign the loader's upgrade, so the executed bytes are the voted bytes. The API holds no upgrade key.

Permissionless work never depends on a user's token account

Every payout that a keeper or any other signer triggers lands in the owner's claim balance in `OwnerState`, backed one to one by the vault's `reserved_claims_usdc` and the share escrow: refunds, closes, liquidations, settlements, executed withdrawals, deposit shares and bond returns. Keeper instructions name only program accounts, the fixed fee accounts and the keeper's own token account. Closing, freezing or reassigning a user's token account therefore cannot stall a fill, a liquidation or a settlement. Only instructions the owner signs touch the owner's token accounts, and `claim_balances` pays to any account of the right mint that the wallet owns.

Rounding that favours the vault

All arithmetic is checked integer arithmetic in `anyperp_domain`, and every rounding goes to the vault. Notional, fees, required and maintenance margin, funding and borrow charges, utilization and the borrow rate round up. PnL, deposit shares and withdrawal amounts round down. Buys fill at a price rounded up and sells at a price rounded down. A merged position's entry price rounds against the trader, and its checkpoints keep old charges owed. The fee split rounds the protocol, insurance and proposer shares down and leaves the remainder in the vault. NAV counts outstanding charges rounded down and reserves funding credits rounded up. The side cap is strict: an opening fill must leave notional below the cap.

The order queue barrier

Each market's orders form one queue in landing order, and only its head can execute. When a market syncs to the first print eligible for the head, it records that print as its barrier. The market cannot move to a later print (`PendingEligibleOrder`) until every order eligible at the barrier is filled or rejected, and margin removal cannot use a later print while the barrier stands. A keeper therefore cannot skip an order's print by syncing a newer round, and each order fills at exactly one print.

Anti-option devices

Next-print execution. An order fills only at the round with `previous_finalized_min_fetched_at` ≤ `landed_at` < `min_fetched_at`, the first print whose observations were all fetched after it landed, so its price is unknown when it lands. Cancellation opens only when no such print can still arrive, never while a round with visible observations is open, so a pending order is never a choice between filling and walking away.

Deposits at the next NAV. A deposit fills at the first completed NAV generation whose prints were all fetched after it landed, and the vault opens no new generation while an eligible LP intent waits at a queue head.

The withdrawal queue. A withdrawal waits 259,200 seconds, the production minimum, and is then priced at the first completed NAV whose prints were all fetched after the unlock. An LP who sees an adverse print coming cannot exit at the NAV before it. Once priced, the withdrawal is a fixed claim that executes only while the remaining NAV is not negative.

Server-built transactions checked before signing

The API builds every user transaction and never holds a user key. Before building, it checks the action against indexed state: a matching live quote, margin for the acceptable price, an owned position or order, an eligible country and a current attestation. The wallet shows the transaction's cluster, programs and amounts before the user signs. `tag_transaction` binds the transaction to one intent without changing state. The API records the SHA-256 of the unsigned message and accepts only a signed transaction with that exact message, the session's wallet as fee payer and valid signatures. Every rule that moves value is enforced again by the programs, so a prepared transaction can do no more than its instructions allow.

Sessions bound to a signed wallet message

A session starts from a single-use challenge bound to an allowed origin, expiring after 5 minutes, which the wallet signs with its own key. The API verifies the Ed25519 signature over the exact message, stores only the SHA-256 of a random 32-byte session token, and sets it in an `HttpOnly`

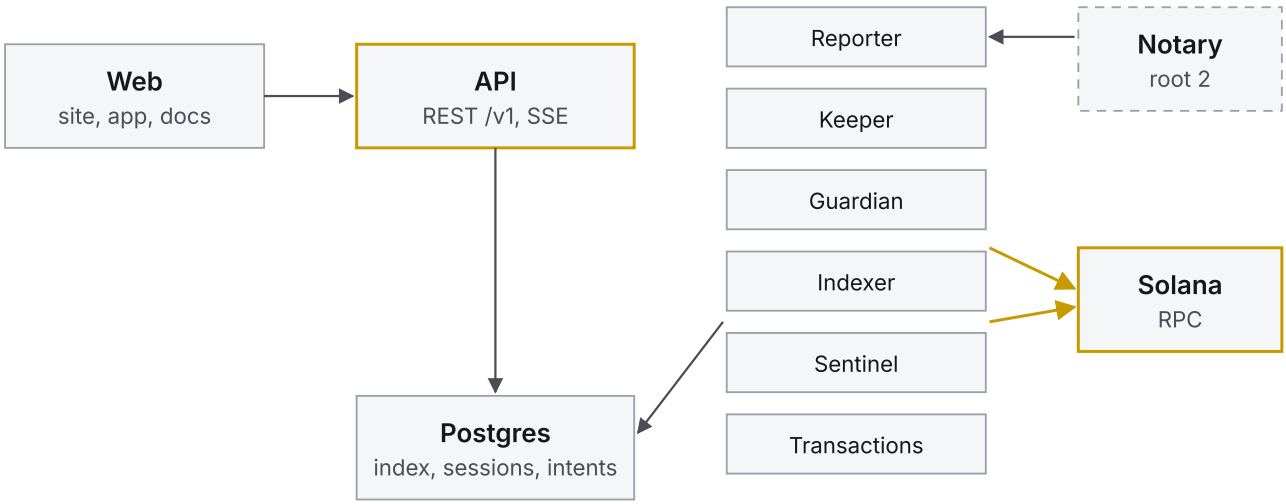
cookie that lasts 28,800 seconds. A session authorizes reads of the wallet's own data and the preparation of transactions; it signs nothing, and every transaction still needs the wallet's signature.

Geo access

In production the API reads the visitor's country only from one configured header, and only on connections from exact trusted proxy addresses. An unknown country fails closed, and the blocked-country list must contain `US`. Trading, liquidity provision, proposals and sign-in require an allowed country and a non-US-person attestation for the current terms version, rechecked on every write (section 13.6). Reads stay open to everyone.

13. Off-chain infrastructure

The two programs, `anyperp_oracle` and `anyperp_perps`, hold every balance and enforce every rule. The services around them move the protocol forward by sending permissionless instructions on a clock, turn finalized chain state into fast reads for the app, and build unsigned transactions for wallets to sign. Each service either submits an instruction that a program checks in full or reads what the programs have already finalized, so no service can write a price, change a balance or skip a check.



The API and the workers read and write Solana through RPC; wallets sign in the browser, and the API never holds a user's k

The web app reads from the API; the reporter, keeper and guardian sign their own instructions; the indexer and transaction tracker read finalized chain state into Postgres; the root-2 notary signs observations from its own infrastructure.

13.1 Services and what each does

Service	Signing key	Loop	Work
API	none	per request	REST under `/v1`, the live event stream, quotes, unsigned transactions, sessions and eligibility
Reporter	`REPORTER_KEYPAIR_JSON`	1 s between passes	opens rounds, obtains both roots' signed observations and submits them, marks expired rounds failed
Keeper	`KEEPER_KEYPAIR_JSON`	1 s between passes	finalizes rounds, fills and expires orders, liquidates, values positions, prices LP deposits and withdrawals, runs listing and settlement transitions
Guardian	`GUARDIAN_KEYPAIR_JSON`	1 s between passes	halts a metric that has reached its failed-round threshold

Service	Signing key	Loop	Work
Indexer	none	1 s between passes	projects finalized accounts and program events into Postgres
Sentinel	none	10 s between passes	writes advisory flags per market; drafts specs and cost dossiers on request
Transaction tracker	none	1 s between passes	reconciles submitted transactions and publishes their state
Notary	its own Ed25519 attestation key, no Solana key	per request	root 2: fetches the source itself and signs root 2's claim; run by an independent operator

API

The API is a Node 22 HTTP server. It validates every request against the contract's JSON Schema before dispatch and every response against the same schema before sending it; a response that does not match the published contract is replaced by `INTERNAL_ERROR`. Reads come from the indexed projections in Postgres. The API holds no signing key: the Anchor provider it builds instructions with throws on any attempt to sign. Its `/health` returns ready only when the database answers, the indexed finalized clock is at most `CHAIN_STALL_AFTER_SECONDS` (60) behind wall time, both programs are executable, every required account is indexed and both indexer cursors exist.

