

Whitepaper

OpenProp: a prop AMM with an open quoting seat

The complete design: the mechanism and its formulas, the Solana program and its accounts, the fill checks, settlement against a passive hold, performance-weighted allocation, the reference price, the limits, and the services around the program.

Version 1.0 October 2026 Program wire version 5 Account version 5

Contents

1. Introduction

- 1.1 Prop AMMs and the three parts OpenProp separates
 - 1.2 The problem
 - 1.3 Design principles
 - 1.4 The system at a glance
 - 1.5 Conventions
 - 1.6 Map of the paper
-

2. Market structure

- 2.1 Roles
 - 2.2 Accounts
 - 2.3 How order flow reaches a book
 - 2.4 A book compared with a pool and an order book
 - 2.5 One SOL/USDC pair per deployment
 - 2.6 The registry of books
-

3. The book

- 3.1 Accounts of a book
 - 3.2 The quote ladder
 - 3.3 Prices from mid and offsets
 - 3.4 Retail delta
 - 3.5 How inventory and allocation bound the sizes
 - 3.6 The update_quote instruction
 - 3.7 Landing and freshness
 - 3.8 Hot and cold keys
-

4. The reference price

- 4.1 The RefPrice account
 - 4.2 The post_ref_price instruction
 - 4.3 Verifying the signed updates
 - 4.4 The two-source median
 - 4.5 Confidence and divergence gates
 - 4.6 Reference staleness
 - 4.7 The pair breaker
 - 4.8 Parameters
-

5. The five checks

- 5.1 Order of evaluation
- 5.2 Before the checks: epoch end and the pre-fill mark

- 5.3 Check 1: book state
 - 5.4 Check 2: quote freshness
 - 5.5 Check 3: reference validity
 - 5.6 Check 4: band
 - 5.7 Check 5: inventory skew
 - 5.8 Checks 6 and 7: hourly and epoch loss
 - 5.9 Check 8: pair breaker
 - 5.10 Check 9: slippage
 - 5.11 Equality at the limit
-

6. Swap execution

- 6.1 Instruction and accounts
 - 6.2 The level walk
 - 6.3 Partial fills and the minimum output
 - 6.4 Worked example: the sample ladder
 - 6.5 The two token transfers
 - 6.6 Inventory tracking on the book
 - 6.7 The retail flow class and retail delta
 - 6.8 Compute cost
-

7. The vault and LP shares

- 7.1 Accounts
 - 7.2 Units and valuation
 - 7.3 NAV and share price
 - 7.4 Deposits
 - 7.5 Withdrawals
 - 7.6 Idle and allocated inventory
-

8. Bonds, tiers and allocation

- 8.1 Registration and the bond
 - 8.2 Tiers
 - 8.3 The score
 - 8.4 From score to allocation
 - 8.5 Worked example: three books
-

9. Checkpoints, settlement and fees

- 9.1 Epoch and checkpoint clocks
- 9.2 checkpoint_book
- 9.3 settle_epoch
- 9.4 A worked epoch
- 9.5 The protocol's fee share

10. Parameters and governance

- 10.1 The Config account
 - 10.2 Parameters by purpose
 - 10.3 The immutable envelope
 - 10.4 Proposing and activating a change
 - 10.5 Identities and their rotation
 - 10.6 Initialization
 - 10.7 The deployment's parameter file
-

11. Security design

- 11.1 Custody
 - 11.2 Signer roles
 - 11.3 Replay protection
 - 11.4 The checks on every fill
 - 11.5 Timelocks
 - 11.6 The guardian's halt and the program's breaker
 - 11.7 Compute and account-list bounds
 - 11.8 Reconciliation of tracked inventory
 - 11.9 The indexer and unproven history
-

12. Program architecture

- 12.1 Wire version and dispatch
 - 12.2 Accounts
 - 12.3 Instructions by caller
 - 12.4 Events
 - 12.5 The address lookup table
 - 12.6 Compute budgets
 - 12.7 The shared rules crate
-

13. Infrastructure and integration

- 13.1 Deployment layout
 - 13.2 PostgreSQL
 - 13.3 Indexer
 - 13.4 API
 - 13.5 The web app
 - 13.6 Cranker
 - 13.7 Guardian
 - 13.8 Quoter SDK
 - 13.9 Jupiter Amm adapter
-

Appendix A. Parameters

Appendix B. Instructions

Appendix C. Account layouts

Appendix D. Events

Appendix E. Program errors

Glossary

1. Introduction

OpenProp is a proprietary automated market maker (prop AMM) for SOL/USDC on Solana with an open quoting seat. Independent quoters post a USDC bond, each receives its own segregated book, and each prices that book by streaming a five-level quote ladder to the chain. The SOL and USDC that the books trade come from one shared LP vault. The program checks a fixed set of risk limits on every fill, so a quoter decides prices but never holds custody of the inventory it prices and has no instruction that moves it.

This chapter defines OpenProp, describes the two groups it is built for, states the four principles the design follows, gives an overview of the parties and the lifecycle of capital, fixes the units used throughout the paper, and maps the remaining chapters.

1.1 Prop AMMs and the three parts OpenProp separates

A prop AMM is a venue in which one operator writes its own prices into a small on-chain account and takers fill against those prices, normally through an aggregator route. On Solana these venues now fill most SOL–stablecoin trades. Estimates put them between about one third of Solana on-chain volume and 62% of Solana spot DEX volume in the first quarter of 2026, and more than 90% of Jupiter's SOL–stablecoin flow is routed to them. Each existing venue is a closed book run by a single operator on its own capital: HumidiFi, BisonFi, Tessera V, SolFi, Obric, ZeroFi, GoonFi and others. None of them accepts outside LP deposits.

A prop AMM bundles a venue, an inventory and a pricing function. OpenProp assigns each of these to a different party:

Part	What it consists of	Who supplies it in OpenProp
Venue	An on-chain program, an aggregator integration, a fast path for landing quote updates	Shared by every quoter: the single openprop program, an adapter for Jupiter's Amm interface that exposes each book as a market, and a quoter SDK that sends updates down the landing paths a quoter plugs in — BAM's Maker Priority Plugin, the Block Engine, the TPU
Inventory	The SOL and USDC that fill trades	LPs, through the pair's LpVault; the vault allocates inventory to books at each epoch's settlement, and each quoter's USDC bond takes the first loss on its book
Pricing	The quote ladder and the cadence at which it moves	Competing quoters, each writing to its own Quote account through <code>update_quote</code>

The program sits between these parties. It holds every token account, accepts prices only from a book's registered hot keys, and executes a swap only when the book, its quote and the pair's reference price pass every check at the slot of execution.

1.2 The problem

Quoters without a venue

A mid-tier quantitative or market-making firm can have competitive pricing and still have no venue. Running a prop AMM of its own requires an on-chain program, aggregator approval, landing infrastructure and

inventory. Jupiter's DEX integration is permissioned and assesses audits, traction and team, and Jito's Maker Priority Plugin (MPP) is open only to selected makers. That is months of work before the first fill.

Independent quants and AI-agent teams face a second barrier: they have no way to demonstrate skill on live flow without committing all of their own capital, and no track record to show an aggregator's review.

OpenProp gives such a quoter a venue, an aggregator adapter and landing code from the first epoch, and LP capital that scales with its own bond and its measured record. A new quoter registers permissionlessly with `register_quoter`, starts at tier 0 with an allocation of at most 1× its bond, and earns leverage of 5× (tier 1) and 15× (tier 2) through epochs of positive performance.

LPs without access to prop flow

SOL/USDC flow has moved to prop AMMs, and the flow that remains in public pools is the less profitable part of it. In a 12-month sample, passive SOL/USDC AMM makers had gross 2-second markouts of -0.22 bps, against +0.37 bps for prop AMMs. Prop AMMs take no deposits, so a passive LP has no direct way to hold the side of the market that prop AMMs occupy.

OpenProp lets an LP deposit SOL and USDC into a vault whose inventory is traded by bonded quoters. The LP carries SOL direction risk as in any SOL/USDC pool. Trading losses are charged to the quoter's bond before they reach the vault.

The design target

The largest operators are not short of capital: HumidiFi reported routing \$0.5–1B a day on about \$8M of inventory, and they keep their full edge. OpenProp is therefore built as the distribution and bounded-capital layer for quoters that lack a venue, capital, or both. Aggregators route each trade on price, so a book receives the trades on which its quote is the best available; a quoter does not need to beat the top of the SOL/USDC book on average to be filled.

1.3 Design principles

Quoters price

Pricing runs off-chain on the quoter's own servers, and only its output reaches the chain. A quoter's hot key signs `update_quote`, which replaces the book's entire ladder in one write to the 304-byte Quote account: a mid price, five bid levels and five ask levels, each an offset in basis points from the mid and a size in lamports, plus a retail delta. The runtime verifies the transaction signature, so authenticating the update costs the program no compute; the measured cost of the instruction is 178 CU against a 200 CU target. `update_quote` writes only the Quote account and has no path to any token account. The quoter pays its own update fees and sets its own update cadence.

The program enforces limits on every fill

`swap_exact_in` evaluates five checks before any level fills: the quote is fresh (no older than `quoteStalenessSlots`), its mid lies inside a band around the reference price, the book's inventory skew stays inside its limit after the fill, the book's hourly and epoch loss windows are not breached, and the pair's circuit breaker is not tripped. A fill that fails a check does not happen, and the rejection is atomic: no state is

written and no token moves. Because a limit can never be crossed and punished afterwards, OpenProp has no slashing for rule violations. The same rules crate (`openprop-rules`) is compiled into the program and to WASM, so aggregators, the indexer and the browser evaluate a swap with the code that executes it.

LP capital sits in one vault

Each pair has one `LpVault` with its own SOL and USDC token accounts and a classic SPL share mint (OPLP, 6 decimals). Deposits are proportional to the vault's current free basket and mint shares at net asset value. At each epoch's settlement, `settle_epoch` recalls every book, values it against the reference price, and reallocates the free basket to books by a deterministic score-based rule: score from checkpoint excess returns, a 10% exploration pool for tier-0 books, at most 35% of the vault per quoter and at most a 25% change per epoch. Withdrawal requests are served at the next settlement, within at most 2 epochs.

Bonds take first loss

Each quoter posts a USDC bond held in a program-owned token account. A book's performance is measured as excess over hold, the value of its inventory now minus the value of its starting inventory, both marked at the current reference price:

$$X = \left(\frac{B \cdot P}{10^9} + Q \right) - \left(\frac{B_0 \cdot P}{10^9} + Q_0 \right)$$

where:

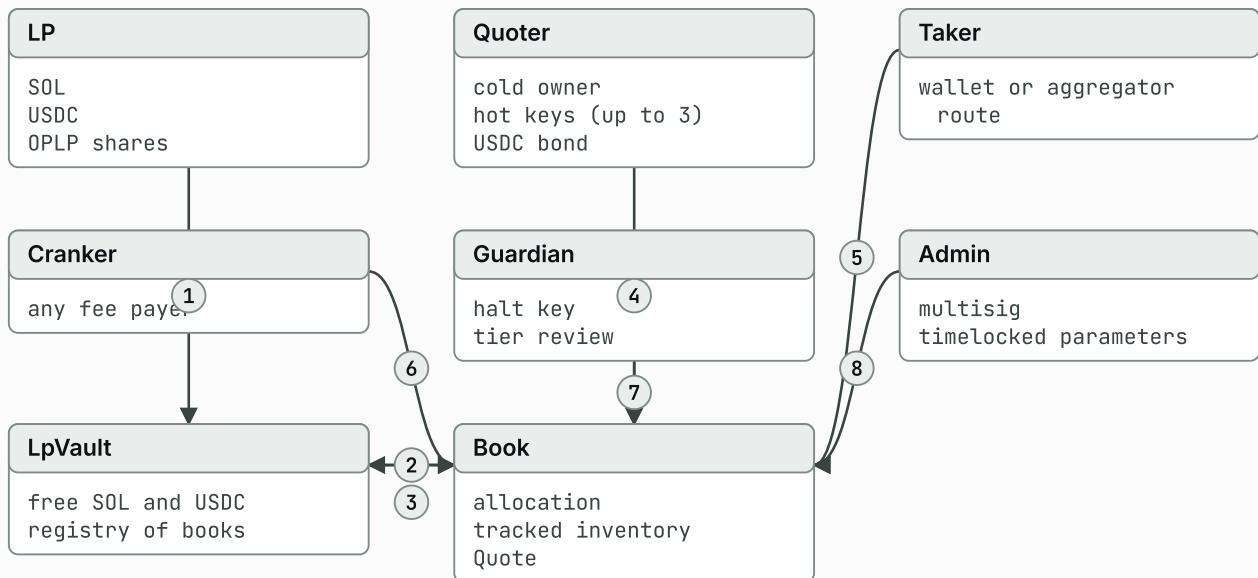
Symbol	Meaning
X	Excess over hold, in USDC micros
B, Q	The book's tracked SOL (lamports) and USDC (micros) now
B_0, Q_0	The book's starting inventory, set at the last settlement
P	The reference price, in USDC micros per SOL

A quoter is therefore judged on spread capture and inventory management, not on the direction of SOL. At settlement, negative excess is charged to the bond first and only the remainder falls on LPs. Positive excess above the book's high-water mark is split 50% to LPs, 40% to the quoter and 10% to the protocol, and the quoter's share is credited to its bond, which adds first-loss capacity. Allocation is capped at tier leverage times bond. At tier 2 the bond is therefore at least 1/15 (6.7%) of allocation, more than the 2% epoch loss at which the book stops filling.

The protocol's share burns \$OPENPROP

The protocol keeps 10% of positive excess above each book's high-water mark, and that share has one use: it buys \$OPENPROP, the protocol's token, on the market and burns it. \$OPENPROP has a fixed supply of 1,000,000,000, all of it sold in one public launch on equal terms. Chapter 9 gives the flow.

1.4 The system at a glance



- ① LP → LpVault: deposits SOL and USDC, receives OPLP
- ② LpVault → Book: allocation at each settlement
- ③ Book → LpVault: inventory and settled excess at each settlement
- ④ Quoter → Book: prices it with update_quote, bond takes first loss
- ⑤ Taker → Book: swap_exact_in
- ⑥ Cranker → Book: post_ref_price, checkpoint_book, settle_epoch
- ⑦ Guardian → Book: halt_pair stops fills in every book
- ⑧ Admin → Book: risk limits through set_params

Figure 1.1. The six parties and the two accounts that carry capital. Only swap_exact_in and settle_epoch move book inventory; no party's key can transfer it directly.

Capital moves through a fixed cycle of program instructions:

Step	Instruction	Signed by	Effect
Deposit	deposit	LP	SOL and USDC enter the vault; OPLP shares are minted
Allocate	settle_epoch	Cranker (any fee payer)	Books are recalled and refilled from the free basket by score
Quote	update_quote	Quoter hot key	The book's ladder is replaced
Swap	swap_exact_in	Taker	A fill against the ladder, after every check passes
Checkpoint	checkpoint_book	Anyone	The book is marked to the reference price, its score sums updated, and it halts on a loss limit
Settle	settle_epoch	Cranker (any fee payer)	Excess is split or charged to the bond; allocations are reweighted; withdrawals are reserved

An epoch lasts epochSlots slots, 432,000 slots or about 24 hours at 200 ms slots, and checkpoints run about every 1,500 slots.