Reporter

For every market, a reporter pass checks the `MetricSpec` and its spec hash, sends `mark_failed_round` if the current round closed without both observations, sends `open_round` once the cadence has elapsed, and then obtains and submits one observation per missing root. The reporter holds neither root's signing key; it carries their signatures to the chain.

- **Root 1.** The reporter requests a Reclaim zkFetch proof of the artist's public profile page, each in its own subprocess with a 120-second deadline; the proof's context binds a hash of the program, metric, spec hash, round and sequence. It checks the attestor's Ethereum-style address against root 1's on-chain keys and sends a Secp256k1 precompile instruction followed by `submit_reclaim_observation`. The oracle rebuilds the claim identifier and signed message from the artist ID, value, time, owner and epoch, and requires the metric's ID to be the hash of that artist's metric identity, so a proof of another page cannot price this market.
- **Root 2.** The reporter calls the notary's `POST /v1/notarize` with a bearer token and a 20-second deadline and sends an Ed25519 precompile instruction followed by `submit_notary_observation`. The notary's claim binds a fixed domain tag, the program, metric, spec hash, round, sequence, root, value and fetch time.

The oracle accepts a precompile only in its exact self-contained layout and only from a signer in that root's `AttestorSet` for the observation time, which must lie inside the round's window. Reporting is

permissionless, and a second `open_round` for the same sequence simply fails. The service works through `REPORTER_CONCURRENCY` markets at a time (default 1, at most 16).

Notary

The notary is root 2's HTTPS service, run by an independent operator with its own Ed25519 key, a bearer token and the oracle program ID; it holds no Solana key and needs no RPC. On `POST /v1/notarize` it checks that the metric and round addresses derive from the stated program, artist and sequence and that the round is open, then fetches the artist's fixed public profile page itself, never a caller-supplied URL or value. It rejects redirects, error responses, pages over 2,000,000 bytes and missing or conflicting values, takes the integer monthly-listener field and signs. It serves at most 4 fetches at a time and 120 requests per client per minute.

Each root's `AttestorSet` lists up to 8 keys with a validity window. A rotation is scheduled by a curator vote and activates after the 48-hour attestor timelock; the previous keys stay valid only for claims observed before activation.

Keeper

One keeper pass runs these steps in order, each item its own retry boundary:

- **LP queues:** `execute_deposit` and `price_withdraw` for each FIFO queue head at its first eligible complete NAV snapshot; `activate_market` for listings past their timelock, at a complete valuation cohort.
- **Rounds:** `finalize_round` when both observations are present, `mark_failed_round` after close, four metrics in parallel.
- **Markets and orders:** `sync_market` to the first executable print, `liquidate`, `value_position`, then `execute_order` from the queue head at each order's first eligible print, or `expire_order` for a halted metric or a timed-out order.
- **Settlement and listings:** `settle_metric`, `halt_market`, `settle_market` and `settle_position` for a halted metric; `promote_market`, `slash_bond` and `return_bond` as listings resolve.
- **Housekeeping:** closes NAV snapshots and mint allocations more than 24 generations old and extends the address lookup table.

Keeper transactions request a 1,000,000 compute-unit limit, and the keeper fee is paid to the keeper's USDC token account.

Guardian

When a metric is not halted and `consecutive_failed_rounds` has reached `failed_rounds_before_halt`, the guardian sends `halt_metric`. The guardian key can do two things in the programs: `halt_metric`, which the oracle accepts only at that threshold, and `guardian_pause_metric`, a pause on opens whose episode ends at most 24 hours after it starts. It cannot write a price, change a parameter or move funds.

Indexer

The indexer reads at finalized commitment. One pass fetches every account of both programs, every share-token account, the vault, protocol, insurance and proposer USDC token accounts, the share mint and the Clock sysvar, and accepts the read only when all of it comes from one slot. It applies that snapshot, then decodes program events: for each program, the oldest `INDEXER_LOG_BATCH_SIZE` (default 250) finalized transactions after its cursor per pass, so projections refresh between batches during a backfill. The `AttestationVerified` event it stores lets the round explorer rebuild the exact message each root signed.

Sentinel

The Sentinel worker reads the projections every 10 seconds and writes one advisory assessment per market, such as a critical flag when the metric is halted. Drafting runs inside the API: `POST /v1/proposals/draft` turns the proposer's description and researched inputs into a `ProposalDraft` with a metric spec, a cost dossier whose target caps are $\lfloor \kappa\, c_{up} L_0 \rfloor$ and $\lfloor \kappa\, c_{down} L_0 \rfloor$, flags, a suggested bond and a `missingConfiguration` list. Every spec and financial field is computed deterministically; a language model adds advisory notes and nothing else. Every output carries `advisoryOnly: true`; only a wallet-signed `propose_market` locks a bond or creates a listing.

Transaction tracker

The tracker reconciles up to 100 submitted or confirmed transaction intents per pass, oldest first. It reads each signature's status, checks that the included message is byte-identical to the prepared one, records the intent as confirmed, finalized, failed or expired, and publishes each transition to the owner's stream as `transaction.updated`. It also deletes stale quotes and expired rate-limit buckets.

13.2 Coordination

Leases and heartbeats

Each role loops on a `JobLease` row keyed by its name. A process takes the lease only when it is free, expired or already its own, holds it for 300 seconds and extends it every 30 seconds during a long pass. On success it clears the lease and records `completedAt`; on failure it records `RETRY_REQUIRED` and tries again. Several replicas of a role can run while only one works at a time, and the programs' checks decide the outcome if two passes ever overlap. Each worker serves only `/health`, healthy when its last pass ended without an error and its lease is live or its last pass completed within 300 seconds.

Idempotency

- **Prepared transactions:** each `POST /v1/transactions/prepare` carries an `idempotencyKey` of 8 to 128 characters. The intent is unique per owner, operation and key hash and stores a hash of the

request; the same key with other input returns `CONFLICT`. Each transaction begins with `tag_transaction`, which carries a hash of owner, operation and key and changes no state, so two intents never share bytes.

- **Nonces:** orders, closes, deposits, withdrawals and proposals are PDAs seeded by a per-wallet nonce. The API reserves nonces under a per-wallet advisory lock above `OwnerState.last_nonce`, and the program rejects any nonce not greater than the last one used.
- **Events and outbox:** a `ChainEvent` id is `signature:programAddress:instructionIndex:eventIndex`, and every outbox row has a unique key, so replays insert nothing new.
- **Cursors:** the event cursor advances by compare-and-set on the previous signature.

Finalized-chain indexing and projections

`applySnapshot` takes a Postgres advisory lock and, in one database transaction, ignores any snapshot older than the newest indexed slot, writes only changed rows, deletes accounts absent from the snapshot and advances both cursors. Applying the same snapshot twice creates no new events. Event ingestion resolves v0 lookup-table addresses, attributes each event to its program and top-level instruction, and retries a transaction whose finalized data is not yet available instead of skipping it.

`ChainProjection` holds each account exactly as the program IDL decodes it, with integers as decimal strings. The API's read model builds every response from these rows, computing positions' PnL, accrued funding and borrow, and liquidation prices with the same integer functions the programs use; wallet history comes from `ChainEvent`. Balances are always what the finalized accounts say. `ChainIdentity` records the genesis hash and both program IDs the derived state belongs to. The first process to start records it after the production checks of section 13.7; every later start compares it with the RPC and refuses to serve or index on a mismatch.

13.3 Data model

Table	What it holds
`ChainIdentity`	genesis hash and oracle and perps program IDs for this database
`ChainProjection`	every decoded program account, the vault and fee token accounts, share-token accounts, the share mint supply and the chain clock, each with its finalized slot
`ChainEvent`	append-only decoded program events with signature, program, instruction index, event index, slot and block time
`IndexerCursor`	finalized account slot and last processed signature per program

Table	What it holds
`LiveOutbox`	ordered live events with deduplication keys; owner-scoped rows for transaction updates
`TransactionIntent`	owner, operation, idempotency key hash, request hash, unsigned bytes and message hash, blockhash and last valid block height, reported and verified signatures, state, expiry, affected resources and order address
`OrderQuote`	the exact quote payload, its owner, input hash and expiry
`OwnerNonce`	the next nonce reserved for each wallet
`AuthChallenge`	sign-in message, nonce hash, message hash, domain, origin, expiry and consumption time
`WalletSession`	SHA-256 of the session token, wallet, terms version, attestation time, expiry and revocation
`WriteRateBucket`	per-minute write counts keyed by a hash of window and client IP
`SentinelDraft`	advisory drafts per owner and input hash, with spec and dossier hashes
`SentinelAssessment`	advisory flags per market and their artifact hash
`JobLease`	lease owner and expiry per role, attempts, last completion and last error code

No financial column is floating point: projections carry decimal strings and normalized columns use `Decimal(78, 0)` integers. Migrations apply in order at service start.

13.4 Public API

Requests and responses

Every route lives under `/v1`. A success is `{ data, meta }` with `meta = { requestId, asOf, mode, contractVersion }`, where `mode` is `indexed` and `contractVersion` is `1.0.0`. Collections return `{ items, nextCursor, total }` with `limit` from 1 to 100 (default 50); market series accept 1 to 168 and default to the latest 168 points in ascending order. Browser calls are accepted only from the exact origins in `CORS_ORIGINS`. Routes under `/v1/me`, proposal drafts and transactions need a wallet session.

Method	Path	Returns
GET	`/v1/protocol`	parameters with their provenance, units and rounding rules, chain cluster, public RPC, program IDs and mints, fee split, attestation roots with signer sets
GET	`/v1/markets`	page of market summaries: state, index, 24 h change, open interest and caps, funding, borrow, next round, side pauses, circuit band
GET	`/v1/markets/:marketId`	one market with its metric spec, cost dossier, Sentinel flags, external markets, recent rounds, settlement and risk parameters
GET	`/v1/markets/:marketId/series`	hourly or daily index points
GET	`/v1/markets/:marketId/rounds`	page of rounds with observations, agreement and band state
GET	`/v1/rounds/:roundId`	one round with both observations, signers, signatures, fetch and source times and claim payload
GET	`/v1/vault`	NAV, share price, total shares, available liquidity, reserved margin and claims, net exposure, fees, insurance fund, protocol and proposer fee balances, pending deposits, withdrawal queue
GET	`/v1/attested/subjects`	subjects with attested data: latest value and time, root agreement rate
GET	`/v1/attested/subjects/:subjectId/series`	daily attested series for one subject
GET	`/v1/attested/compare`	two subjects' daily series aligned by day, raw or rebased to 100, with their ratio
GET	`/v1/proposals`	page of proposals with bond, shadow agreement, curator votes, timelock and probation end
GET	`/v1/proposals/:proposalId`	one proposal with its spec, cost dossier and Sentinel flags
GET	`/v1/status`	liveness of oracle, reporter, keeper, indexer and Sentinel; last finalization and consecutive failed rounds per market; halts; chain slot
GET	`/v1/eligibility`	the visitor's country determination and whether trading, liquidity and proposals are open to them
GET	`/v1/session`	the wallet session or the anonymous state
GET	`/v1/me/orders`	the wallet's orders with unfilled-round count and cancellation eligibility
GET	`/v1/me/positions`	the wallet's positions with mark, PnL, accrued funding and borrow, equity, maintenance margin and liquidation price
GET	`/v1/me/lp`	shares and their value, claim balances, pending deposits and withdrawals

Method	Path	Returns
GET	`/v1/me/proposer`	proposals, locked, returned and slashed bonds, claimable and claimed proposer fees
GET	`/v1/me/history`	fills, funding, liquidations, deposits, withdrawals, margin changes, bonds, fee claims and settlements
POST	`/v1/auth/challenge`	a sign-in challenge: message, nonce, domain and expiry
POST	`/v1/auth/verify`	the session, after verifying the signed challenge and setting the session cookie
POST	`/v1/auth/logout`	the anonymous state, after revoking the session
POST	`/v1/quotes/orders`	an indicative next-print quote: fill estimate, skew impact, fees, required margin, liquidation estimate, cap headroom and maximum size
POST	`/v1/proposals/draft`	a Sentinel draft of a metric spec and cost dossier
POST	`/v1/transactions/prepare`	an unsigned Solana transaction for one action
POST	`/v1/transactions/submit`	the transaction's state after binding the wallet's signature or relaying its signed bytes
GET	`/v1/transactions/:transactionId`	prepared, submitted, confirmed, failed or expired
GET	`/v1/events`	the live event stream (Server-Sent Events)
GET	`/v1/openapi.json`, `/v1/schema.json`	the generated OpenAPI document and JSON Schema
GET	`/health`	readiness of the API, or of a worker's own loop

The contract also defines fixture selectors that serve named scenarios to client developers; the production API answers them with `FORBIDDEN`.

Live events

`GET /v1/events` streams `text/event-stream`. Each message's `id` is the outbox id, its `event` the event type and its `data` the JSON `LiveEvent`. The server polls the outbox every second and writes a heartbeat comment every 15 seconds. A client resumes with `Last-Event-ID` or a `cursor` query and can filter with `marketId`; expiry or logout closes an authenticated stream.

Event	Meaning
`round.finalized`	a round printed; carries the full round

Event	Meaning
`order.filled`	an order filled: market, order, position, round, side, size, fill price and fee, with no wallet address
`market.state_changed`	a lifecycle, halt or side-pause change, with the new summary and the previous state
`vault.updated`	the vault account changed; carries the vault
`transaction.updated`	a prepared transaction changed state; delivered only to the owning wallet's session
`stream.reset`	the cursor is outside retained history; the client refetches its current reads

The TypeScript contract package

`@anyperp/contract` is the boundary between the app and the backend: the wire types, the operation registry `operations` (method, path and public flag of each of the 29 JSON operations), the `eventStream` descriptor and a client for browsers and Node 22 with `createAnyperpClient({ baseUrl, cookie })`, `request(operationId, { params, query, body })`, `subscribe(handler)` and `ApiClientError`. It ships compiled JavaScript, declarations, OpenAPI and JSON Schema, with no backend runtime dependency. Additions stay in `/v1`; changing an established field, unit or meaning is an explicit contract change.

Units

Amounts travel as decimal integer strings. USDC uses 6 decimals; index prices, contract sizes, shares and rates use fixed9. Rates are per hour and funding velocity is per hour squared. Basis points, counts and leverage are JSON numbers; timestamps are ISO 8601 in UTC. One index point is `indexDivisor` source units, one contract unit is worth the index price in USDC, and notional is $\lceil q \cdot I_t / 10^{12} \rceil$ base units. Rounding follows the programs: buys round up and sells down; notional, margin, borrow and fees round up; trader PnL rounds down; share issuance, withdrawals and TWAP round down.

Errors

A failure is `{ error: { code, message, retryable, details?, retryAfterSeconds? }, meta }`, where `details` can list invalid fields (path, message, rule), a resource id, its current state or the configuration it requires.