1.5 Conventions

All on-chain integers are little-endian, and every account and instruction described in this paper belongs to wire and account version 6: each instruction begins with the bytes [5, tag] and each account begins with version byte 5 and a kind byte.

Quantity	Unit	Scale
SOL amounts	lamports	1 SOL = 10^9 lamports (base mint, 9 decimals)
USDC amounts and values	micros	1 USDC = 10^6 micros (quote mint, 6 decimals)
Prices	micros per SOL	200.000000 USDC/SOL is stored as 200,000,000
LP shares	share atoms	OPLP, 6 decimals
Offsets, bands and limits	bps	1 bps = 0.01% = 10^{-4}
Fill-quality metrics	BpsE4	basis points $\times 10^4$
Scores	E6	value $\times 10^6$
Scoring returns	E9	return $\times 10^9$
Time	slots	200 ms per slot on mainnet-beta since epoch 1053
Settlement period	OpenProp epoch	epochSlots slots, 432,000 in the v0 parameter set

Parameter values quoted in this paper are the v0 starting set, for example `quoteStalenessSlots` = 2 slots and `epochLossBps` = 200. Parameters whose values the deployment's parameter file sets at initialization, such as the band width `bandBps` and the skew limit `skewBps`, are described by name and role only. Worked examples use a sample book quoted at a 200.00 USDC mid with levels at 1, 2, 4, 8 and 12 bps of 1, 2, 4, 8 and 16 SOL.

1.6 Map of the paper

Chapter	Content
2. Market structure	The roles and what each signs, the accounts, how order flow reaches books, how a book differs from a pool and an order book, the single pair and the registry of books
3. The book	The five-level ladder, <code>update_quote</code> , price derivation, retail delta, inventory bounds, freshness and the hot/cold key split
4. The reference price	<code>post_ref_price</code> , the Pyth Pro and Switchboard Surge proofs, slot attestation, the two-source midpoint and confidence
5. The five checks	Freshness, band, skew, loss windows and the pair breaker, with their formulas and error codes
6. Swap execution	<code>swap_exact_in</code> step by step: ladder walk, rounding, partial fills at the skew limit, transfers and state updates
7. The vault and LP shares	Deposits, NAV, OPLP issuance, virtual shares, withdrawal requests and claims

Chapter	Content
8. Bonds, tiers and allocation	Bond posting and unbonding, tier promotion and downgrade, scoring and the allocation rule
9. Checkpoints, settlement and fees	checkpoint_book, excess over hold, the high-water mark, first-loss charging, the fee split and profit locks
10. Parameters and governance	The parameter set, the immutable safety envelope, set_params, timelocks and identity rotation
11. Security design	Custody, signer authority per instruction, replay protection, checks and timelocks
12. Program architecture	The Pinocchio program, hot and cold paths, account layouts, events and compute measurements
13. Infrastructure and integration	The cranker, guardian, indexer and API, the quoter SDK, landing paths and the aggregator adapter
Appendices	Parameters, instructions, accounts, errors and the glossary

2. Market structure

This chapter describes who participates in an OpenProp market, what each participant signs, which accounts carry the market's state, and how a taker's order reaches a book. It closes with the two structural facts that bound a deployment: it serves exactly one SOL/USDC pair, and its registry holds at most five books.

2.1 Roles

Six roles interact with the program. Authority is assigned by key: every instruction either names the account whose key must sign it, or requires no signer beyond the transaction fee payer.

Role	Key	Instructions it signs	What it can change	What it cannot do
Quoter, cold owner	The wallet that registered (Quoter.owner)	register_quoter, add_bond, request_unbond, claim_unbond, set_quote_authority, pause_book, resume_book, deregister_quoter	Its bond, its hot keys, the paused state of its own book	Withdraw book or vault assets
Quoter, hot authority	Up to three keys cached in Quote	update_quote	Its own book's Quote account	Write any other account or move any token
LP	Any wallet	deposit, request_withdraw, claim	Its own shares and withdrawal tickets	Direct inventory to a book
Taker	Any wallet, or an aggregator's transfer authority	swap_exact_in	One book's inventory, within the checks	Fill a stale, out-of-band or halted book
Cranker	Any fee payer	post_ref_price, checkpoint_book, settle_epoch	The order in which deterministic state transitions run	Choose prices, allocations or splits
Guardian	Config.guardian; optional Config.guardianHaltSigner	halt_pair, review_tier (guardian only)	Halt the pair; approve a tier-2 review	Resume the pair, change parameters, move funds
Admin	Config.admin, a multisig	initialize_config, initialize_pair, set_params, resume_pair, rotate_identities	Parameters and identities, under a timelock for every loosening	Move vault, book or bond assets

Quoters

A quoter controls two classes of key. The **cold owner** is the wallet that called register_quoter; the Quoter account is a program-derived address seeded by it (["quoter", owner]), so one wallet holds at most one quoter seat. Registration transfers the initial USDC bond, which must be at least minBondQuoteAtoms, and creates in one instruction the Quoter account, its USDC bond token account, the Book, the Quote and the book's SOL and USDC token accounts. The cold owner manages the bond (add_bond, request_unbond,

`claim_unbond`), pauses and resumes its book, installs hot keys, and can deregister a fully unbonded, flat seat.

A **hot authority** signs only `update_quote`. It is the key a quoting engine holds in memory and uses many times a second, so its permissions are reduced to writing one 304-byte account. Chapter 3 describes the split between the two key classes in detail.

LPs

An LP calls `deposit` with a maximum SOL amount, a maximum USDC amount and a minimum share count. The program sizes a basket proportional to the vault's current free SOL/USDC holdings across the vault and all books, consumes the limiting maximum in full, leaves unused tokens in the wallet, and mints OPLP shares against the vault's net asset value. Exits go through `request_withdraw`, which burns liquid shares into pending shares, and `claim`, which pays the reserved basket after the next settlement.

Takers

A taker calls `swap_exact_in` with a side (0 sells SOL, 1 buys SOL), an exact input amount and a signed minimum output. Most takers do not call the program directly: they request a route from an aggregator, and the aggregator's router program invokes `swap_exact_in` by cross-program invocation. The four aggregators that cover nearly all routed Solana volume are Jupiter, DFlow, Titan and OKX.

Cranker

The cranker runs three permissionless instructions. `post_ref_price` stores a new reference price after verifying signed Pyth Pro and Switchboard Surge payloads (Chapter 4). `checkpoint_book` marks one book to the reference price about every 1,500 slots (`checkpointSlots`) and halts it if a loss limit is breached. `settle_epoch` runs once the epoch's last slot has passed and settles every registered book in one transaction. None of these instructions takes a decision from its caller: each computes its result from accounts, the clock and verified oracle proofs, so any fee payer can run them, and a repeated or competing call either produces the same transition or is a no-op (a checkpoint inside its cadence, an already posted oracle update, an already settled epoch). The protocol's cranker service runs all reference-writing jobs under one fenced lease per pair and journals each signed packet before broadcast.

Guardian

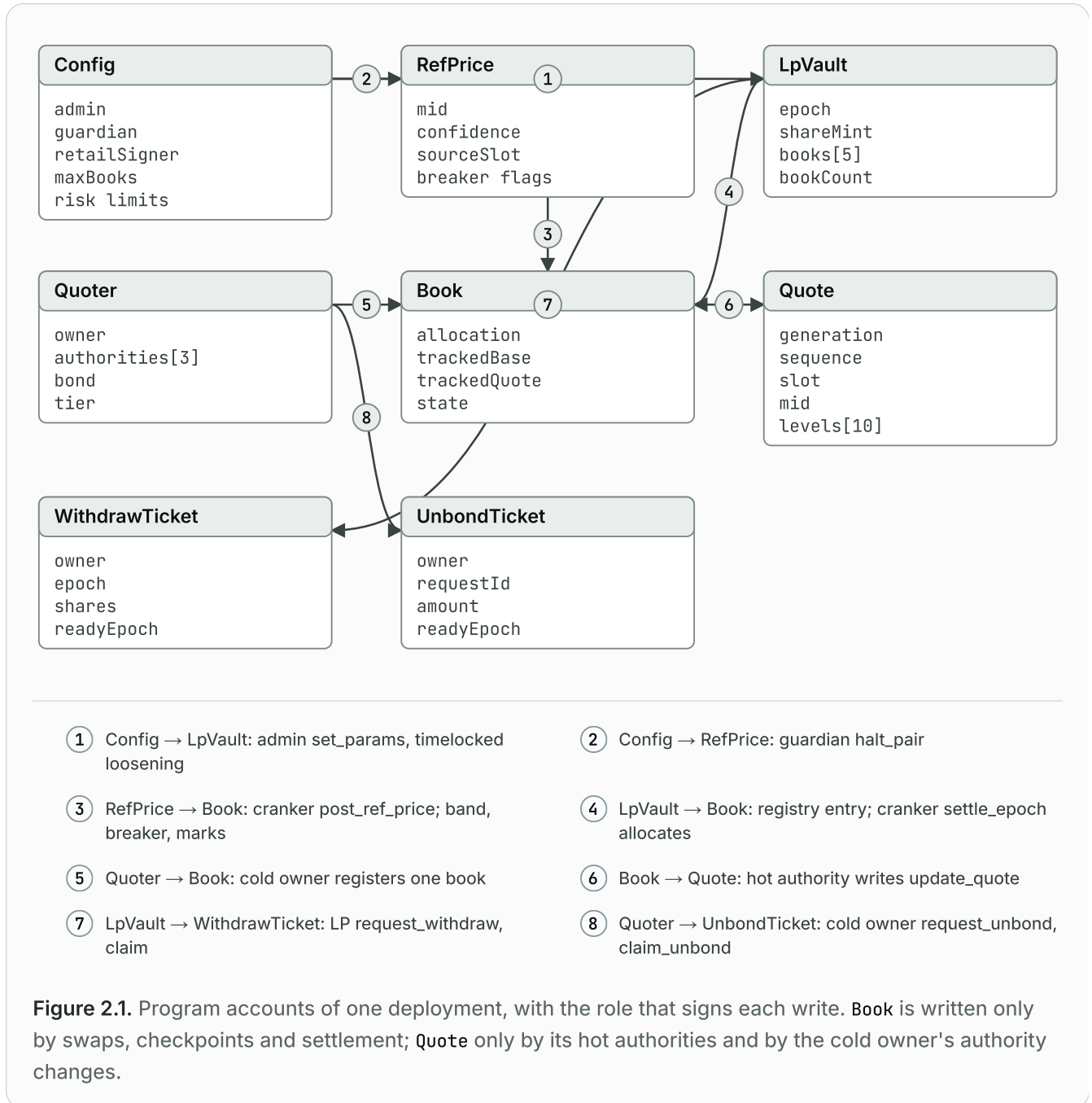
The guardian is a multisig whose only market action is `halt_pair`, which sets the manual-halt flag on the pair's `RefPrice` and so stops fills in every book. An optional second key, `guardianHaltSigner`, may also call `halt_pair` and nothing else; the protocol's guardian bot uses it when an explicitly configured, sustained oracle divergence is observed. The guardian also signs `review_tier`, the v0 review required for tier 2. It cannot resume a halted pair, change parameters, quote or transfer assets.

Admin

The admin is a Squads multisig. It initializes the deployment, proposes parameter changes with `set_params`, resumes a manually halted pair with `resume_pair`, and rotates identities with `rotate_identities`. A change that tightens a limit applies immediately. A change that loosens a limit is stored as pending and can be applied only after both $\text{Clock.slot} \geq s_p + \text{timelockEpochs} \cdot \text{epochSlots}$ and $e_v \geq e_p + \max(w, w') + 1$, where s_p and e_p are the proposal slot and vault epoch, e_v is the current vault epoch, and w, w' are the current and

proposed withdrawEpochs. An LP who disagrees with a loosening therefore always has a full withdrawal cycle in which to exit. No admin instruction transfers vault, book or bond assets.

2.2 Accounts



Every account is a program-derived address (PDA) of the openprop program, and every token account is owned by a PDA, so the program is the only signer that can move tokens out of them.

Account	Seeds	Bytes	Holds
Config	["config"]	1,728	Identities (admin, guardian, oracle signers, retail signer, cranker), mints, the protocol USDC account and the buyback mint, the active and pending parameter sets, the immutable safety envelope
PairConfig	["pair", baseMint, quoteMint]	168	Links from the pair to its Config, RefPrice and LpVault
RefPrice	["ref", pair]	3,776	Reference mid, confidence, source slots, both source prices, breaker state and a 150-slot price history
LpVault	["vault", pair]	408	Epoch, share supply, reserved withdrawal assets, deferred fees and the registry of books
Vault token accounts	["vault_base", pair], ["vault_quote", pair]	165 each	Free SOL and USDC
Share mint	["shares", pair]	82	OPLP, 6 decimals, mint authority LpVault
Quoter	["quoter", owner]	1,304	Cold owner, hot authorities, bond, tier, review flag, trip history, profit-lock tranches
Bond token account	["bond", quoter]	165	The quoter's USDC bond
Book	["book", quoter, pair]	576	Allocation, starting and tracked inventory, excess, high-water mark, loss windows, scoring sums, state
Book token accounts	["book_base", book], ["book_quote", book]	165 each	The book's SOL and USDC
Quote	["quote", book]	304	Generation, sequence, slot, mid, retail delta, ten levels, cached hot authorities
WithdrawTicket	["wd", owner, epoch]	144	One LP's queued withdrawal for one epoch
WithdrawBatch	["withdraw_batch", vault, epoch]	104	The epoch's aggregate withdrawal reservation
UnbondTicket	["unbond", quoter, requestId]	136	One unbond request with paid, haircut and remaining amounts
OperationReceipt	["receipt", owner, requestId]	320	The original payload of a monetary request, for replay-safe retries

Quote is deliberately separate from Book. A quote update write-locks only the small Quote account; a swap reads Quote and writes Book. Quote updates and swaps on the same book therefore never contend for the same write lock, and an update scheduled ahead of swaps in a block reprices the book before they execute.