Code	HTTP	Raised when
`VALIDATION_ERROR`	400	input fails the schema or a range rule
`UNAUTHORIZED`	401	a wallet route has no session, or a challenge or signature is invalid

Code	HTTP	Raised when
`SESSION_EXPIRED`	401	the session is past its expiry
`FORBIDDEN`	403	the origin is not allowed, or the wallet lacks the role for a curator action
`GEO_BLOCKED`	403	country determination or attestation does not permit the write
`NOT_FOUND`	404	the resource or route does not exist
`CONFLICT`	409	an idempotency key is reused with other input, an order is already pending in the market, a position is open on the other side, or an LP queue is full
`CAP_EXCEEDED`	409	the fill would reach the side cap or exceed the vault's liquidity reserve
`MARKET_NOT_TRADABLE`	409	the market is halted, settled, paused on that side or has no print yet
`ORDER_NOT_CANCELLABLE`	409	the order has not reached a cancellation condition
`WITHDRAWAL_LOCKED`	409	the withdrawal is not yet priced
`QUOTE_EXPIRED`	409	the quote is past its 60-second life
`TRANSACTION_FAILED`	409	preflight or the program rejected the transaction
`RATE_LIMITED`	429	a write quota or the stream limit is reached
`CONFIGURATION_REQUIRED`	503	a required account, lookup table or configured input is not available
`INTERNAL_ERROR`	500	an unexpected failure, or 503 while finalized chain data has stopped advancing

Rate limits and request bounds

Writes are limited to 120 POST requests per client IP per minute (`PUBLIC_WRITE_RATE_LIMIT`), counted in fixed one-minute windows in Postgres so every API replica shares one count; above it the API returns `RATE_LIMITED` with `retryAfterSeconds` and a `Retry-After` header. Each API process serves up to 100 open event streams. Bodies are `application/json` up to 64 KiB, and a request has 15 seconds to complete.

13.5 Server-built transactions and the wallet

The API builds every user transaction; the wallet reviews, signs and sends it.

1. **Quote.** `POST /v1/quotes/orders` computes from the finalized projections the fill estimate, fees, required margin, liquidation estimate, cap headroom and the largest size the side's cap and the vault's reserve allow. Because the order fills at a future print, the margin budget uses a price 1,000 bps above the estimated fill. The quote lives 60 seconds.
2. **Prepare.** `POST /v1/transactions/prepare` takes one action and an idempotency key. For `open_order` it requires a live matching quote and enough margin for the order's acceptable price. It reserves a nonce and assembles a 1,000,000 compute-unit limit, `tag_transaction`, any token-account creation and the action, with the wallet as fee payer and sole signer: a legacy message when it fits Solana's 1,232-byte packet, v0 with the address lookup table otherwise. The intent stores the unsigned bytes and expires after 60 seconds.
3. **Sign.** The wallet shows and signs the transaction, then sends it and reports the signature, or returns the signed bytes for the API to relay.
4. **Submit.** For relayed bytes the API checks that the signed message hashes to the prepared one, that the wallet is the first signer and that every signature verifies, then sends with preflight; for a reported signature it checks that the included message matches.
5. **Track.** The tracker follows the signature to confirmed, finalized, failed or expired, and the indexer projects the result.

Closes use the same flow with `close_position`, a reduce-only order, and `claim_balances` pays to a token account the wallet owns. Curator votes, listing transitions, attestor rotations and upgrade votes are prepared the same way and signed in each curator's own wallet. The web app connects wallets through the Wallet Standard.

13.6 Sessions and eligibility

A session ties API calls to one wallet without a password.

1. `POST /v1/auth/challenge` takes the wallet and the app's origin, which must be an allowed origin and match the request's `Origin` header. It returns a challenge that expires in 5 minutes, with a 24-byte random nonce and a sign-in message naming the domain, the wallet, the URI, the nonce and the expiry, and stating that the signature does not authorize a transaction.
2. The wallet signs the exact message bytes. `POST /v1/auth/verify` checks the Ed25519 signature, wallet, origin and expiry, requires `nonUsPersonAttested: true`, the current `termsVersion` and an eligible country, and consumes the challenge in one conditional update, so it works once.
3. The API creates a 32-byte random token, stores only its SHA-256, and sets the `anyperp_session` cookie with `HttpOnly`, `Secure`, a lifetime of 28,800 seconds and `SameSite=Lax` for same-site domains or `SameSite=None` across sites. Logout revokes the session and clears the cookie.

US persons cannot use the service. The API reads the visitor's country from one configured header (`cf-ipcountry` by default) and trusts it only on a request from the trusted edge in front of the API, which overwrites that header and adds a secret `x-anyperp-geo-token` that browsers never see; the API compares the token in constant time, or accepts an exact trusted proxy address instead. The value must be an ISO 3166-1 alpha-2 code, and the blocked list always contains `US`. `GET`

`/v1/eligibility` returns the country, the determination (`allowed`, `geo_blocked` or `unknown`) and whether trading, liquidity provision and proposals are open. Reads are open to everyone. Transaction preparation, submission and proposal drafts recheck eligibility, the session's attestation and its terms version on every call and return `GEO_BLOCKED` unless the determination is `allowed`.

13.7 Hosting layout

Eight backend services run on Railway as Node 22 processes rooted at `backend/`, beside one Postgres and the web service. Builds run `npm ci --include=dev && npm run build`, which generates the Prisma client and the contract without contacting the database; each start wrapper runs `prisma migrate deploy` and then the process. Every service binds the `PORT` Railway injects, answers health checks on `/health` and restarts on failure.

Service	Start	Secrets it holds
`Postgres`	Railway Postgres	—
`api`	`npm start`	the geo token and language-model credentials; no signing key
`reporter`	`npm run worker:reporter`	`REPORTER_KEYPAIR_JSON`, the Reclaim application ID and secret, the notary bearer token
`keeper`	`npm run worker:keeper`	`KEEPER_KEYPAIR_JSON`
`guardian`	`npm run worker:guardian`	`GUARDIAN_KEYPAIR_JSON`
`indexer`	`npm run worker:indexer`	none
`sentinel`	`npm run worker:sentinel`	language-model credentials
`transactions`	`npm run worker:transactions`	none
`notary`	`npm run worker:notary`	the notary's Ed25519 key and bearer token, held by its independent operator
`web`	the Next.js server	none

The API and the six chain workers share one configuration, including `NODE_ENV=production`, `SETTINGS_PATH=config/production.json`, `CHAIN_CLUSTER`, `CHAIN_RPC_URL`, `DATABASE_URL=${{Postgres.DATABASE_URL}}`, `ANYPERP_STACK_JSON` (the public program, mint, vault, market and governance addresses), both program IDs and `ATTESTATION_MODE=real`. The API adds origin, terms, cookie and geo settings; the reporter adds root 1's attestor addresses and the notary's URL and public key. The notary needs only its key, its token, the oracle program ID and the database URL for its migration step. The web service installs the packed contract package and reaches the API through the trusted edge at `NEXT_PUBLIC_ANYPERP_API_URL`.

Only the reporter, keeper and guardian hold Solana keys, one each, and each must match its entry in the public role registry and differ from the others, the admin, the protocol treasury and every curator. Curator and upgrade keys never reach a service. Before serving, every production process checks that its settings are labelled `production` with production timings, that the registry lists five distinct curators, that both programs' upgrade authority is the curator PDA, that the on-chain `OracleConfig`, `Exchange` and `CuratorConfig` match the registry, and that the chain identity matches.

Services reach Solana through `CHAIN_RPC_URL`, the operator's RPC provider; they send at confirmed commitment and index at finalized commitment. Wallets get their endpoint from `/v1/protocol` as `chain.rpcUrl`, an intentionally public HTTPS endpoint without credentials, and prepared transactions name the same cluster.

14. Parameters

This chapter lists every protocol and risk parameter with its launch value, its unit, the place that enforces it and the path that changes it. Values come from the production protocol configuration, the launch cohort configuration and the programs.

How parameters are set and changed

The "Set by" column uses seven paths.

Path	How a value is set	How it changes
Program	a constant or a fixed check in `anyperp_oracle` or `anyperp_perps`	a program upgrade
Service	a constant in the API or worker code	a service release
Exchange	written once into `Exchange.params` by `initialize_exchange`, signed by the deployment admin while it is still the upgrade authority	a program upgrade; no instruction rewrites `Exchange.params`
Oracle	written once into `OracleConfig` by `initialize_oracle_config`	a program upgrade
Market	filled from the protocol configuration into each market's `propose_market`, or `bootstrap_launch_market` for the launch cohort, checked against the exchange policy and, for a proposal, bound into the curators' approval hash	fixed for that market once posted
Curators	a vote of three of the five `CuratorConfig` members, executed by the curator PDA	another vote
Operator	the production settings file (`SETTINGS_PATH`) or an environment variable	redeploying the service; it never overrides a program check

A program upgrade needs three curator votes that bind the target program, the buffer, the full ELF SHA-256 and a governance nonce, followed by a 259,200-second (72-hour) delay. Both programs' upgrade authority is the curator PDA. In production mode the programs also hold hard bounds on many values below; the "Enforced in" column states them.

Rounds and oracle

A round prints when both observations v_1, v_2 agree within the tolerance τ; the index is their floored median m_t over the divisor D, and a print outside the circuit band β around the reference close R waits for confirmation:

$$|v_1 - v_2| \cdot 10^4 \le \min(v_1, v_2) \cdot \tau, \qquad I_t = \left\lfloor \frac{10^9 \cdot m_t}{D} \right\rfloor, \qquad |I_t - R| \cdot 10^4 \le R \cdot \beta$$

Parameter	Launch value	Unit	Enforced in	Set by
Round cadence `cadence_seconds`	900 (15 min)	seconds	`open_round` opens no earlier than the last open plus the cadence; a round closes at open plus cadence; `propose_market` requires exactly 900	Market
Attestation roots	2: root 1 Reclaim (secp256k1), root 2 independent notary (Ed25519)	roots	`submit_reclaim_observation`, `submit_notary_observation`, `finalize_round`	Program
Keys per root	up to 8	keys	`AttestorSet` (`MAX_ATTESTORS`)	Program
Root tolerance τ `tolerance_bps`	50 (0.5 %)	bps of the smaller observation	`finalize_round`; production at most 50	Market
Circuit band β `band_bps`	1,000 (10 %)	bps of R	`finalize_round`; production at most 1,000	Market
Circuit reference window	86,400 (24 h)	seconds	R is the last hourly close at or before now minus the window; production exactly 86,400	Oracle
Circuit confirmations `confirmation_rounds`	2	agreeing rounds	an out-of-band level prints on the second agreeing round in the same direction; at least 2	Market
Side pause `side_pause_seconds`	21,600 (6 h)	seconds	opens pause on the moving side, or both sides after a disputed or failed round; production at least 21,600, oracle at most 86,400	Market
Failed rounds before halt `failed_rounds_before_halt`	4	consecutive rounds without a print	`halt_metric`	Market
Index divisor D `index_divisor`	1,000,000	source units per index point	`submit_reclaim_observation`, `submit_notary_observation`, `finalize_round`	Market
Notary time binding	fetch time equals source time	—	`submit_notary_observation`	Program

Parameter	Launch value	Unit	Enforced in	Set by
History bucket	3,600 (1 h)	seconds	`PriceHistory.record` keeps one close per hour; production exactly 3,600	Oracle
History capacity	168	hourly closes	`HISTORY_CAPACITY`	Program
Settlement TWAP window	86,400 (24 h)	seconds	`settle_metric` averages the closes over the window ending at the halt; production exactly 86,400	Oracle
Attestor rotation timelock	172,800 (48 h)	seconds	`schedule_attestor_rotation` to `activate_attestor_rotation`; production at least 172,800	Oracle; rotations by Curators
Attestor key validity end	4,102,444,800 (2100-01-01 UTC)	Unix seconds	`AttestorSet.permits`	Oracle; then each rotation
Guardian pause episode	86,400 (24 h)	seconds	`guardian_pause_metric`; production exactly 86,400	Oracle; Curators reset or extend with `curator_pause_metric`

With the divisor of 1,000,000, an artist at 60,000,000 listeners has an index of 60 points, stored as 60,000,000,000 in fixed9.

Caps and listing

Target caps are $cap_{long} = \lfloor \kappa_9 \, c_{up} L_0 / 10^9 \rfloor$ and $cap_{short} = \lfloor \kappa_9 \, c_{down} L_0 / 10^9 \rfloor$ in USDC base units, and an opening fill must leave a side's notional strictly below its active cap. During probation the active cap is $\lfloor cap \cdot a / 10^4 \rfloor$ with a = 2,500.

Parameter	Launch value	Unit	Enforced in	Set by
Cap fraction κ `kappa_fixed9`	250,000,000 (0.25)	fixed9	`propose_market` requires the target caps to equal the formula; above 0 and at most 1	Exchange
Up cost c_{up} `cost_up_usdc_per_unit`	1,000 (0.001 USDC per listener)	USDC base units per source unit	`propose_market`, stored in `Proposal`	Market, from the cost dossier
Down cost c_{down} `cost_down_usdc_per_unit`	3,000 (0.003 USDC per listener), 3 × c_{up}	USDC base units per source unit	`propose_market`	Market, from the cost dossier

Parameter	Launch value	Unit	Enforced in	Set by
Reference level L_0 `reference_level`	10,000,000 listeners per launch market; per proposal after launch	listeners	`propose_market`	Market, from the cost dossier
Launch cohort	20 markets, live at full target caps without bond or shadow	markets	`bootstrap_launch_market`, accepted only from the deployment admin while it holds the upgrade authority and before the vault's first valuation	Exchange admin, once
Approved caps	above 0, at most the target caps	USDC base units	`approve_market`; every approving vote names the same caps	Curators
Approval quorum	3 of 5	curator votes	`approve_market`, `reject_market`	Program
Shadow minimum `shadow_seconds`	1,209,600 (14 days)	seconds	`approve_market`; production at least 1,209,600	Exchange
Minimum root agreement `min_agreement_bps`	9,900 (99 %)	bps of evaluated rounds	`approve_market` over shadow, `promote_market` and `slash_bond` over probation; 9,900 to 10,000	Exchange
Agreement coverage	window ÷ cadence: 1,344 rounds for shadow, 2,880 for probation	evaluated rounds	the agreement test runs only once this many rounds are evaluated	Program
Listing timelock `approval_timelock_seconds`	172,800 (48 h)	seconds	`activate_market`; production at least 172,800	Exchange
Probation `probation_seconds`	2,592,000 (30 days)	seconds	`promote_market`; production at least 2,592,000	Exchange
Probation cap multiplier a `probation_multiplier_bps`	2,500 (25 %)	bps of approved caps	`activate_market`; exactly 2,500	Exchange
Minimum bond `min_bond_usdc`	5,000,000,000 (5,000 USDC)	USDC base units	`propose_market`; production at least 5,000 USDC	Exchange
Maximum bond `max_bond_usdc`	25,000,000,000 (25,000 USDC)	USDC base units	`propose_market`; production at most 25,000 USDC	Exchange
Registered markets	up to 24	markets in the vault	`activate_market`, `bootstrap_launch_market` (`MAX_MARKETS`)	Program

At the launch costs and reference level, each launch market's target caps are 2,500 USDC long and 7,500 USDC short. A later proposal with the same inputs trades at 625 USDC long and 1,875 USDC short during probation.

Trading and margin

An order fills at $P = I_t \left(1 + \frac{s_{before} + s_{after}}{2S}\right)$, rounded up for buys and down for sells. A position with notional N needs initial margin $m_{init}(N) = \lceil N \cdot 10^4 / \lambda_{bps} \rceil$, and the escrow also covers the trading fee ϕ and the keeper fee k. It is liquidatable when its equity e, margin plus PnL minus the funding and borrow charges c_F and c_B, falls below $m_{maint} = \lceil N(q, I_t) \cdot \mu_{bps} / 10^4 \rceil$.