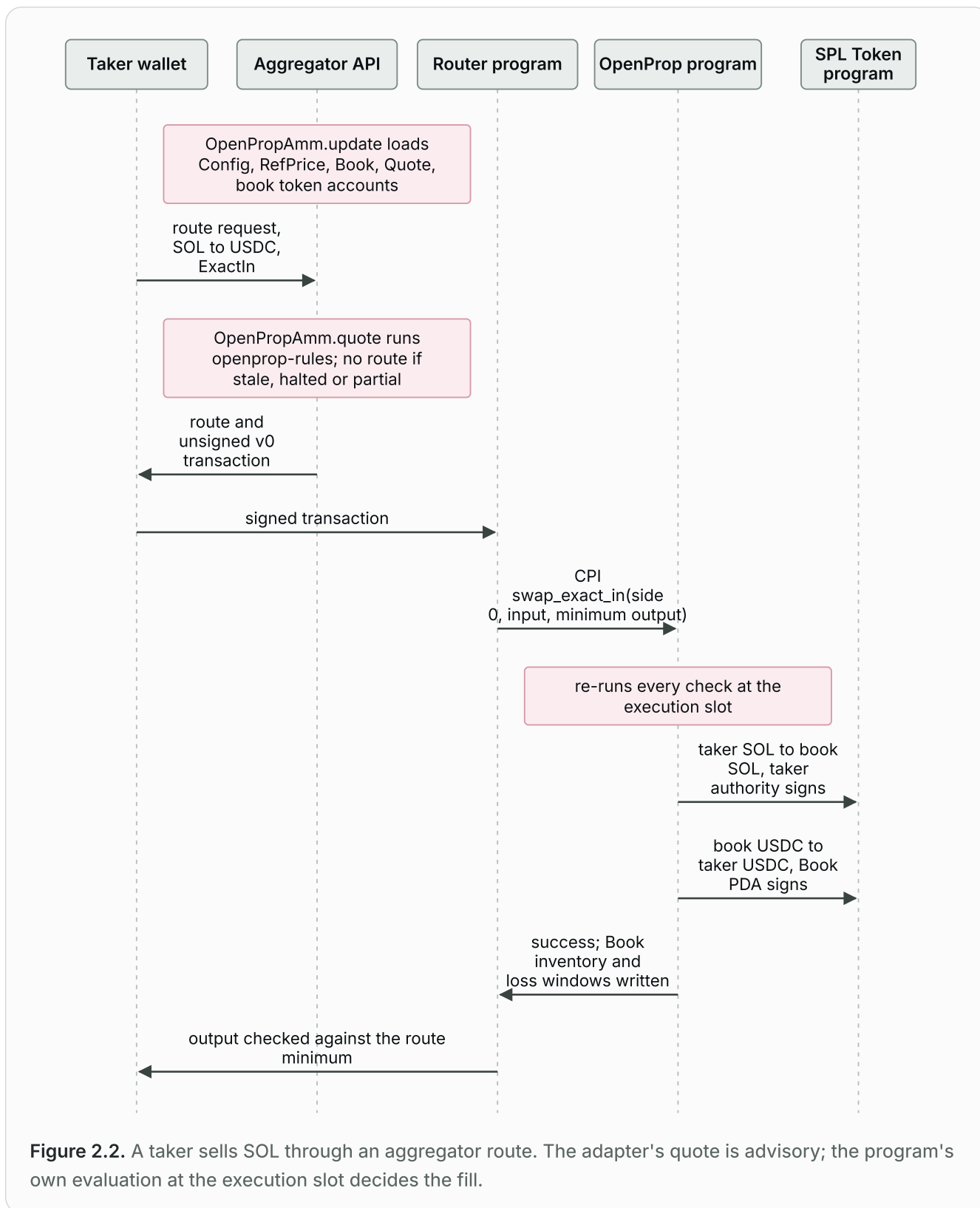
2.3 How order flow reaches a book

Aggregator routing through the Jupiter Amm adapter

An aggregator's routing engine treats each Book as one market. The reference integration is `OpenPropAmm`, an implementation of the `Amm` trait from `jupiter-amm-interface` 0.6.1. The adapter is constructed from a `Book` account; it derives `Config` (`["config"]`), `RefPrice` (`["ref", pair]`) and `Quote` (`["quote", book]`) and reads the book's two token-account addresses from the `Book`. Its `get_accounts_to_update` returns exactly these six accounts.

`update` receives the six accounts from the aggregator's account cache and validates program ownership, exact version, kind and length, the PDA links between them, the mints, the token authorities, and that actual token balances cover the book's tracked inventory. An invalid update clears the previous snapshot, so a market never quotes from stale or mismatched state. `quote` performs no network calls: it evaluates the swap with the same `openprop-rules` crate the program executes, at the slot supplied by the aggregator's shared clock. It returns no route when the quote or reference is stale, the epoch has ended, the book or pair is halted or paused, the ladder is invalid, the reference confidence or source divergence is outside its gate, or a loss limit is breached. It also declines a route that would consume the input only partially at the skew or depth boundary, because an `ExactIn` route must not leave input unspent. The quoted fee is zero; the quoter's spread is contained in the level prices.

`get_swap_and_account metas` returns the ten accounts of `swap_exact_in` in program order, with the router's token transfer authority as the signer. The instruction is encoded with an explicit user-approved minimum output; if the actual output is smaller, the program rejects the whole swap with `SLIPPAGE_EXCEEDED`.



The quote an aggregator computes and the fill the program executes can differ only through state that changed between the two slots. If the book's quote aged past `quoteStalenessSlots`, its mid left the band, or the pair halted in between, the swap fails and no token moves. If the quoter replaced the ladder in between, the swap executes against the new ladder and the signed minimum output protects the taker.

Direct swaps

Any wallet can call `swap_exact_in` without a router. The contract package's `swapExactIn` builder assembles the ten accounts: taker (signer), `Config`, `RefPrice`, `Book` (writable), `Quote`, the book's SOL and USDC token accounts (writable), the taker's SOL and USDC token accounts (writable), and the SPL Token program. A direct caller also receives the program's native partial-fill behaviour: when the skew limit or the book's inventory stops the ladder walk, the program transfers only the consumed input and the corresponding output, provided the output meets the signed minimum.

Flow class

A swap is either standard or retail. A retail swap carries an eleventh account, the key stored in `Config.retailSigner`, and that key must sign the transaction. The designated signer is an aggregator frontend's signing key; Jupiter documents a frontend signer for its retail flow and classifies that flow as non-toxic. On a retail swap each level's offset is reduced by the quote's retail delta (Chapter 3). A zero `retailSigner` disables the class, and an aggregator's fee-mode hint without the configured signature never grants retail pricing.

2.4 A book compared with a pool and an order book

A book is neither a liquidity pool nor a central limit order book. Its price comes from one quoter, its inventory comes from LPs through the vault, and the program checks its risk limits on each fill.

Property	Constant-function pool	Central limit order book	OpenProp book
Who sets the price	A curve over the pool's reserves; it moves only when trades move the reserves	Each maker's resting orders	One quoter, by replacing the whole ladder with <code>update_quote</code>
Price anchor	None	None	The quote's mid must lie within <code>bandBps</code> of the reference price
Capital	Passive LP deposits	Each maker's own balance	LP capital allocated from the vault, priced by the quoter, with the quoter's bond in first loss
Cost of moving the price	None, but it cannot move without a trade	100–300k CU per order-book update on Phoenix	178 CU per ladder replacement, measured
Matching	Along the curve	Price-time priority across all makers	Within one ladder; an aggregator splits a trade across books as across venues
Who absorbs adverse fills	LPs	The maker	The quoter's bond first, then LPs
On-chain risk limits	None beyond the curve	None per maker	Freshness, band, skew, hourly and epoch loss, pair breaker on every fill

The book shares the oracle-centred ladder design of BisonFi's prop AMM: a mid price and per-level offsets, rather than absolute prices, so that a single field moves the whole ladder.

2.5 One SOL/USDC pair per deployment

A deployment of the program serves exactly one pair. Config is a singleton PDA (["config"]) that stores one base mint and one quote mint, and mint identities are immutable: rotate_identities replaces the admin, guardian, oracle, retail, cranker and buyback-mint keys but never the pair's mints or the protocol USDC account. initialize_pair requires the base mint to have 9 decimals and the quote mint 6, and creates PairConfig at ["pair", baseMint, quoteMint] together with the pair's RefPrice, LpVault, vault token accounts and share mint. Every swap, checkpoint and settlement checks that the Book, RefPrice and LpVault it receives belong to that pair. Books and the vault hold SOL as SPL token balances of the base mint; wrapping native SOL is a separate step outside the program.

2.6 The registry of books

LpVault holds the registry: a 160-byte array of five 32-byte book addresses (books) and a counter (bookCount). Config.maxBooks sets how many seats are open and must lie between 1 and 5, the storage capacity of the array.

register_quoter appends the new book at index bookCount and increments the counter. When bookCount has reached maxBooks it fails with REGISTRY_FULL (6038), "The vault has no free quoter seats. A quoter must deregister before another can register." deregister_quoter frees a seat. It requires the seat to be flat: zero bond, zero pending unbond, zero locked profit, zero allocation, zero tracked inventory and zero deferred fees, otherwise it fails with BOND_IN_USE (6021). It removes the book from the array, shifts the later entries down so registry order is preserved, clears the hot authorities and pauses the book with reason DEREGISTERED. The Quoter and Book accounts remain as history.

Registry order is canonical. deposit and settle_epoch must receive every registered book, with its token accounts, Quoter and bond account, in registry order; the program checks each entry against the array. A deposit therefore prices shares against the inventory of every book, and a settlement cannot omit a book's loss or liability. The account count grows by five per book:

Instruction	Fixed accounts	Per registered book	With five books
deposit	13	5 (book, book SOL, book USDC, quoter, bond)	38
settle_epoch	10	5 (quoter, bond, book, book SOL, book USDC)	35

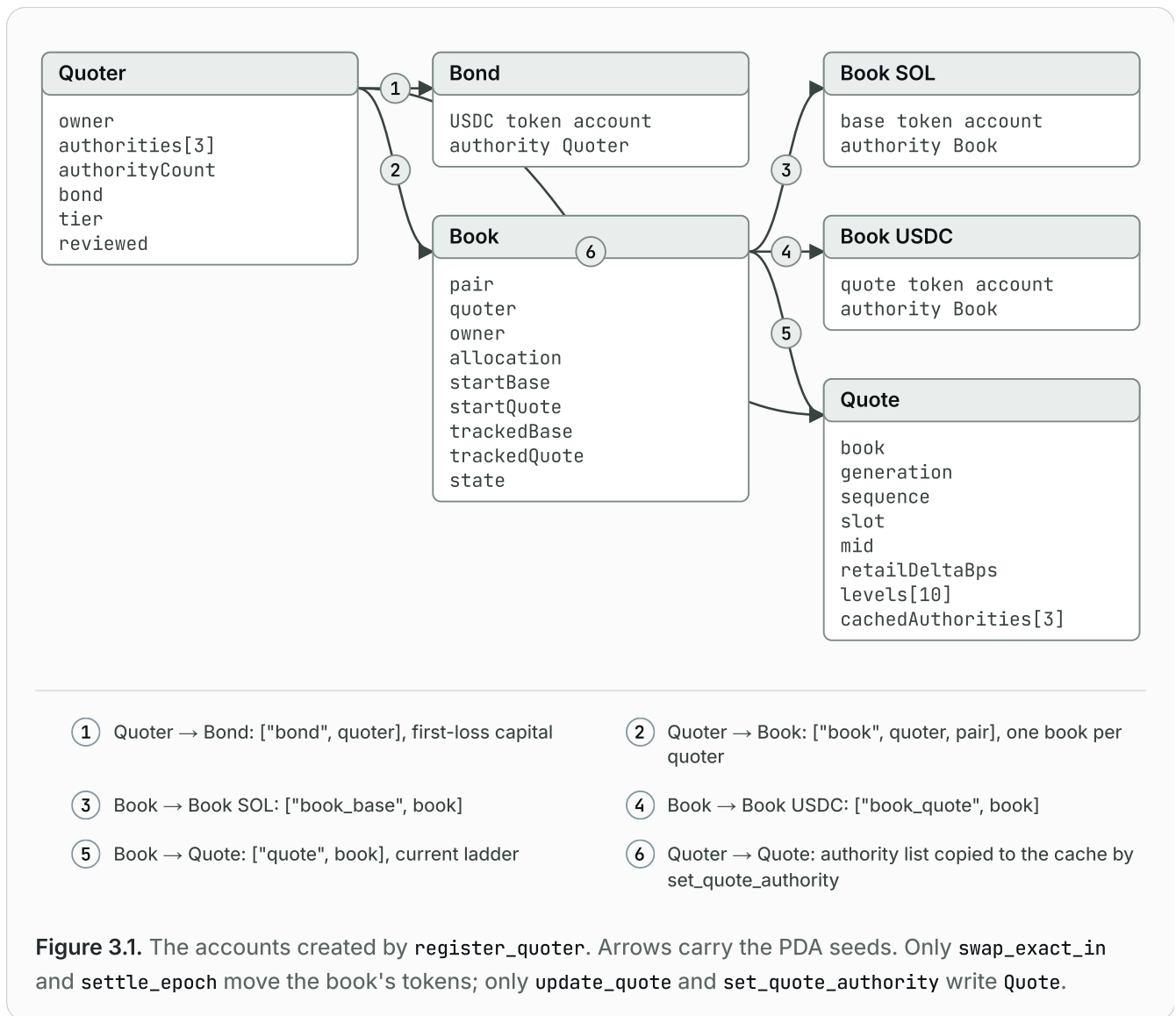
At five books both instructions are sent as v0 transactions with an address lookup table.

Each swap write-locks its Book. Solana caps the compute that transactions writing one account can use in a block at 12M CU; at about 40k CU per swap this allows roughly 300 fills per book per block. Measured swaps cost between 17,934 CU (a representative three-level swap) and 37,497 CU (a skew-limited partial fill on a large book). Depth spread across several books therefore also spreads this per-account capacity.

3. The book

A book is one quoter's segregated market. It holds its own SOL and USDC, filled from the LP vault at each settlement, and it is priced by one account, the Quote, which the quoter's hot keys overwrite with a complete five-level ladder on every update. This chapter covers the ladder and its prices, the limits that bound what it can fill, the `update_quote` instruction, freshness, and the split between the quoter's cold and hot keys.

3.1 Accounts of a book



The Quoter holds the authoritative list of hot keys; Quote holds a cache of the same list so that `update_quote` can authenticate its signer by reading only the account it writes. The Book records the book's state byte: 0 live, 1 halted by a loss limit, 2 paused by its owner (or deregistered).

3.2 The quote ladder

A quote is a mid price and two ladders of five levels. Each level is an offset from the mid in whole basis points and a size in lamports. Bid levels are the prices at which the book buys SOL from a taker who sells; ask levels are the prices at which it sells SOL to a taker who buys. A retail delta, also in basis points, tightens every offset for retail-class swaps (Section 3.4).

The 304-byte Quote account stores the ladder in fixed positions:

Offset	Field	Type	Written by
0	version (5), kind (6), bump, flags, reason	5 × u8	register_quoter
8	book	public key	register_quoter
40	sequence	u64	update_quote; reset to 0 by set_quote_authority
48	slot	u64	update_quote, from the Clock sysvar; reset to 0 by set_quote_authority
56	mid	u64, micros per SOL	update_quote; reset to 0 by set_quote_authority
64	retailDeltaBps	u32	update_quote
72	levels	10 × (u32 offset, u64 size), 120 bytes	update_quote
192	cachedAuthorityCount	u8	set_quote_authority
200	cachedAuthorities	3 × public key	set_quote_authority
296	generation	u64	set_quote_authority

Within `levels`, entries 0 to 4 are bid levels 1 to 5 and entries 5 to 9 are ask levels 1 to 5. Each entry is 12 bytes: the offset as a u32 at +0 and the size as a u64 at +4.