Parameter	Launch value	Unit	Enforced in	Set by
Maximum leverage λ_{bps} `max_leverage_bps`	30,000 (3×)	bps	margin checks in `execute_order` and `modify_margin`; 10,000 to 30,000; the API accepts leverage 1, 2 or 3	Exchange
Maintenance margin μ_{bps} `maintenance_margin_bps`	2,000 (20 %)	bps of notional at the index	`liquidate`; at least the exchange minimum, and $\mu_{bps} \cdot \lambda_{bps} < 10^8$, which keeps it below the one-third initial margin at 3×	Market
Minimum maintenance margin `min_maintenance_margin_bps`	2,000	bps	`propose_market`	Exchange
Skew scale S `skew_scale_usdc`	1,000,000,000,000 (1,000,000 USDC)	USDC base units	fill price and funding drift	Market
Execution rule	first finalized round whose observations were all fetched after the order landed	—	`execute_order`	Program
Acceptable price	set on each order	fixed9	`execute_order`: buys at or below, sells at or above	Per order
Orders per position	1 pending	orders	`open_order`	Program
Cancellation threshold `cancel_after_rounds`	2	completed rounds without a print	`cancel_order`; at least 2	Exchange
Order timeout `cancel_timeout_seconds`	86,400 (24 h)	seconds	`cancel_order`, `expire_order`	Exchange

Parameter	Launch value	Unit	Enforced in	Set by
Quote price buffer `quotePriceBufferBps`	1,000 (10 %)	bps above the estimated fill	API quote margin budget	Operator
Quote lifetime	60	seconds	API	Service

Funding and borrow

Funding drifts with skew and accrues at every synchronized print over the elapsed time Δt, integrated as a trapezoid. A positive rate charges longs and credits shorts. The borrow rate b rises with the utilization u, the higher of the two sides' open interest in bps of their caps.

$$f_t = f_{t-1} + \left\lfloor \frac{v \cdot s_{t-1} \cdot \Delta t}{S \cdot 3600} \right\rfloor, \qquad b = b_0 + \left\lceil \frac{b_1 \cdot u}{10^4} \right\rceil, \qquad u = \max\left(\left\lceil \frac{10^4 \cdot N_{long}}{cap_{long}} \right\rceil, \left\lceil \frac{10^4 \cdot N_{short}}{cap_{short}} \right\rceil \right)$$

Parameter	Launch value	Unit	Enforced in	Set by
Funding velocity v `funding_velocity`	1,000 (0.000001)	fixed9 per hour squared at $s = S$	`sync_market`, drift rounded down	Market
Borrow base rate b_0 `borrow_base_rate`	1,000 (0.000001, about 0.876 % a year simple)	fixed9 per hour	`sync_market`	Market
Borrow slope b_1 `borrow_slope_rate`	9,000 (0.000009 more at full utilization)	fixed9 per hour	`sync_market`	Market

Both sides pay the borrow fee on their size, rounded up, and both funding and borrow accrue into cumulative indices that each position settles against at close, liquidation or settlement.

Vault and withdrawals

Parameter	Launch value	Unit	Enforced in	Set by
Withdrawal queue `withdrawal_queue_seconds`	259,200 (72 h)	seconds	`request_withdraw` sets the unlock time; production at least 259,200 in program and configuration	Exchange
Withdrawal pricing	first complete NAV snapshot whose prints were all fetched after unlock	—	`price_withdraw`	Program

Parameter	Launch value	Unit	Enforced in	Set by
Deposit pricing	first complete NAV snapshot whose prints were all fetched after the deposit landed	—	`execute_deposit`	Program
Deposit refund	an unpriced queue head with no eligible snapshot, 259,200 seconds after landing	—	`cancel_deposit`	Program
Queued deposits `max_pending_deposits`	32	requests	`deposit`; 1 to 64	Exchange
Queued withdrawals `max_pending_withdrawals`	32	requests	`request_withdraw`; 1 to 64	Exchange
Requests per wallet	1 deposit and 1 withdrawal pending	requests	`OwnerState`	Program
Exposure reserve ratio r `vault_reserve_bps`	10,000 (100 % of net skew)	bps	the exposure reserve R_X is the larger of $\lceil \lvert N_{long} - N_{short} \rvert \cdot r / 10^4 \rceil$ and the PnL credit bound; opening fills keep the vault's cash above all reserves	Exchange
Minimum permanent liquidity	1,000,000,000 (1,000 USDC)	USDC base units	`initialize_vault` locks it and mints its shares to the dead-shares account	Operator, at vault initialization
Share decimals	9	decimals	the first deposit mints 1 share per USDC; later deposits mint amount × total shares ÷ NAV, rounded down	Program
NAV snapshot retention	24 generations	generations	the keeper closes older snapshots	Service

Fees

Parameter	Launch value	Unit	Enforced in	Set by
Trading fee b_{trade} `trading_fee_bps`	10 (0.1 %)	bps of fill notional, on opens and closes	`execute_order`, rounded up	Exchange

Parameter	Launch value	Unit	Enforced in	Set by
Liquidation fee b_{liq} `liquidation_fee_bps`	100 (1 %)	bps of notional at the index	`liquidate`	Exchange
Keeper fee k `keeper_fee_usdc`	10,000 (0.01 USDC)	USDC base units per fill, liquidation or settlement	paid from the position's margin to the keeper	Exchange
Vault share `vault_fee_bps`	6,000 (60 %)	bps of each trading, liquidation and borrow fee	`distribute_fee`: the remainder after the other shares, which round down; exactly 6,000	Exchange
Protocol share `protocol_fee_bps`	2,000 (20 %)	bps	to the protocol fee account, which buys ANYPERP and burns it; exactly 2,000	Exchange
Insurance share `insurance_fee_bps`	1,000 (10 %)	bps	to the insurance fund token account; exactly 1,000	Exchange
Proposer share `proposer_fee_bps`	1,000 (10 %)	bps	credited per market, claimed with `claim_market_fees`; exactly 1,000	Exchange
Slashed bond	100 % to the insurance fund	—	`slash_bond`	Program

Operations

Parameter	Launch value	Unit	Enforced in	Set by
Program upgrade delay	259,200 (72 h) after 3 of 5 curator votes	seconds	`execute_program_upgrade` (`UPGRADE_TIMELOCK_SECONDS`)	Program
Curator set	5 members, quorum 3	members	`CuratorConfig`	Exchange
Compute limit	1,000,000	compute units per keeper or prepared transaction	keeper, API	Service
Transaction size	1,232; legacy when the message fits, v0 with the address lookup table otherwise	bytes	API	Service
Prepared transaction lifetime	60	seconds	API	Service

Parameter	Launch value	Unit	Enforced in	Set by
Write rate limit `PUBLIC_WRITE_RATE_LIMIT`	120	POST requests per client IP per minute; 1 to 100,000	API, shared in Postgres	Operator
Session lifetime `SESSION_TTL_SECONDS`	28,800 (8 h)	seconds; 60 to 2,592,000	API	Operator
Sign-in challenge lifetime	300	seconds	API	Service
Geo proxy token `GEO_TRUSTED_PROXY_TOKEN`	at least 32 characters	characters	API eligibility	Operator
Blocked countries `GEO_BLOCKED_COUNTRIES`	`US`	ISO 3166-1 alpha-2 codes; must include `US`	API eligibility	Operator
Event streams	100	open streams per API process	API	Service
Event heartbeat	15	seconds	API	Service
Chain stall threshold `CHAIN_STALL_AFTER_SECONDS`	60	seconds; 1 to 3,600	API readiness and `/v1/status`	Operator
Indexer batch `INDEXER_LOG_BATCH_SIZE`	250	transactions per program per pass; 1 to 1,000	indexer	Operator
Reporter concurrency `REPORTER_CONCURRENCY`	1	markets at a time; 1 to 16	reporter	Operator
Reclaim proof deadline `RECLAIM_REQUEST_TIMEOUT_MS`	120,000	milliseconds; 1,000 to 600,000	reporter	Operator
Notary request deadline `NOTARY_REQUEST_TIMEOUT_MS`	20,000	milliseconds; at most 120,000	reporter	Operator
Notary source deadline `NOTARY_SOURCE_TIMEOUT_MS`	15,000	milliseconds; at most 60,000	notary	Operator

Parameter	Launch value	Unit	Enforced in	Set by
Notary request limits	4 concurrent fetches, 120 requests per client per minute, 4,096-byte requests, 2,000,000-byte source pages	—	notary	Service
Worker pass interval	1 (Sentinel 10)	seconds	every worker	Service
Worker lease	300, extended every 30 during a pass	seconds	`JobLease`	Service
Worker health window	300 since the last completed pass	seconds	worker `/health`	Service
Page size	1 to 100, default 50; series 1 to 168, default 168	items	API	Service

Token

Parameter	Launch value	Unit	Notes
ANYPERP supply	1,000,000,000	tokens	fixed; minted once, lowered by every burn
Launch	100 %	of supply	one public launch on equal terms; no presale, team, investor, airdrop or reserved allocation, no vesting
Use of the protocol share	100 %	of the protocol's 20 %	buys ANYPERP on the market; the bought ANYPERP is burned

15. The ANYPERP token

ANYPERP is the protocol's token. This chapter gives its supply and launch, the flow that turns the protocol's fee share into a buyback and burn, and where the token sits relative to the rest of the design.