`update_quote` stores the ladder without inspecting it. Every swap validates the ladder it is about to walk, and rejects the swap with `QUOTE_LADDER_INVALID` (6034) unless all of the following hold:

Rule	Condition
Mid present	<code>mid > 0</code>
Retail delta bounded	<code>retailDeltaBps ≤ retailDeltaLimitBps</code>
Offsets below 100%	every offset < 10,000 bps
Offsets monotone	within each side, no offset is smaller than the offset of the level before it
Sizes bounded	every size ≤ <code>maxLevelBaseAtoms</code>

A level with size 0 is valid and is skipped by the walk, so a quoter can post fewer than five levels on a side, or withdraw one side entirely, by zeroing sizes. Deferring validation to the fill keeps the update path to a

fixed sequence of comparisons and copies, and the swap checks the ladder against the configuration that is current at execution.

3.3 Prices from mid and offsets

The program derives each level's price from the mid and the level's offset at fill time:

$$p_i^{\text{bid}} = \left\lfloor \frac{m(10^4 - o_i)}{10^4} \right\rfloor, \quad p_i^{\text{ask}} = \left\lceil \frac{m(10^4 + o_i)}{10^4} \right\rceil$$

where:

Symbol **Meaning**

m Quote.mid, in USDC micros per SOL

o_i The effective offset of level i in bps: the stored offset for a standard swap, or the retail offset of Section 3.4

$p_i^{\text{bid}}, p_i^{\text{ask}}$ The price of level i , in USDC micros per SOL

A level fills in whole lamports. When a taker sells q lamports into bid level i it receives $\lfloor q p_i^{\text{bid}} / 10^9 \rfloor$ micros. When a taker buys with U micros remaining, ask level i delivers

$$q_i = \min \left(\left\lfloor \frac{U \cdot 10^9}{p_i^{\text{ask}}} \right\rfloor, s_i, B \right) \quad \text{for a payment of} \quad \left\lceil \frac{q_i p_i^{\text{ask}}}{10^9} \right\rceil$$

where s_i is the level's size in lamports and B is the book's remaining tracked SOL. Rounding favours the book by at most one micro per level: a seller's proceeds round down and a buyer's payment rounds up.

Worked price table

The sample ladder has a mid of 200.000000 USDC (200,000,000 micros), offsets of 1, 2, 4, 8 and 12 bps on both sides, sizes of 1, 2, 4, 8 and 16 SOL, and a retail delta of 1 bps.

Level	Offset (bps)	Size (SOL)	Size (lamports)	Bid (USDC)	Ask (USDC)	Retail offset (bps)	Retail bid	Retail ask
1	1	1	1,000,000,000	199.980000	200.020000	0	200.000000	200.000000
2	2	2	2,000,000,000	199.960000	200.040000	1	199.980000	200.020000
3	4	4	4,000,000,000	199.920000	200.080000	3	199.940000	200.060000
4	8	8	8,000,000,000	199.840000	200.160000	7	199.860000	200.140000
5	12	16	16,000,000,000	199.760000	200.240000	11	199.780000	200.220000

For level 1, $p_1^{\text{bid}} = \lfloor 200,000,000 \times 9,999 / 10,000 \rfloor = 199,980,000$ micros. Because sizes double from level to level, the depth-weighted average offset of the full side is $(1 \cdot 1 + 2 \cdot 2 + 4 \cdot 4 + 8 \cdot 8 + 16 \cdot 12) / 31 = 277 / 31 \approx 8.94$ bps.

Levels swept	SOL	Taker receives selling into bids (USDC)	Average bid	Taker pays buying asks (USDC)	Average ask	Average distance from mid
1	1	199.98	199.980000	200.02	200.020000	1.00 bps
1-2	3	599.90	199.966667	600.10	200.033333	1.67 bps
1-3	7	1,399.58	199.940000	1,400.42	200.060000	3.00 bps
1-4	15	2,998.30	199.886667	3,001.70	200.113333	5.67 bps
1-5	31	6,194.46	199.821290	6,205.54	200.178710	8.94 bps

Worked fills

These two fills are the shared rules crate's own test vectors.

A taker sells 2.5 SOL (2,500,000,000 lamports) with a signed minimum of 499.90 USDC. Level 1 takes 1 SOL at 199.980000 for 199,980,000 micros; level 2 takes the remaining 1.5 SOL at 199.960000 for 299,940,000 micros. The taker receives 499,920,000 micros (499.92 USDC), an average of 199.968, 1.6 bps below the mid.

A taker buys with 300.04 USDC (300,040,000 micros). At level 1, $\lfloor 300,040,000 \times 10^9 / 200,020,000 \rfloor$ exceeds the level's 1 SOL, so the level fills in full for 200,020,000 micros. At level 2, the remaining 100,020,000 micros buy $\lfloor 100,020,000 \times 10^9 / 200,040,000 \rfloor = 500,000,000$ lamports for exactly 100,020,000 micros. The taker receives 1.5 SOL for the full input.

3.4 Retail delta

For a retail swap, the one co-signed by the key in `Config.retailSigner`, every level's offset is reduced by the quote's retail delta δ , saturating at zero:

$$o_i^{\text{retail}} = \max(o_i - \delta, 0)$$

where o_i is the stored offset of level i and δ is `Quote.retailDeltaBps`. A positive delta tightens the spread on both sides for flow the quoter treats as non-toxic. In the sample ladder a 1 bps delta moves level 1 on both sides to the mid itself and every other level 1 bps closer to it, as the retail columns of the table above show. The quoter chooses δ on every update, and $\delta = 0$ prices both classes identically. `retailDeltaLimitBps`, set in `Config`, bounds it; a quote whose delta exceeds the limit cannot fill in either class.

3.5 How inventory and allocation bound the sizes

The sizes a quoter posts are upper bounds. The amount a swap actually fills is the smallest of four limits.

Tracked inventory. At each settlement `settle_epoch` gives the book a basket from the vault's free SOL and USDC in proportion to its target allocation, so every book starts the epoch with the vault's SOL:USDC ratio. The basket is recorded as both the starting inventory (`startBase`, `startQuote`) and the tracked inventory (`trackedBase`, `trackedQuote`), and its value at the settlement's reference price becomes `allocation`. Only swaps and settlement change tracked inventory; tokens sent to a book account by anyone else are excluded from it and swept to the vault at settlement. A buy can take no more SOL than `trackedBase`, and a

sell can be paid no more USDC than trackedQuote; when either runs out, the walk stops and the swap fills partially with INSUFFICIENT_LIQUIDITY as the stop reason.

Level cap. Each level's size is at most maxLevelBaseAtoms, which bounds the gap a single level can expose between quote updates.

Skew. After each level the program measures the inventory skew at the reference price P :

$$\sigma(B, Q) = \frac{|V_B - Q|}{V_B + Q}, \quad V_B = \left\lfloor \frac{B \cdot P}{10^9} \right\rfloor$$

where:

Symbol	Meaning
B, Q	Tracked SOL (lamports) and USDC (micros) after the fill
V_B	The SOL inventory's value in micros at the reference price
σ	Skew, 0 for a book balanced by value and 1 for a book holding one asset

A fill is admissible while $10^4 \cdot \sigma \leq \text{skewBps}$. A book that starts a swap already beyond the limit may fill only while its skew stays at or below the skew it started the swap with. When a level would cross the limit, the program finds the largest quantity at that level that stays inside it, fills that amount and stops; Chapters 5 and 6 give the check and the search.

Loss windows. The book's hourly and epoch losses, marked at the reference price, must stay within fractions of allocation:

$$10^4 \cdot L_h \leq A \cdot \text{hourlyLossBps}, \quad 10^4 \cdot L_e \leq A \cdot \text{epochLossBps}$$

where A is Book.allocation in micros, $L_e = \max(-X, 0)$ is the loss against the starting inventory, X is the book's excess over hold, and L_h is the fall in X since the start of the current hourly window of hourlyWindowSlots slots. A swap whose post-fill loss would exceed either limit is rejected.

Worked bounds on the sample book

The sample book holds 900 SOL and 180,000 USDC, worth 360,000 USDC at a 200.00 reference, so its allocation is 360,000 USDC and its starting skew is 0. With the v0 limits of 50 bps hourly and 200 bps per epoch, the book stops filling once its hourly loss exceeds 1,800 USDC or its epoch loss exceeds 7,200 USDC; a loss exactly at a limit still fills.

Swap	Book after (SOL / USDC)	Value at 200.00	Excess	Skew after
Taker sells 2.5 SOL	902.5 / 179,500.08	360,000.08	+0.08	27.8 bps
Taker sells 31 SOL into all bids	931 / 173,805.54	360,005.54	+5.54	344.3 bps
Taker buys all 31 SOL of asks	869 / 186,205.54	360,005.54	+5.54	344.6 bps

With the mid equal to the reference price, each fill adds the spread it captured to excess. A full sweep of one side uses 3.4% of the book's USDC and 3.4% of its SOL, so on this book the ladder sizes and the skew

limit, not inventory, decide how much a sweep can fill: the sweep completes only if skewBps is at least 345. Losses arise afterwards, when the reference moves against the inventory the book has taken on, or at fill time when the quote's mid sits on the adverse side of the reference within the band.

3.6 The update_quote instruction

update_quote (tag 0) takes three accounts: the hot authority (signer, fee payer), the Quote (writable) and the Clock sysvar. Its instruction data is 150 bytes, or 151 with the authority hint:

Bytes	Field	Type
0–1	Header [6, 0]	2 × u8
2–9	generation	u64
10–17	sequence	u64
18–25	mid	u64, micros per SOL
26–145	Ten levels: bids 1–5, then asks 1–5, each offset u32 and size u64	120 bytes
146–149	retailDeltaBps	u32
150	Authority index, optional	u8

The program performs these checks in order and fails on the first that does not hold:

Step	Check	Error
1	Exactly three accounts; body of 148 or 149 bytes	INVALID_INSTRUCTION (6000)
2	The hot authority signed	UNAUTHORIZED (6002)
3	Quote is owned by the program, is 304 bytes and carries version 6, kind 6	INVALID_ACCOUNT (6001)
4	The signer equals the cached authority at the hinted index, or, without a hint, one of the cached authorities	UNAUTHORIZED (6002)
5	generation equals Quote.generation	STALE_SEQUENCE (6014)
6	sequence is strictly greater than Quote.sequence	STALE_SEQUENCE (6014)
7	The third account is the Clock sysvar with 40 bytes of data	INVALID_ACCOUNT (6001)

It then writes sequence, slot (the current Clock.slot), mid, retailDeltaBps and the 120 level bytes. It makes no cross-program invocation, emits no log and allocates no memory. The strictly increasing sequence makes updates idempotent and ordered: a delayed or duplicated packet with a sequence at or below the stored one fails, so an older ladder can never overwrite a newer one, whatever path each packet took.

The hinted compute path

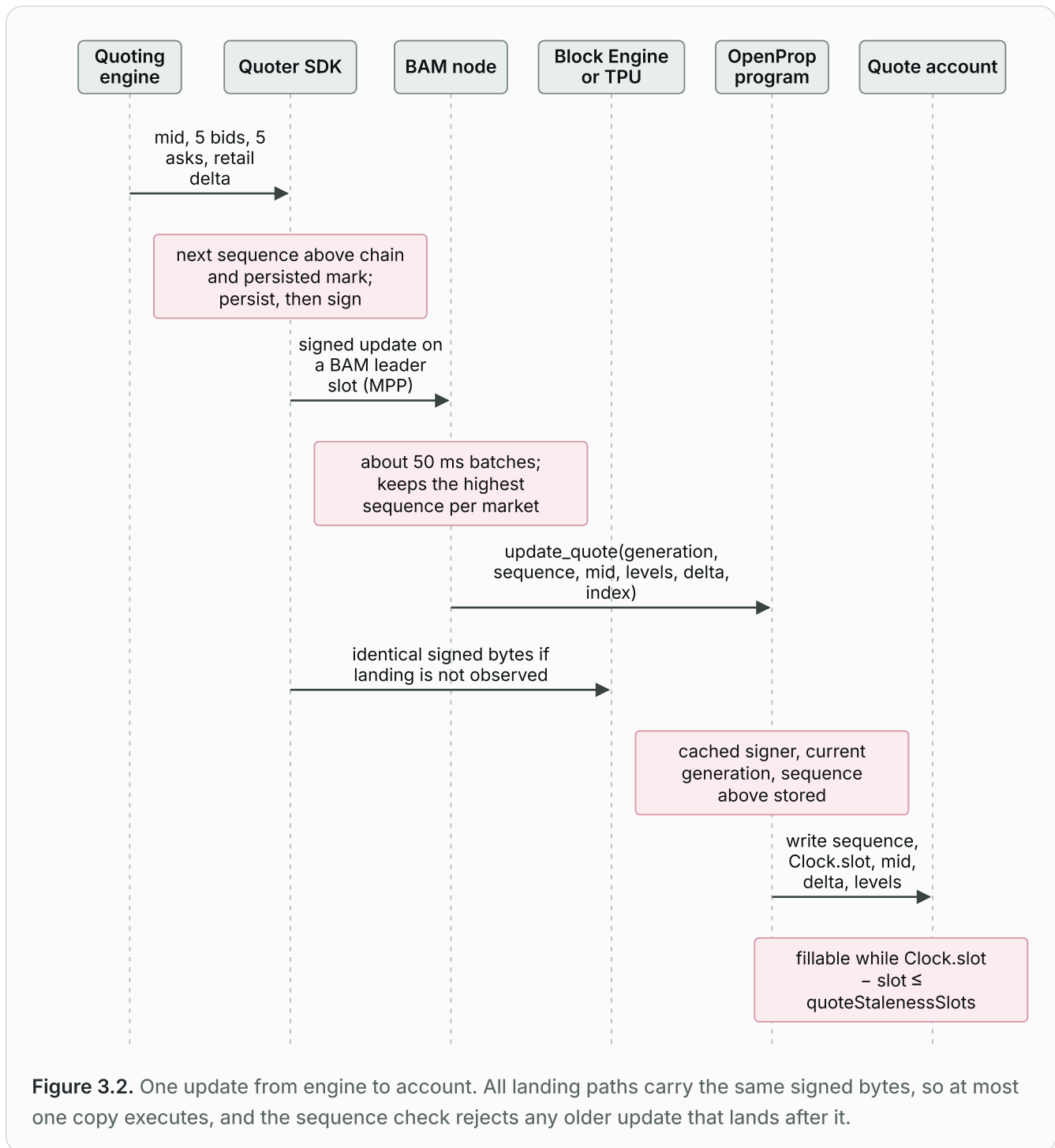
When the loader passes exactly three accounts, the program's entrypoint deserializes only three account slots and, if the instruction data is 150 or 151 bytes beginning with `[5, 0]`, calls the update handler directly; every other instruction takes the general path. The Clock is read from the sysvar account after its key and length are checked.

The optional final byte names the position of the signing key in the cached authority list. With the hint, the program compares the signer against one cached key, and the cost is the same for every position. Without it, the program compares the signer against the cached keys in order. Measured on the release artifact (LiteSVM 0.6.1 executing the SBF program, `cu-measurements.json`):

Path	Signing key	Measured CU	Target
Hinted	Any of three positions	178	200
Scan	The only key	170	200
Scan	First of three	172	200
Scan	Middle of three	177	200
Scan	Last of three	183	200

A quote-update transaction contains a compute-unit limit, a compute-unit price and the single `update_quote` instruction, signed by the hot key as sole signer and fee payer; it is about 300 bytes, well under the 1,232-byte packet limit. The quoter pays every update's fees, so the update cadence is set by the quoter's own engine.

3.7 Landing and freshness



The SDK's `land` sends one signed update down an ordered list of paths: Jito BAM's Maker Priority Plugin on a known BAM leader slot when the quoter is enrolled, otherwise the Block Engine, then the TPU. The quoter supplies the network client for each path (an enrolled sender for MPP, a Block Engine JSON-RPC client, a TPU client); the SDK formats the packets and keeps the order. The reference quoter in the repository sends through a standard RPC endpoint. MPP collects maker updates in batches of about 50 ms, deduplicates them by sequence number and places the latest update first in the batch, so on BAM slots a book's newest ladder is the one that executes and older copies are dropped before they reach the program. A submission acknowledgement does not stop the fallback; only an observed landing does.

Freshness is measured in slots by the program's own clock. The `slot` field is written from `clock.slot` when the update executes, not supplied by the quoter, so the age of a quote is the time since it landed:

$$a = \text{Clock.slot} - \text{Quote.slot}, \quad \text{fill requires } a \leq \text{quoteStalenessSlots}$$

With the v0 value of 2 slots, a quote that lands in slot s can fill in slots s , $s + 1$ and $s + 2$, and every swap against it in slot $s + 3$ or later fails with QUOTE_STALE (6004) until the next update lands. At 200 ms slots that is at most 400 ms after the landing slot. Staleness uses Clock.slot and never Clock.unix_timestamp, which the block leader sets.

3.8 Hot and cold keys

The cold owner and the hot authorities have disjoint powers. The cold owner can manage the bond, pause and resume the book, and replace the hot keys, but no cold instruction writes a ladder. A hot authority can write its book's ladder and nothing else. A leaked hot key therefore exposes at most bad quotes, which every fill still checks against the band, skew and loss limits, and the cold owner can revoke it in one transaction.

Up to three hot authorities

set_quote_authority (tag 11) takes the cold owner (signer), the Quoter and the Quote (both writable), and then one signer account for each newly added key. Its data is a count from 0 to 3 followed by that many public keys, and optionally a trailing byte 1 that requests a full sequence reset. The program enforces:

Rule	Error
At most three keys	AUTHORITY_LIMIT (6022)
Keys nonzero and distinct	INVALID_AUTHORITY_CHANGE (6032)
The cold owner signed; the quoter is registered	UNAUTHORIZED (6002)
Every key not already in the list co-signs, in list order	UNAUTHORIZED (6002)
With a reset, no new key appears in the current list	INVALID_AUTHORITY_CHANGE (6032)

Retained keys and removals need only the cold owner. The co-signature exists because swaps whose taker is the cold owner or a current hot authority are known self-flow: they still change the book's inventory, losses and excess, but they add nothing to its scoring returns or scored turnover. Without the new key's consent a cold owner could list an unrelated taker's key as a hot authority and remove that taker's fills from its own score.

Replacement and generation

Every change to the list writes Quoter and Quote in the same instruction: the new list replaces Quoter.authorities and the cache in Quote, Quote.generation increases by one, and sequence, slot and mid are set to zero. The book has no fillable quote until a key from the new list lands an update carrying the new generation and a sequence of at least 1.

Operation	Data	Extra signers	Effect on Quote
Add a key	Current keys plus the new key	The new key	Generation + 1; sequence, slot and mid cleared
Remove a key	Current keys minus the removed key	None	Generation + 1; sequence, slot and mid cleared
Replace a key	The new key in place of the old	The new key	Generation + 1; sequence, slot and mid cleared
Repeat the current list	Identical list, no reset byte	None	No change; generation, sequence and live quote kept
Full reset	A list disjoint from the current one, reset byte 1	Every new key	Generation + 1, even for an identical empty list; sequence, slot and mid cleared

Because every update packet names a generation, a packet signed before a change cannot execute after it, even if its key is later re-added and its blockhash is still valid. The sequence alone cannot provide this, since a change resets it to zero.

Sequence discipline in the SDK

The protocol accepts up to three hot authorities for engines that coordinate sequences among themselves. The reference Rust SDK requires exactly one configured hot authority and takes an exclusive lease on the Quote's sequence before it reads the reference price or inventory. It starts above both the on-chain sequence and its own persisted high-water mark for the current generation, and it durably writes {generation, last_issued} before it returns signed bytes, so a crash cannot cause it to sign two different ladders with one sequence number. After local state is lost, the cold owner opens a new generation with set_quote_authority before the engine resumes.

4. The reference price

Each pair has one reference mid, stored in its RefPrice account, and every risk decision in OpenProp is taken against it. The band check compares a quoter's mid with it. The skew and loss checks mark book inventory at it. checkpoint_book and settle_epoch value books and the vault at it. The pair breaker, which halts every book in the pair at once, keeps its state in the same account.

A reference is written by post_ref_price, which anyone can call. It accepts a value only as two signed updates, one from Pyth Pro and one from Switchboard Surge, verified by the runtime in the same transaction and bound to one recent slot, and it stores their median next to each source's own price. A failed confidence, divergence or move test does not reject the post. The value is stored and the pair breaker trips.

4.1 The RefPrice account

RefPrice is a program-derived account with seeds ["ref", pair], account kind 2 and version byte 5. It is 3,776 bytes long: a 176-byte body followed by a 3,600-byte price history used by the move rule. initialize_pair creates it and records the pair address. Prices are written only by post_ref_price. The breaker fields are also written by halt_pair, resume_pair and, when the reference is stale, checkpoint_book.

All prices are integers in micro-USDC per SOL (six decimals), so 200.00 USDC per SOL is stored as 200000000.

Offset	Field	Type	Content
3	flags	u8	bit 0: pair breaker active; bit 1: guardian halt
4	reason	u8	reason code of the current halt
8	pair	pubkey	the pair account (PairConfig)
40	mid	u64	reference mid: the two-source median
48	confidence	u64	combined confidence
56	sourceSlot	u64	slot <i>s</i> that both signed updates are bound to
64	postedSlot	u64	clock.slot when the post executed
72	pythPrice	u64	Pyth Pro price
80	pythConfidence	u64	Pyth Pro confidence
88	pythSlot	u64	equal to sourceSlot
96	pythTimestampUs	u64	Pyth Pro signed timestamp, microseconds
104	switchboardPrice	u64	Switchboard Surge price
112	switchboardConfidence	u64	zero: Surge publishes no confidence
120	switchboardSlot	u64	equal to sourceSlot

Offset	Field	Type	Content
128	switchboardTimestampUs	u64	zero: Surge publishes no timestamp
136	trippedSlot	u64	slot at which the current halt began
144	clearedSlot	u64	slot at which the halting condition was first seen clear; zero while it persists
152	resumeSlot	u64	first slot at which a clean post lifts the halt
160	lastPythTimestampUs	u64	replay cursor for Pyth Pro
168	lastSwitchboardSlot	u64	replay cursor for the source slot
176	history	150 × 24 bytes	ring of (slot, low, high) per source slot

The reason byte uses the incident reason codes shared with book halts:

reason	Condition	Written by
1	guardian halt	halt_pair
2	stale reference	post_ref_price (gap detected), checkpoint_book
3	confidence too wide	post_ref_price
4	price move	post_ref_price
5	source divergence	post_ref_price

4.2 The post_ref_price instruction

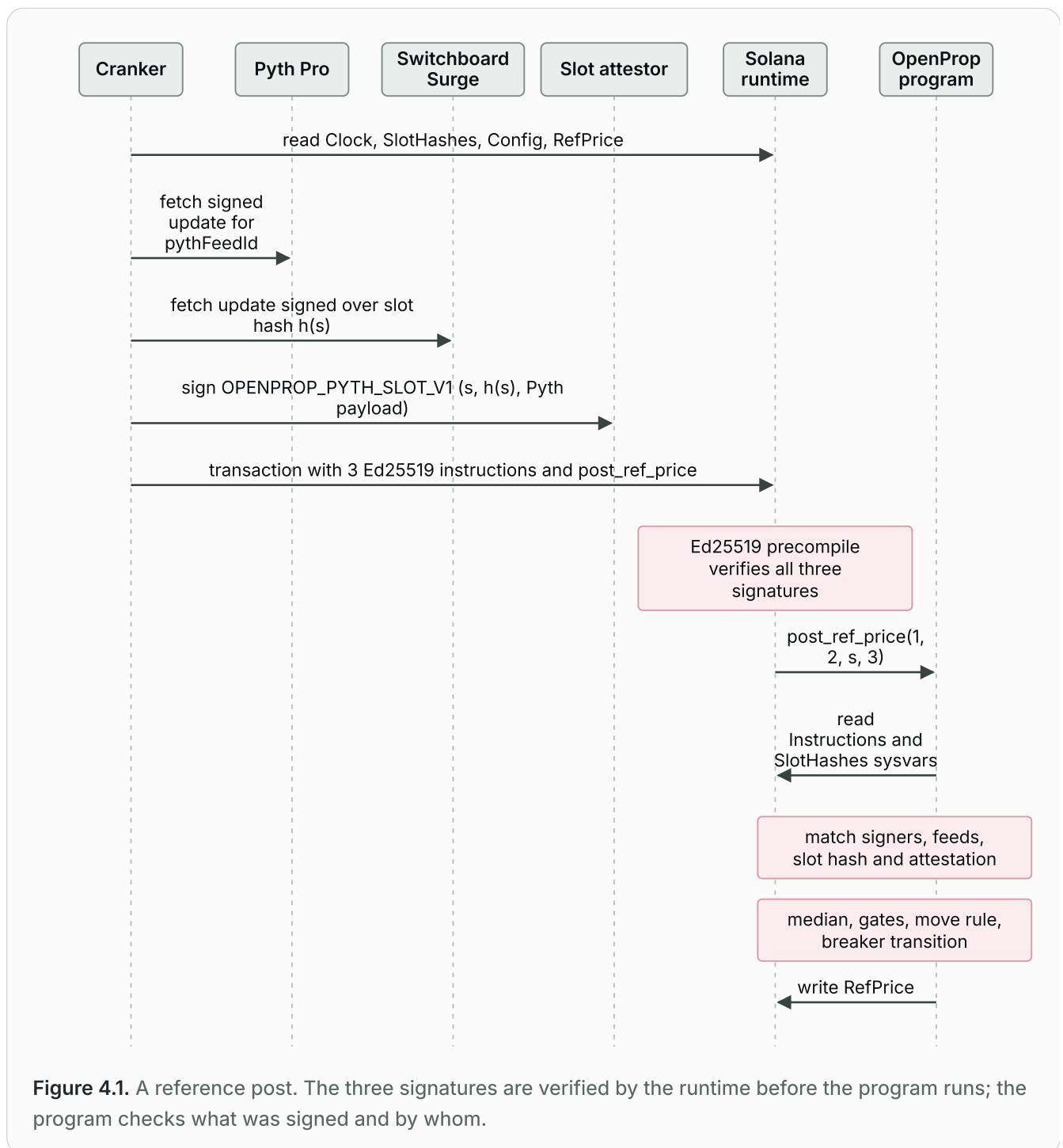
post_ref_price has wire tag 2. Its 14-byte body names where the three signed messages sit in the transaction and which slot they are bound to:

Bytes	Field
0–1	index of the Pyth Pro Ed25519 instruction (u16)
2–3	index of the Switchboard Surge Ed25519 instruction (u16)
4–11	source slot <i>s</i> (u64)
12–13	index of the slot-attestation Ed25519 instruction (u16)

It takes four accounts in this order: Config, RefPrice (writable), the Instructions sysvar and the SlotHashes sysvar. No signature is required beyond the fee payer's. The instruction makes no cross-program call. It reads the already verified messages from the Instructions sysvar.

The protocol cranker builds each reference transaction as a compute-budget instruction, three Ed25519 verification instructions (Pyth Pro, Switchboard Surge, slot attestation), and post_ref_price(1, 2, *s*, 3). It takes *s* from the newest entry of SlotHashes. It plans a reference post in every slot where Clock.slot is past

postedSlot. When a book is due for a checkpoint, checkpoint_book is appended to the same transaction, so the checkpoint marks the book at the price posted a moment earlier. The design estimates the instruction at 10–20k CU.



4.3 Verifying the signed updates

Ed25519 instructions

The Ed25519 precompile verifies every signature in the transaction before any program executes, and an invalid signature fails the whole transaction. post_ref_price then checks, for each of the three indices, that:

- [illegible]

Any mismatch fails with `INVALID_ORACLE` (6027).

The Pyth Pro update

The Pyth Pro message is the provider's own signed binary payload, read byte for byte:

Bytes	Field	Rule
0–3	magic, u32	must be 2479346549
4–11	timestamp, u64 microseconds	must advance (replay order, below)
12	channel	
13	feed count	must be 1
14–17	feed id, u32	must equal <code>Config.pythFeedId</code>
18	property count	
19–	properties	id 0: price i64; id 4: exponent i16; id 5: confidence i64

The three properties may appear in any order, each exactly once. An unknown property, a duplicate or trailing bytes fail with `INVALID_ORACLE`. The price must be positive, the confidence non-negative and the exponent between `-18` and `6`. Both values are converted to micro-units:

$$\text{scale}(v, e) = \begin{cases} v \cdot 10^{e+6} & \text{if } e+6 \geq 0 \\ \lfloor v / 10^{-(e+6)} \rfloor & \text{if } e+6 < 0 \end{cases} \quad p_P = \text{scale}(v_P, e), \quad c_P = \text{scale}(v_c, e)$$

where:

- v_P is the signed Pyth Pro price mantissa and v_c its confidence mantissa;
- e is the signed exponent;
- p_P and c_P are the Pyth Pro price and confidence in micro-USDC per SOL.

A result that does not fit in a u64 fails with `ARITHMETIC_OVERFLOW` (6025). For example, a mantissa of 19,999,000,000 at exponent -8 gives $p_P = \lfloor 19,999,000,000/100 \rfloor = 199,990,000$, which is 199.99 USDC.

The Switchboard Surge update

The Surge message is the 81-byte record defined by Switchboard's OracleQuote format:

Bytes	Field	Rule
0–31	signed slot hash	must match SlotHashes at slot s
32–63	feed hash	must equal Config.switchboardFeed
64–79	value, i128, 18 decimals	must be positive
80	minimum sample count	must be above zero

The price is $p_s = \lfloor v_s / 10^{12} \rfloor$, where v_s is the signed 18-decimal value. Surge signs a slot hash, not a slot number. The program therefore looks up s from the instruction body in the SlotHashes sysvar, which holds up to 512 recent (slot, hash) pairs, and requires an entry whose slot is s and whose hash equals the signed hash. A Surge update can only carry a hash the cluster had already produced, so it was signed at slot s or later. The unsigned slot footer that the Switchboard SDK appends to its instruction is never read.