Supply and launch

Ticker	ANYPERP
Supply	1,000,000,000, fixed
Launch	100% to one public launch, on equal terms
Allocations	none: no presale, no team or investor allocation, no airdrop, no reserved pool, no vesting

The supply is minted once and only ever falls. Nothing in the protocol mints more, and every unit the buyback acquires is burned.

The fee flow

Every fee the protocol takes splits the same way (chapter 7): 60 % to the vault, 20 % to the protocol, 10 % to the insurance fund and 10 % to the market's proposer. The split happens inside the instruction that charges the fee: `execute_order` for the trading fee on an open, an addition or a close, `liquidate` for the liquidation fee, and the collection of the borrow fee whenever quantity closes. The protocol's share arrives in USDC in the protocol fee account that `Exchange` records.

That share has one use. It buys ANYPERP on the market, and the ANYPERP it buys is burned with the token program's burn instruction, which lowers the mint's supply.

Over any period, the USDC that reaches the buyback is

$$B = 0.2 \cdot (\phi_{trade} + \phi_{liq} + \phi_{borrow})$$

where the trading fees are 10 bps of the notional of every open, addition and close, $\phi_{trade} = 0.001 \cdot V$ for a traded notional V; the liquidation fees are 1 % of the notional liquidated, $\phi_{liq} = 0.01 \cdot V_{liq}$; and ϕ_{borrow} is the borrow fee collected (chapter 6). The keeper fee and funding are not split, so neither reaches the buyback.

For example, a period whose opens, additions and closes total 10,000,000 USDC of notional pays 10,000 USDC of trading fees, of which 2,000 USDC buys ANYPERP to burn. A 10,000 USDC position opened and later closed pays two 10 USDC trading fees and sends 4 USDC to the buyback.

Where the token sits

The token's link to the protocol is the buyback. Margin, vault deposits and shares, payouts, claim balances and listing bonds are all USDC. Prices come only from the two attestation roots, and caps only from the cost dossier and κ, so the token carries no weight in either: an attacker who holds ANYPERP moves neither a print nor a cap. Curation stays with the curators and their bonds (chapter 9).

This keeps each part of the system answerable to the thing it measures. The vault's NAV reflects trader PnL and fees in USDC, and the caps reflect what moving a listener count costs. The token's supply reflects how much the markets are used.

16. Glossary and notation

Glossary

Term	Definition
ANYPERP	The protocol's token: a fixed supply of 1,000,000,000 from one public launch; the protocol's 20 % fee share buys it on the market and burns it (chapter 15).
Round	One oracle period of one metric, opened by `open_round` at most once every 900 seconds. It accepts one observation per root fetched inside its window and ends finalized, disputed, out of band or failed.
Print	The index a finalized round records: the floored median of the two roots' observations divided by the index divisor. Orders fill at prints and positions are marked to them.
Index	A market's price in fixed9 index points, set by each print; one contract unit is worth the index in USDC.
Root	An independent attestation signer whose signature makes an observation valid. Root 1 is a Reclaim attestor signing with secp256k1, root 2 an independent notary signing with Ed25519, and each has its own `AttestorSet` of allowlisted keys.
Observation	One root's signed reading of the source in one round: raw value, fetch time, source time, signer and signature, recorded by `submit_reclaim_observation` for root 1 or `submit_notary_observation` for root 2.
Attestation	A root's signature over its claim for one round, which binds the program, metric, spec, round and sequence: a Reclaim claim for root 1, a notary claim for root 2. It is verified in the same transaction through Solana's secp256k1 or Ed25519 precompile against that root's allowlisted keys.
Tolerance	The widest gap between the two observations at which a round can print: 50 bps of the smaller observation. A wider gap leaves the round disputed, with no price.
Circuit band	The largest move of a new print against the hourly close of 24 hours earlier: 1,000 bps. A larger move prints only once a second round confirms it, and opens on the moving side pause meanwhile.
Open interest	The USDC notional of all open positions on one side of a market, valued at the current index.
Cap	The ceiling on one side's open interest: κ times the cost of moving the metric one listener times its reference level. An opening fill must leave the side strictly below it; probation applies 25 % of the approved caps.
κ (kappa)	The cap fraction, 0.25 (`kappa_fixed9` = 250,000,000): the share of break-even open interest a side may carry.
Skew	Long minus short open interest in USDC, $s = N_{long} - N_{short}$. The vault holds $-s$ in every market.
Funding	A per-hour rate between traders and the vault whose velocity is proportional to skew, so it drifts toward charging the side that holds the skew. A positive rate charges longs and credits shorts.

Term	Definition
Borrow fee	An hourly charge on both sides' positions at a rate that rises with the market's higher cap utilization, split like a trading fee.
Maintenance margin	The equity a position must keep: `maintenance_margin_bps` of its notional at the index. Below it, any keeper can liquidate.
Liquidation	Closing a position whose equity is below maintenance margin at the current print; the keeper fee and the 100 bps liquidation fee come out of its equity.
Vault	The single USDC pool that takes the other side of every position in every market. LPs own it through shares.
NAV	The vault's net asset value: its USDC minus trader margin, pending deposits, priced withdrawals and claim balances, minus traders' unrealized PnL, plus the funding and borrow traders owe. It displays as zero when negative.
NAV generation	One valuation epoch in which every registered market syncs to a print and every open position is valued. Its completed snapshot is immutable and prices the deposits and withdrawals eligible for it.
Share	The vault's SPL token with 9 decimals, a receipt on NAV. Deposits mint and withdrawals redeem shares at the price of their NAV generation.
Withdrawal queue	The FIFO of withdrawal requests. Each unlocks 72 hours after its request and is priced at the first complete NAV generation fetched after the unlock.
Claim balance	USDC and shares the program holds for a wallet, credited by closes, liquidations, settlements, refunds, priced withdrawals, minted LP shares and bond returns. The wallet moves them out with `claim_balances`.
Insurance fund	A vault-owned USDC account that receives 10 % of every fee and every slashed bond.
Proposal	A bonded request to list a metric, posted with `propose_market`, carrying the metric spec, risk parameters, manipulation costs, reference level and a hash of the cost dossier.
Cost dossier	The published evidence for a metric's manipulation costs c_{up} and c_{down} and its reference level, from which the target caps follow.
Bond	The proposer's USDC stake, 5,000 to 25,000 USDC. It returns to the proposer's claim balance when the market goes live or is rejected, and goes to the insurance fund when probation fails.
Shadow mode	The first listing phase, at least 14 days, in which a proposed metric runs full oracle rounds with no trading. Curators can approve only once at least 99 % of its evaluated rounds agree.
Probation	The 30 days after activation in which a market trades at 25 % of its approved caps. It goes live if its roots keep 99 % agreement; its bond is slashed if they do not, if the metric halts or if the market takes uncovered bad debt.
Curator	One of the five members of `CuratorConfig`. Three votes approve or reject listings with caps, rotate attestor keys, pause metrics and approve program upgrades.