The Pyth slot attestation

Pyth Pro signs a timestamp, not a Solana slot. To bind the Pyth update to the same slot s , a third key, Config.pythSlotAttestor, signs a message that wraps the complete Pyth payload:

Bytes	Field
0–20	ASCII OPENPROP_PYTH_SLOT_V1
21–52	OpenProp program id
53–84	Config address
85–92	source slot s , u64
93–124	slot hash of s , equal to the Surge signed hash
125–126	Pyth payload length n , u16
127–126+ n	the Pyth Pro payload, byte for byte

The message must be exactly $127 + n$ bytes long. The program id and Config address bind the attestation to one deployment. Config.oracleSlotAttestationPolicy must be 1, which configuration validation also enforces; any other value fails the post with UNDECIDED_PARAMETER (6003).

Freshness and replay order

Once both updates are bound to s , the program applies two ordering rules:

1. **Source age.** $t - s \leq N_r$, where t is Clock.slot and N_r is Config.refStalenessSlots. Otherwise the post fails with REF_STALE (6005).
2. **Replay order.** If the Pyth timestamp and s both equal the stored cursors lastPythTimestampUs and lastSwitchboardSlot, the post returns success and changes nothing. Otherwise the timestamp must be strictly greater than lastPythTimestampUs and s must be at least lastSwitchboardSlot, or the post fails with INVALID_ORACLE.

The Pyth timestamp orders updates and nothing else. Every expiry, staleness and cooldown rule counts `Clock.slot`, never `Clock.unix_timestamp`.

4.4 The two-source median

With two sources the median is the midpoint. The program stores

$$m = \left\lfloor \frac{p_P + p_S}{2} \right\rfloor, \quad c = \max\left(c_P, \left\lfloor \frac{|p_P - p_S|}{2} \right\rfloor\right)$$

where:

- m is the reference mid written to `RefPrice.mid`;
- c is the combined confidence written to `RefPrice.confidence`;
- p_P, c_P are the scaled Pyth Pro price and confidence, and p_S the scaled Surge price.

Surge publishes no confidence interval, so the second term stands in for it: half the disagreement between the sources. c is never narrower than either Pyth's own interval or the spread between the two feeds. The same post writes p_P, c_P and p_S to their own fields, and sets `pythSlot`, `switchboardSlot` and `sourceSlot` to s and `postedSlot` to t .

Using the contract's sample sources, $p_P = 199,990,000$ and $p_S = 200,010,000$ with $c_P = 10,000$ (0.01 USDC):

Quantity	Value	Relative to m
m	200,000,000 (200.00 USDC)	
$ p_P - p_S $	20,000	1.0 bps
$c = \max(10,000, 10,000)$	10,000	0.5 bps

4.5 Confidence and divergence gates

Two gates test the quality of the stored reference:

$$\text{confidence: } c \cdot 10^4 \leq m \cdot \kappa \quad \text{divergence: } |p_P - p_S| \cdot 10^4 \leq m \cdot \delta$$

where:

- κ is `Config.confidenceBps` and δ is `Config.divergenceBps`, both in basis points;
- c, m, p_P, p_S are as in §4.4.

Both comparisons use integer arithmetic with no division, and equality passes. Because c includes half the source disagreement, the confidence gate also fails whenever the sources differ by more than 2κ basis points of m . Both limits are set by the deployment's parameter file. Each must be at least its immutable floor, `minimumConfidenceBps` and `minimumDivergenceBps`, fixed at initialization, and neither may exceed 10,000. The sample reference above passes for every configuration that validation accepts: it sits at 0.5 bps of confidence and 1 bps of divergence, and both gates are at least 1 bps.

The gates are applied in three places with different consequences:

Where	Failing confidence	Failing divergence
post_ref_price	post stored, breaker trips with reason 3	post stored, breaker trips with reason 5
swap_exact_in (check 3, chapter 5)	CONFIDENCE_TOO_WIDE (6007), no fill	REFERENCE_DIVERGENCE (6035), no fill
checkpoint_book, settle_epoch valuation	CONFIDENCE_TOO_WIDE	REFERENCE_DIVERGENCE

Valuation also requires a positive, fresh mid and two positive source prices. It does not read the breaker flag, so a book can be marked and an epoch settled through a halt caused by a price move, but never at an uncertain or contested price.

4.6 Reference staleness

Staleness is measured in slots against `Config.refStalenessSlots` (N_r). The v0 set uses 10 slots, about 2 seconds at 200 ms slots. It is checked at four points:

Point	Test	On failure
post_ref_price, new update	$t - s \leq N_r$	REF_STALE; the post is rejected
post_ref_price, previous reference	$t - \text{sourceSlot}_{\text{old}} \leq N_r$	gap detected: breaker trips with reason 2 before the new value is written
swap_exact_in	$t - t_x \leq N_r$ for t_x each of <code>sourceSlot</code> , <code>pythSlot</code> , <code>switchboardSlot</code>	REF_STALE, no fill
checkpoint_book	$t - \text{sourceSlot} \leq N_r$	breaker trips with reason 2 and any cooldown in progress restarts; no checkpoint is taken

deposit also requires a fresh reference and a clear breaker, and fails with REF_STALE otherwise.

The gap check runs before the new value overwrites `sourceSlot`. A pair whose reference went stale therefore cannot skip the cooldown by posting a fresh price: the post that repairs the gap trips the breaker first, and the resume delay runs from there. `refStalenessSlots` must be at least 1 and at least the immutable floor `minimumRefStalenessSlots`.

4.7 The pair breaker

The pair breaker halts every book in the pair at once. Swaps read bit 0 of `RefPrice.flags` as check 8 and fail with PAIR_HALTED (6008) while it is set (chapter 5). The breaker trips on four conditions detected by the program, stale reference, wide confidence, source divergence and price move, and on a guardian halt.

The price history

history is a ring of 150 buckets of 24 bytes each: a source slot, the lowest mid posted for that slot and the highest. A post writes into bucket $s \bmod 150$. If the bucket already holds slot s , the post widens its low and high. Otherwise it overwrites the bucket with (s, m, m) . Several posts within one source slot are kept as their low and high, and the ring covers the most recent 150 source slots.

The move rule

Before writing its own bucket, the post compares the new mid with every bucket inside the window:

$$\text{trip if } \exists j, \exists p \in \{\ell_j, h_j\}, p > 0 : \quad 0 \leq s - s_j < W \text{ and } |m - p| \cdot 10^4 > p \cdot \mu$$

where:

- s is the new source slot and m the new mid;
- s_j, ℓ_j, h_j are the slot, low and high of bucket j ;
- W is `Config.breakerWindowSlots`, between 1 and 150;
- μ is `Config.breakerMoveBps`, at least the immutable floor `minimumBreakerMoveBps`.

The window is strict: with $W = 150$ it holds slot s and the 149 slots before it. The move is measured relative to the older price p , and a move of exactly μ does not trip. The v0 set uses $\mu = 300$ bps and $W = 150$ slots: a move above 3% within 150 slots, about 30 seconds. Against a stored 200.00 the threshold is $200 \times 300/10^4 = 6.00$ USDC, so 206.00 passes and 206.01 trips. Against a bucket with low 200.00 and high 201.00, a new mid of 194.50 trips on the high: $|194.50 - 201.00| = 6.50 > 6.03$.

When several conditions fail in one post, the stored reason is the last one evaluated: price move (4) over divergence (5) over confidence (3).

Trip, clear and resume

After the gates and the move rule, each post applies one transition:

1. **A condition fails.** If bit 0 is clear, the program sets `trippedSlot` to t and emits an incident event (kind 3: reason, action 0, mid). It sets bit 0, writes the reason, and zeroes `clearedSlot` and `resumeSlot`. A condition that persists keeps zeroing them, so the cooldown cannot start while the condition holds.
2. **The post is clean and the breaker is active without a guardian halt.** On the first clean post, `clearedSlot` becomes t and `resumeSlot` becomes $t + D$, where D is `Config.resumeDelaySlots`. On any clean post with $t \geq \text{resumeSlot}$, the program clears bit 0, sets the reason to 0 and emits an incident event with action 1.

The v0 set uses $D = 50$ slots, about 10 seconds. D is bounded above by the immutable `maximumResumeDelaySlots`. Lifting a halt always takes a post: the program lifts it only while processing a clean reference at or after `resumeSlot`.

For a sustained step from 200.00 to about 206.10 between source slots s_0 and $s_0 + 1$, with the v0 values:

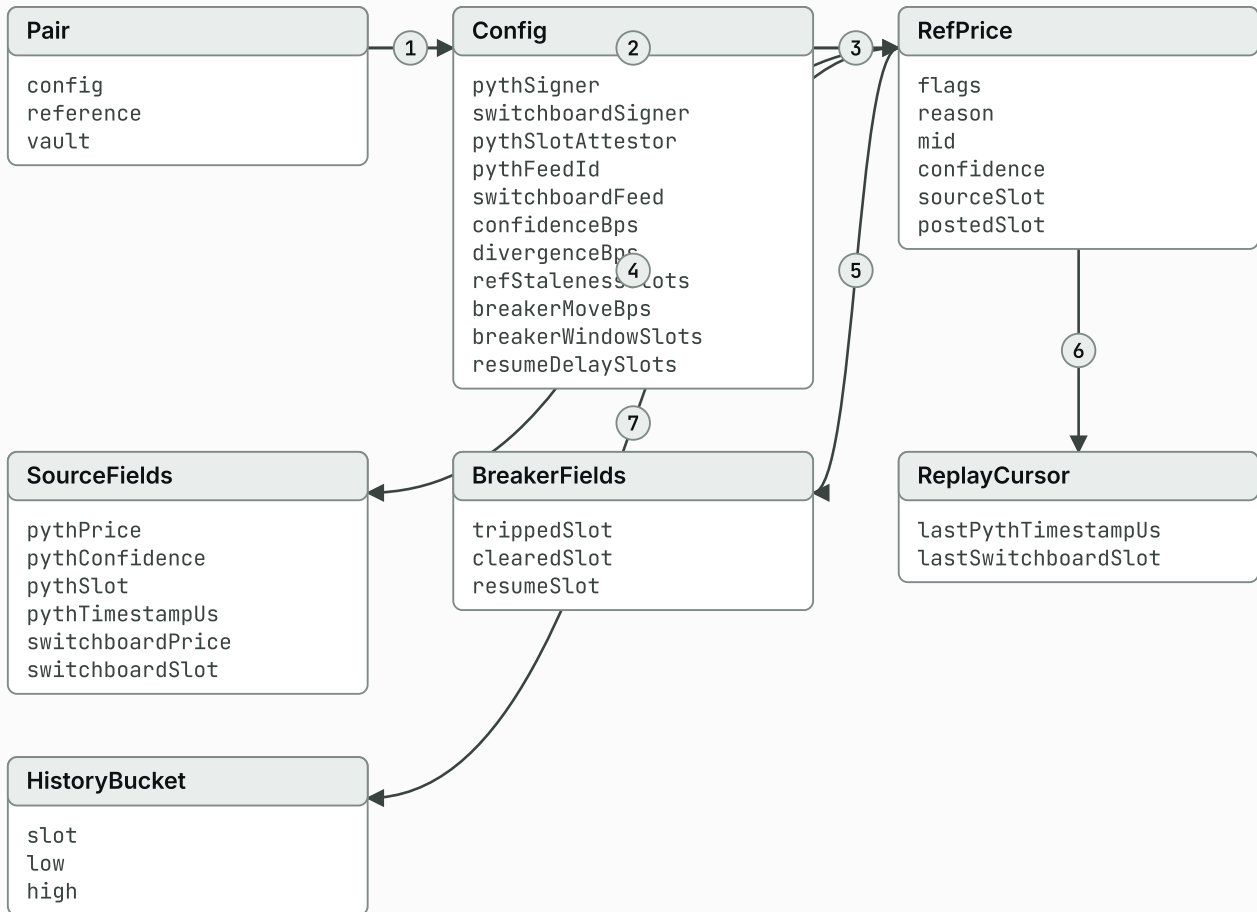
Source slot	Mid	Move test	State after the post
s_0	200.00	clean	clear
$s_0 + 1$	206.10	$6.10 > 6.00$ against bucket s_0	tripped, reason 4
$s_0 + 2 \dots s_0 + 149$	$\approx$ 206.10	bucket s_0 still inside the window	tripped; <code>clearedSlot</code> stays 0
$s_0 + 150$	$\approx$ 206.10	bucket s_0 aged out; clean	tripped; <code>clearedSlot</code> = t , <code>resumeSlot</code> = $t + 50$

Source slot	Mid	Move test	State after the post
posts with <code>Clock.slot ≥ resumeSlot</code>	$\approx$ 206.10	clean	clear; incident with action 1

A sustained step therefore halts the pair for about 200 slots, roughly 40 seconds. Throughout, `checkpoint_book` and `settle_epoch` can still mark at the new price, because valuation does not read the breaker flag.

Guardian halt and admin resume

`halt_pair` (tag 13) is signed by `Config.guardian` or by the optional halt-only delegate `Config.guardianHaltSigner`. It sets both flag bits (value 3), reason 1 and `trippedSlot`, and emits an incident event. Repeating it while a guardian halt is in place changes nothing. While bit 1 is set, no post can lift the halt. Only `resume_pair` (tag 20), signed by `Config.admin`, clears bit 1, and it requires a fresh reference, failing otherwise with `REF_STALE`. It leaves bit 0 set, sets `clearedSlot` to t and `resumeSlot` to $t + D$, and hands the pair back to the ordinary cooldown above. Neither instruction moves funds.



- ① Pair → Config: config
- ② Pair → RefPrice: reference, seeds ["ref", pair]
- ③ Config → RefPrice: signer keys, feeds, gates
- ④ RefPrice → SourceFields: per-source values
- ⑤ RefPrice → BreakerFields: breaker transition
- ⑥ RefPrice → ReplayCursor: replay order
- ⑦ RefPrice → HistoryBucket: history[150]

Figure 4.2. Reference and breaker state. SourceFields, BreakerFields, ReplayCursor and HistoryBucket are field groups inside the single RefPrice account; Config holds the identities and limits it is checked against.

4.8 Parameters

Config field	Role	Value
refStalenessSlots	maximum reference age	10 slots (v0)
breakerMoveBps	move threshold μ	300 bps (v0)
breakerWindowSlots	move window W , 1 to 150	150 slots (v0)
resumeDelaySlots	cooldown D after the condition clears	50 slots (v0)

Config field	Role	Value
confidenceBps	confidence gate κ	set by the deployment's parameter file
divergenceBps	divergence gate δ	set by the deployment's parameter file
oracleSlotAttestationPolicy	Pyth slot attestation required	must be 1
pythSigner, switchboardSigner, pythSlotAttestor	Ed25519 keys whose signatures are accepted	set at initialization
pythFeedId, switchboardFeed	the SOL/USDC feed on each source	set at initialization
minimumConfidenceBps, minimumDivergenceBps, minimumRefStalenessSlots, minimumBreakerMoveBps, maximumResumeDelaySlots	immutable envelope bounding later changes	fixed at initialization

Under `set_params`, raising `confidenceBps`, `divergenceBps`, `refStalenessSlots` or `breakerMoveBps` counts as loosening, as does lowering `resumeDelaySlots` or `breakerWindowSlots`. A loosening change waits `timeLockEpochs × epochSlots` slots, and at least one epoch longer than the withdrawal queue. A tightening change applies immediately. The signer keys and feed identities change only through `rotate_identities`, which always waits the full timelock.

5. The five checks

OpenProp names five conditions a fill must meet: the quote is fresh, its mid is inside the band around the reference, inventory skew stays inside its limit after the fill, the book's loss windows are not breached, and the pair breaker is not tripped. The program enforces them as nine ordered checks. The five are joined by a book-state check, a validity check on the reference itself and a check of the taker's signed minimum output, and the loss check is split into an hourly and an epoch window.

All nine run in the shared rules crate `openprop-rules` (rules version 0.6.0). The same source is compiled into the on-chain program, linked into the quoter SDK and, through it, the Jupiter adapter, and compiled to the WASM module that the API and the web app evaluate. `swap_exact_in` calls `evaluate_runtime_swap_with_state` with the current `Clock.slot`, the `Quote`, the `RefPrice`, the book's tracked inventory and the risk parameters read from `Config`. The engine computes the whole fill in memory and returns a result. The program writes book state and moves tokens only when that result is a fill or a partial fill. A rejection returns the error of the first failing check, and the transaction leaves every account unchanged.

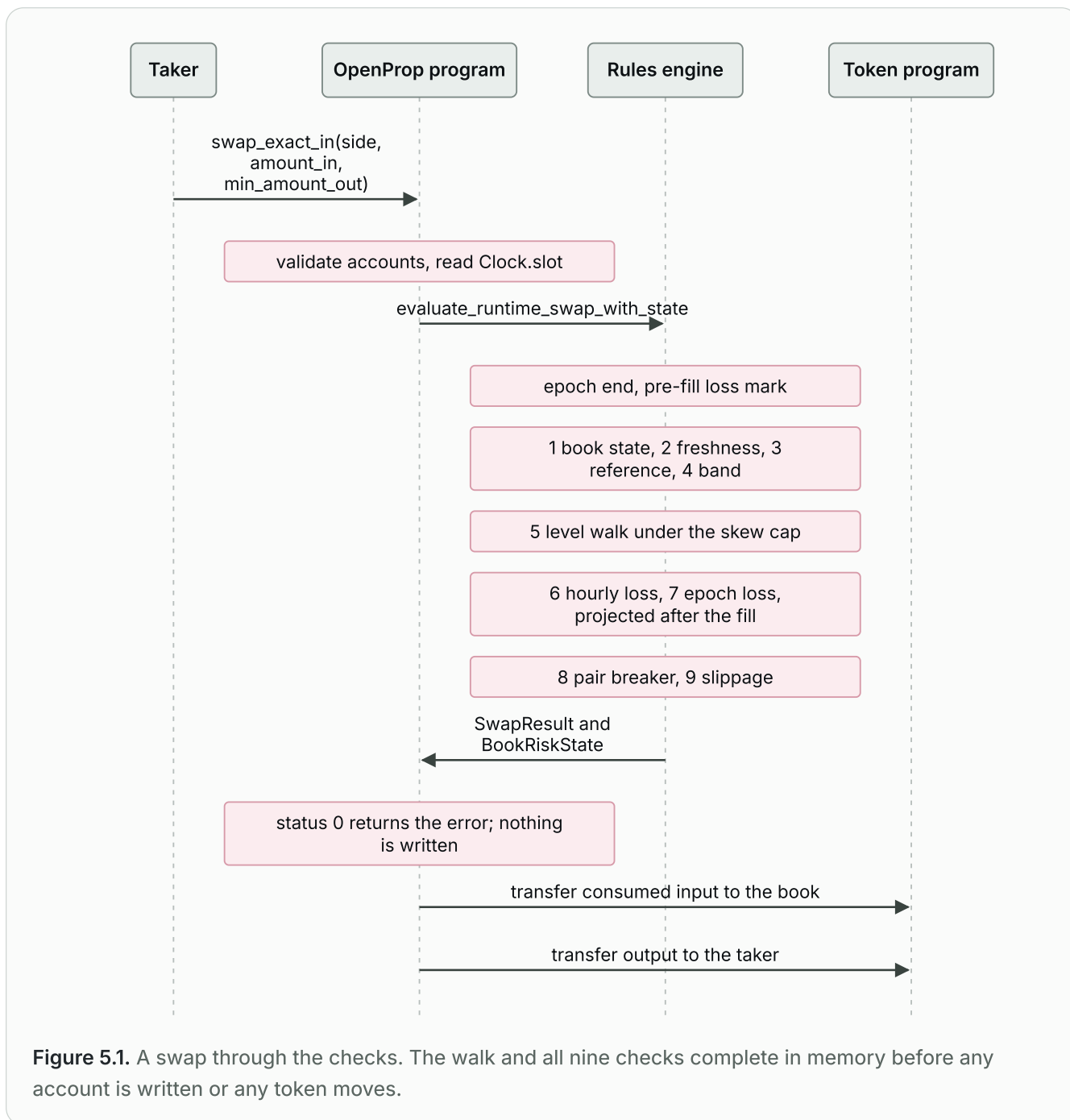
5.1 Order of evaluation

#	Contract key	Test	Config parameters	Error on failure	Outcome
0	(runtime)	slot before the book's epoch end		EPOCH_NOT_READY (6026)	no fill
1	book-state	book live; input nonzero		BOOK_PAUSED (6010), BOOK_HALTED (6009), INVALID_AMOUNT (6033)	no fill
2	freshness	ladder valid; quote age within limit	quoteStalenessSlots, retailDeltaLimitBps, maxLevelBaseAtoms	QUOTE_LADDER_INVALID (6034), QUOTE_STALE (6004)	no fill
3	reference	reference positive, fresh, confident, sources agree	refStalenessSlots, confidenceBps, divergenceBps	INVALID_ORACLE (6027), REF_STALE (6005), CONFIDENCE_TOO_WIDE (6007), REFERENCE_DIVERGENCE (6035)	no fill
4	band	quote mid within the band	bandBps	MID_OUTSIDE_BAND (6006)	no fill
5	skew	post-fill skew within the cap, applied level by level	skewBps	SKEW_LIMIT (6011), INSUFFICIENT_LIQUIDITY (6016)	partial fill at the cap; no fill if nothing fits
6	hourly-loss	projected hourly loss within limit	hourlyLossBps, hourlyWindowSlots	HOURLY_LOSS_LIMIT (6012)	no fill
7	epoch-loss	projected epoch loss within limit	epochLossBps	EPOCH_LOSS_LIMIT (6013)	no fill
8	pair-breaker	RefPrice flag bit 0 clear		PAIR_HALTED (6008)	no fill

Contract		Test	Config parameters	Error on failure	Outcome
#	key				
9	slippage	total output at least the signed minimum		SLIPPAGE_EXCEEDED (6015)	no fill

A configuration with an unset or out-of-range limit (any basis-point limit above 10,000, or `maxLevelBaseAtoms` of zero) fails at position 1 with `UNDECIDED_PARAMETER` (6003). Configuration validation prevents that state from being written.

The engine records one status per check in `check_status[9]`: 0 not reached, 1 passed, 2 failed, 3 partial (skew only). The overall status is 0 for a rejection, 1 for a complete fill and 2 for a partial fill. A partial result carries the reason the walk stopped, `SKEW_LIMIT` or `INSUFFICIENT_LIQUIDITY`, in its error field, but the swap succeeds. The contract exposes the nine keys and statuses to clients so that a quote shown in an interface explains a rejection the same way the program would.



5.2 Before the checks: epoch end and the pre-fill mark

Two runtime steps come first. A book trades only inside its current epoch: if $t \geq \text{Book.epochEndSlot}$, where t is `Clock.slot`, the swap fails with `EPOCH_NOT_READY` until `settle_epoch` rolls the book into the next epoch. A book that has never been allocated has an `epochEndSlot` of zero and fails the same way.

The engine then marks the book's loss state from its tracked inventory before the fill (§5.8). This step is where an expired hourly window resets. Its results replace the stored `hourlyLoss` and `epochLoss` counters as inputs to checks 6 and 7, so the stored counters are records of the last mark and are never trusted as inputs.

5.3 Check 1: book state

The book's state byte (`Book.flags`) must be 0, live. State 2 is a pause set by the quoter's cold key through `pause_book` and fails with `B00K_PAUSED`. State 1 is a risk halt set by `checkpoint_book` on a loss-limit breach and fails with `B00K_HALTED`. A risk halt is cleared only by the epoch rebalance in `settle_epoch`, and a cold-key pause cannot overwrite it. The same position rejects an `amount_in` of zero with `INVALID_AMOUNT`.

5.4 Check 2: quote freshness

The quote is first checked for a well-formed ladder. `update_quote` stores the ladder as written and leaves validation to every swap. The quote fails with `QUOTE_LADDER_INVALID` if its mid is zero, if its `retailDeltaBps` exceeds `Config.retailDeltaLimitBps`, or if on either side a level has an offset of 10,000 bps or more, a size above `Config.maxLevelBaseAtoms`, or an offset smaller than the level before it. Offsets therefore never decrease from level 1 to level 5, and level 1 is the best price.

Freshness is then

$$0 \leq t - t_q \leq N_q$$

where:

- t is `Clock.slot` at execution;
- t_q is `Quote.slot`, the slot `update_quote` last wrote;
- N_q is `Config.quoteStalenessSlots`.

A failure returns `QUOTE_STALE`. The v0 set uses $N_q = 2$: a quote written at slot t_q fills through slot $t_q + 2$ and fails from slot $t_q + 3$, about 600 ms later at 200 ms slots. Staleness is counted in slots, never in `unix_timestamp`.

5.5 Check 3: reference validity

The engine re-validates the stored reference on every swap:

1. `RefPrice.mid`, `pythPrice` and `switchboardPrice` are all positive, or `INVALID_ORACLE`.
2. $t - t_x \leq N_r$ for each of `sourceSlot`, `pythSlot` and `switchboardSlot`, where N_r is `Config.refStalenessSlots`, or `REF_STALE`.
3. $c \cdot 10^4 \leq p \cdot \kappa$, or `CONFIDENCE_T00_WIDE`.
4. $|p_P - p_S| \cdot 10^4 \leq p \cdot \delta$, or `REFERENCE_DIVERGENCE`.

Here p is the reference mid, c its combined confidence, p_P and p_S the two source prices, $\kappa = \text{confidenceBps}$ and $\delta = \text{divergenceBps}$. Chapter 4 derives each quantity. The v0 set uses $N_r = 10$ slots. The confidence and divergence limits are set by the deployment's parameter file.

5.6 Check 4: band

$$|m_q - p| \cdot 10^4 \leq p \cdot \beta$$

where m_q is `Quote.mid`, p the reference mid and $\beta = \text{Config.bandBps}$. A failure returns `MID_OUTSIDE_BAND`. The band bounds how far a quoter's ladder can be centred away from the reference. A quote mid of 202.00 against a reference of 200.00 is 100 bps away and is rejected for any bandBps below 100. The band width is set by the deployment's parameter file.

5.7 Check 5: inventory skew

Skew compares the reference value of the book's SOL with its USDC:

$$V(B) = \left\lfloor \frac{B \cdot p}{10^9} \right\rfloor, \quad \sigma(B, Q) = \frac{|V(B) - Q|}{V(B) + Q}$$

where:

- B is the book's tracked SOL in lamports (`Book.trackedBase`);
- Q is its tracked USDC in micro-USDC (`Book.trackedQuote`);
- p is the reference mid in micro-USDC per SOL;
- $V(B)$ is the SOL marked at the reference, in micro-USDC.

A balanced book has $\sigma = 0$, and a book holding only one asset has $\sigma = 1$. The admissible set for the post-fill inventory (B', Q') depends on where the book starts:

inside the cap: $|V(B') - Q'| \cdot 10^4 \leq (V(B') + Q') \cdot \gamma$ beyond the cap: $|V(B') - Q'| \cdot T_0 \leq (V(B') + Q') \cdot D_0$

where $\gamma = \text{Config.skewBps}$, and $D_0 = |V(B_0) - Q_0|$ and $T_0 = V(B_0) + Q_0$ are taken from the inventory before the swap. A book already beyond the cap, which a reference move can cause, may still trade in the direction that reduces its skew, but no fill may leave it more skewed than it started.

Skew is tested inside the level walk (chapter 6) after each level's proposed fill. When a full level would leave the admissible set, the engine binary-searches for the largest quantity in $[0, q_{\text{level}}]$ that stays inside it.

Absolute skew along one trade direction is V-shaped in quantity, and the starting inventory is admissible in both modes, so the admissible quantities form an interval starting at zero and the search is exact. The walk fills that quantity and stops with reason `SKEW_LIMIT`:

- if earlier levels or a nonzero part of this level filled, the swap is a **partial fill** (status 2, check status 3).
The program transfers only the consumed input, and the unconsumed input never leaves the taker;
- if nothing at all fits, the swap is rejected with `SKEW_LIMIT`.

The limit binds at equality: a fill that brings skew exactly to the cap is admitted. On the contract's sample book, 900 SOL and 180,000 USDC at a reference of 200.00, so $V(B) = Q$, each SOL of one-directional flow moves σ by about 11.1 bps. The 5 SOL sell worked through in chapter 6 leaves σ at 55.55 bps. The skew limit is set by the deployment's parameter file.

5.8 Checks 6 and 7: hourly and epoch loss

Excess over hold

Losses are measured as excess over hold: the book's value now minus the value of its epoch-start inventory, both marked at the current reference:

$$E = (V(B) + Q) - (V(B_0) + Q_0)$$

where B_0 , Q_0 are `Book.startBase` and `Book.startQuote`, set at the epoch rebalance. A move in the SOL price alone does not create excess, because the hold portfolio moves with it. Only trading at prices worse than the reference, or inventory built up before an adverse move, creates a loss.

The two windows

$$L_e = \max(-E, 0), \quad L_h = \max(H - E, 0)$$

where:

- L_e is the epoch loss: negative excess since the epoch began;
- H is the hourly baseline `Book.hourlyStartExcess`, the excess when the current hourly window opened;
- L_h is the hourly loss: the fall in excess since that baseline.

The hourly window is a fixed window, not a rolling one. At the pre-fill mark, if $t - \text{hourlyStartSlot} \geq W_h$, where W_h is `Config.hourlyWindowSlots`, the window resets: `hourlyStartSlot` becomes t and H becomes the current pre-fill excess E_{pre} . The post-fill mark keeps the baseline chosen by the pre-fill mark, so a fill's own loss always counts in the window it executes in. `checkpoint_book` applies the same reset. `settle_epoch` opens a fresh window at the rebalance with $H = 0$. W_h is set at initialization and belongs to the clock geometry that `set_params` cannot change in place.