Term	Definition
Guardian	A separate key with two powers: pause opens on a metric for an episode of at most 24 hours, and halt a metric that has reached its failed-round threshold. It cannot write a price or move funds.
Halt	The terminal oracle state of a metric, reached through `halt_metric` after 4 consecutive rounds without a print, or through the curator halt when probation fails. No further round opens.
Settle	Closing a halted market at its settlement price: `settle_metric` records the 24-hour TWAP, `settle_market` sets the market's index to it, and `settle_position` closes each position and credits its claim balance.
TWAP	The time-weighted average of a metric's hourly closes over the 24 hours before its halt: the settlement price of a halted market.
Keeper	A permissionless bot that finalizes rounds, executes and expires orders, liquidates, values positions, prices LP requests and runs listing and settlement transitions, earning the keeper fee.
Reporter	A permissionless bot that opens rounds, submits both roots' signed observations and marks expired rounds failed.
Notary	Root 2's HTTPS service, run by an independent operator. It fetches the artist's fixed public profile page itself and signs the round's claim with an Ed25519 key allowlisted in root 2's `AttestorSet`; it holds no Solana key.
Sentinel	The advisory AI service that drafts metric specs and cost dossiers and flags anomalies per market. It holds no key, and every output carries `advisoryOnly: true`.
fixed9	Integer fixed point with 9 decimals, so 10^9 is 1.0; used for indices, prices, contract sizes, shares and rates.
bps	Basis points, hundredths of a percent: 10,000 bps is 100 %.

Notation

Amounts in formulas are integers in their stored units unless a formula says otherwise: USDC in base units of 10^{-6}, indices, prices, sizes, shares and rates in fixed9. Where two chapters use one symbol for different quantities, its single entry gives both.

Metric, attack and caps

Symbol	Meaning
L	the metric's level (monthly listeners) at the last print; in the listing formulas, the reference level
L_0	the reference level in a proposal's cost dossier, in listeners
L_t	the level at time t

Symbol	Meaning
c_{up}, c_{down}	cost in USDC to move the metric up or down by one listener
c	a cost per listener in USDC base units, in the cap formula; in margin, a position's accrued charges $c_F + c_B$
x	the fractional move an attacker buys
$C(x)$	the cost of moving the metric by the fraction x and holding it through the print
$\Pi_{up}(x)$, $\Pi_{down}(x)$	an attacker's profit from pushing the metric up or down by x
N	a position's notional in USDC; in an attack, the attacker's notional on the side that profits
N_{long}, N_{short}	open interest per side, in USDC notional
N^*_{long}, N^*_{short}	break-even open interest per side, $c_{up}L$ and $c_{down}L$
κ	cap fraction (0.25)
κ_9	κ as stored, `kappa_fixed9` = 250,000,000
cap_{long}, cap_{short}	the open-interest caps of each side, in USDC
cap_{side}, cap^{prob}_{side}	one side's cap, and that cap during probation
a	the active cap multiplier in bps (10,000 live, 2,500 in probation); in the vault, a deposit amount

Oracle

Symbol	Meaning
v_1, v_2	the raw counts root 1 and root 2 observe in a round
m_t	the floored median of v_1 and v_2
D	the index divisor, source units per index point; in the vault balance sheet, pending deposits; in the approval hash, the listing domain tag
t	a round, or a time
I_t	the index at round t (fixed9)
I_{t-1}	the index of the previous print
τ	root tolerance in bps (50)
τ_k	the time of the k-th hourly close in the settlement window, from $\tau_0 = T_h - W$ to $\tau_{n+1} = T_h$
β	circuit band in bps (1,000)

Symbol	Meaning
R	the reference close for the circuit band: the last hourly close at or before 24 hours earlier
t_{open}, t_{close}	a round's `opened_at` and `closes_at`
t_{fetch}, t_{source}	an observation's fetch time and source time
t_{now}	the chain clock at submission
$t_{fetch}^{min}(r)$, $t_{min}^{fetch}(r)$	the earlier of a print's two fetch times, `min_fetched_at`
r, r_{prev}	a print and the finalized print before it; alone, r is also the exposure reserve ratio `vault_reserve_bps`
ℓ, t_{land}	the time an order or deposit landed
g, g_{prev}	a NAV generation and the one before it
h, T_h	the metric's `halted_at`; alone, h is also the curators' listing approval hash
T, W	the settlement window, 86,400 seconds; in the vault balance sheet, W is the priced withdrawals owed to LPs
I_k, Δt_k	the hourly closes in effect over the settlement window and the intervals they cover
I_0, n	the close in effect at the start of the settlement window, and the number of closes inside it
I_{settle}	the settlement price, the 24-hour TWAP before the halt

Listing

Symbol	Meaning
A	the exchange, proposal, proposer, market and metric addresses in the approval hash
$\|\|$	byte concatenation

Trading and margin

Symbol	Meaning
q	a position's size in index units (fixed9)
q_{long}, q_{short}	a market's total long and short size
q_{side}, q_{order}	a side's total size and an order's size
q_{old}, q_{add}	a position's size before an increase and the size added

Symbol	Meaning
q_c	the quantity a close removes
q_A, q_B, I_A, I_B	the sizes and indices of the two legs of a pair trade
r_A, r_B	the fractional level changes of the two artists in a pair trade
$N(q, P)$	notional of size q at price P, $\lceil q \cdot P / 10^{12} \rceil$ base units
P	fill price; in the approval hash, the two program IDs
P_{entry}, P'_{entry}	entry price, and entry price after an increase
I_{entry}, I_{mark}	the index at entry and the index a position is marked to
$P_{liq,long}$, $P_{liq,short}$	the liquidation price of a long and of a short
s	skew $= N_{long} - N_{short}$
s_{before}, s_{after}	skew before and after a fill
s_{t-1}	skew over the interval since the previous print
S	skew scale
σ	side sign, $+1$ for a long and -1 for a short
m	margin
m_{escrow}	the USDC an order escrows
m_{init}, m_{maint}	initial and maintenance margin requirements
λ, λ_{bps}	leverage, and maximum leverage in bps (30,000)
μ, μ_{bps}	maintenance margin as a fraction and in bps
e, e_{close}	position equity, and the equity of a closed quantity
PnL	a position's profit and loss against the mark
ϕ, ϕ_{liq}	trading fee and liquidation fee in USDC
ϕ_{vault}	the vault's share of a fee
b_{trade}, b_{liq}	trading and liquidation fee rates in bps (10 and 100)
k	keeper fee

Funding and borrow

Symbol	Meaning
f, $\dot{f}$	funding rate and funding velocity
f_t, f_{t-1}, Δf_t	the funding rate after and before a print, and its change
$\bar{f}_t$	the funding rate integrated over the interval between prints
v	funding velocity coefficient, fixed9 per hour squared at $s = S$
Δt	seconds between two synchronized prints
F, F_t, ΔF_t	the cumulative funding index and its step at a print
F_{cp}, F'_{cp}	a position's funding checkpoint, and the checkpoint after an increase
B, B_t, ΔB_t	the cumulative borrow index and its step at a print
B_{cp}	a position's borrow checkpoint
c_F, c_B	a position's accrued funding and borrow charges
$c(q_c)$	the accrued charges of a closed quantity
u, u_{side}	utilization in bps: the higher side's, and one side's
b, b_0, b_1	borrow rate, base rate and slope, fixed9 per hour
r_{borrow}, r_{base}, r_{slope}	the same borrow rate, base and slope
Σ_F, Σ_B, $\Sigma_{\lvert F\rvert}$	a market's funding receivable, borrow receivable and gross funding bases
Φ	net funding and borrow traders owe the vault, `total_charges_usdc`

Vault

Symbol	Meaning
NAV, NAV_g	the vault's net asset value, and its value in generation g
C	the vault's USDC balance
M	reserved margin and order escrow
K	claim balances owed to wallets
Π, Π_{market}	traders' unrealized PnL across all markets, and in one market
V, V_{long}, V_{short}	entry bases, $\sum q_i P_{entry,i}$, of a market's long and short positions

Symbol	Meaning
R_F, R_X	the funding credit reserve and the exposure reserve
Z, Z_g	the share supply, and the supply in generation g
z	shares minted to a deposit or redeemed by a withdrawal
p	the share price in fixed9 USDC