Projection

The engine marks the book twice, at the pre-fill inventory and at the post-fill inventory produced by the walk, and tests the larger of the two:

$$\max(L_h^{\text{post}}, L_h^{\text{pre}}) \cdot 10^4 \leq A \cdot \lambda_h, \quad \max(L_e^{\text{post}}, L_e^{\text{pre}}) \cdot 10^4 \leq A \cdot \lambda_e$$

where:

- A is `Book.allocation`, the book's reference value at the epoch rebalance, in micro-USDC;
- λ_h is `Config.hourlyLossBps` and λ_e is `Config.epochLossBps`.

Failures return `HOURLY_LOSS_LIMIT` and `EPOCH_LOSS_LIMIT`. The post-fill term rejects a fill that would take the book past a limit. The pre-fill term rejects every fill on a book whose window is already breached, so a profitable fill cannot reopen it. Equality passes. The v0 set uses $\lambda_h = 50$ bps (0.5% of allocation) and $\lambda_e = 200$ bps (2%).

Worked example

On the sample book, $A = 360,000$ USDC, so the hourly limit is 1,800 USDC and the epoch limit 7,200 USDC. A fill loses against the reference when its level price is on the book's wrong side of p , which the band permits up to bandBps.

Situation	H	E_{pre}	E_{post}	L_h pre / post	L_e post	Result
Fill losing 60.00 against the reference	+40.00	-1,700.00	-1,760.00	1,740.00 / 1,800.00	1,760.00	passes: hourly loss equals the limit
Same fill, losing 60.0000001	+40.00	-1,700.00	-1,760.0000001	1,740.00 / 1,800.0000001	1,760.0000001	HOURLY_LOSS_LIMIT
Window expired at the pre-fill mark	reset to -1,700.00	-1,700.00	-1,760.00	0 / 60.00	1,760.00	passes
Profitable fill on a breached window	+40.00	-1,760.0000001	-1,750.00	1,800.0000001 / 1,790.00	1,750.00	HOURLY_LOSS_LIMIT

All amounts are in USDC. The contract's recorded vectors reject a book carrying 1,801 USDC of hourly loss or 7,201 USDC of epoch loss on this allocation.

What is persisted

On success the program writes the post-fill hourlyLoss, epochLoss and epochExcess, and the window's hourlyStartSlot and hourlyStartExcess, which may have just reset. On rejection nothing is written, including a pending window reset, which the next swap or checkpoint performs again. The swap never halts a book. checkpoint_book, which a cranker calls about every 1,500 slots, applies the same marks. If epoch loss exceeds its limit it halts the book with reason 7, otherwise if hourly loss exceeds its limit it halts it with reason 6. Either halt sets the book's state to 1 and records the trip in the Quoter's trip bitmaps, which the tier rules read. From then on, check 1 rejects every swap until the epoch rebalance.

5.9 Check 8: pair breaker

The swap fails with PAIR_HALTED while bit 0 of RefPrice.flags is set. Chapter 4 describes how the bit is set by a stale reference, wide confidence, source divergence, a price move above breakerMoveBps within breakerWindowSlots, or a guardian halt, and how it clears resumeDelaySlots after the condition clears. The breaker covers every book of the pair because all of them read the same RefPrice.

5.10 Check 9: slippage

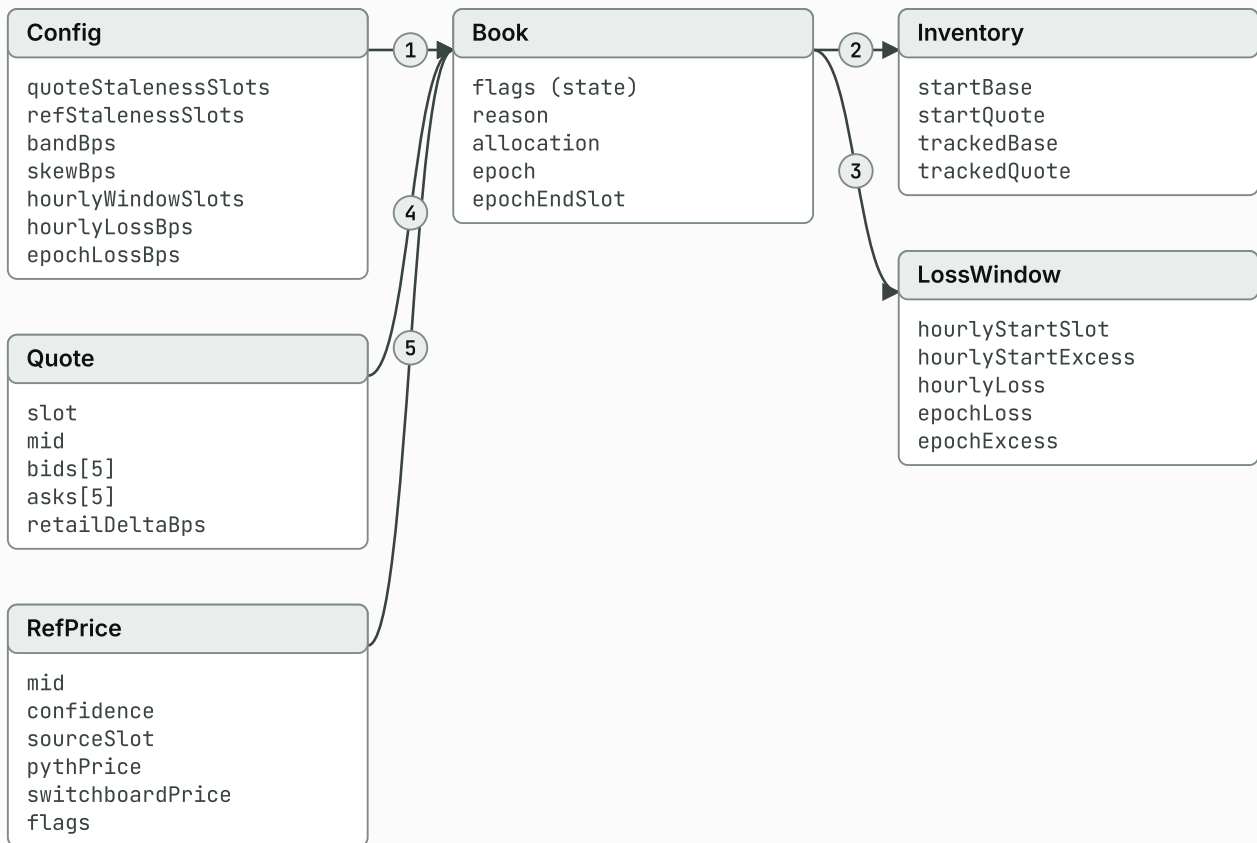
$$\text{out} \geq \text{min_amount_out}$$

where out is the total output of the walk, partial or complete, and min_amount_out is the minimum the taker signed into the instruction. A failure returns SLIPPAGE_EXCEEDED. This check is what bounds a partial fill for the taker: a route that needs the full amount sets a minimum the partial output cannot reach.

5.11 Equality at the limit

Every limit is tested as a cross-multiplied integer inequality, and every limit admits its boundary value:

Check	Passes when	Fails when
Quote freshness	$\text{age} = N_q$	$\text{age} > N_q$
Reference age	$\text{age} = N_r$	$\text{age} > N_r$
Confidence	$c \cdot 10^4 = p \cdot \kappa$	$c \cdot 10^4 > p \cdot \kappa$
Divergence	equal to the limit	above the limit
Band	mid exactly bandBps from the reference	further
Skew	post-fill skew exactly at skewBps	above: the fill is trimmed to the cap
Hourly, epoch loss	loss exactly at the limit	above
Slippage	output equals the minimum	output below the minimum



- ① Config → Book: limits for checks 2 to 7
- ② Book → Inventory: hold baseline and live inventory
- ③ Book → LossWindow: window and projection state
- ④ Quote → Book: checks 2 and 4, ladder for the walk
- ⑤ RefPrice → Book: checks 3, 4 and 8; marks for 5 to 7

Figure 5.2. Limit state on a book. Inventory and LossWindow are field groups of the Book account; Quote and RefPrice are read-only inputs to every swap.

6. Swap execution

`swap_exact_in` is the instruction aggregators route to. It fills an exact input amount against one book's current quote. With side 0 the taker sells SOL for USDC against the bids. With side 1 the taker buys SOL with USDC against the asks. The instruction runs in four stages: it validates the accounts, evaluates the nine checks and the level walk in the shared rules engine (chapter 5), writes the book, and makes two SPL Token transfers. It makes no other cross-program call, emits no logs and allocates nothing on the heap.

6.1 Instruction and accounts

The instruction has wire tag 1 and a 17-byte body:

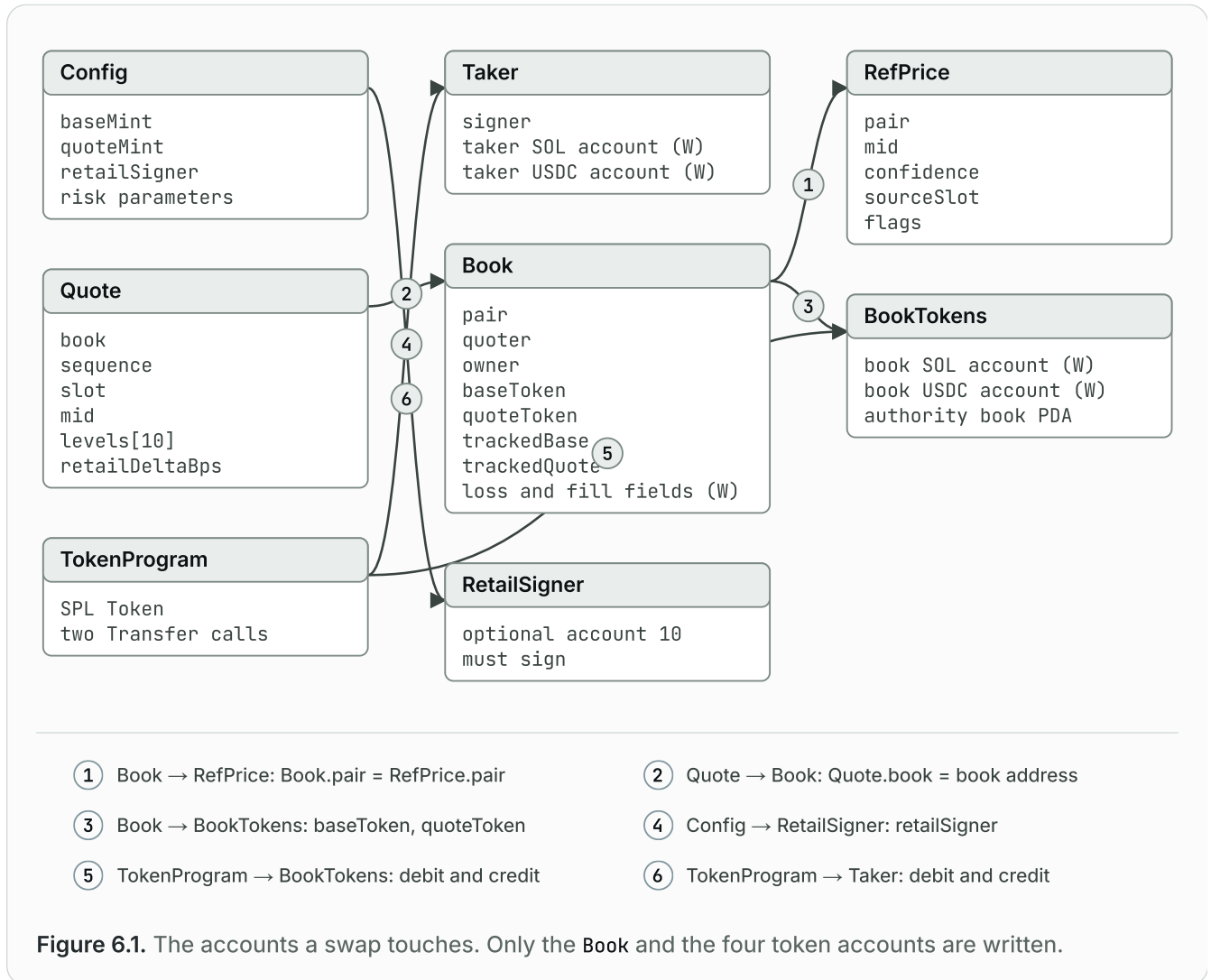
Bytes	Field	Meaning
0	side	0: sell SOL (SellBase); 1: buy SOL (BuyBase)
1-8	amount_in	exact input: lamports when selling, micro-USDC when buying
9-16	min_amount_out	smallest acceptable total output, signed by the taker

It takes ten accounts, or eleven with a retail signer:

#	Account	Access	Validation	Error
0	taker	signer	must sign	UNAUTHORIZED (6002)
1	Config	read	program-owned, kind 0, version 6	INVALID_ACCOUNT (6001)
2	RefPrice	read	kind 2; pair equals Book.pair	INVALID_ACCOUNT
3	Book	write	kind 5	INVALID_ACCOUNT
4	Quote	read	kind 6; Quote.book equals the book address	INVALID_ACCOUNT
5	book SOL account	write	equals Book.baseToken; mint Config.baseMint; authority the book	INVALID_ACCOUNT, INVALID_TOKEN_ACCOUNT (6029)
6	book USDC account	write	equals Book.quoteToken; mint Config.quoteMint; authority the book	as above
7	taker SOL account	write	mint Config.baseMint; authority the taker	INVALID_TOKEN_ACCOUNT
8	taker USDC account	write	mint Config.quoteMint; authority the taker	INVALID_TOKEN_ACCOUNT
9	SPL Token program	read	TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA	INVALID_TOKEN_ACCOUNT
10	retail signer (optional)	signer	equals Config.retailSigner	none: pricing falls back to standard

Every token account must be a 165-byte account owned by the classic SPL Token program and initialized. A wrong account count, body length or side value fails with `INVALID_INSTRUCTION (6000)`. The book's token balances must also cover its tracked inventory, trackedBase and trackedQuote (\$6.6), or the swap fails with `INVALID_ACCOUNT`.

A swap write-locks the Book and the four token accounts and only reads the Quote. update_quote write-locks only the Quote. The book PDA has seeds ["book", quoter, pair], and the Quote PDA has seeds ["quote", book].



6.2 The level walk

After checks 1 to 4 pass (chapter 5), the engine walks one side of the ladder: the bids for a sell, the asks for a buy. It takes levels in order from level 1 outward. Offsets never decrease from one level to the next, so level 1 is the best price. A level with zero size is skipped. For each level i :

$$o_i^* = \begin{cases} \max(o_i - \rho, 0) & \text{retail flow} \\ o_i & \text{otherwise} \end{cases} \quad P_i = \begin{cases} \lfloor m_q (10^4 - o_i^*) / 10^4 \rfloor & \text{sell (bids)} \\ \lfloor m_q (10^4 + o_i^*) / 10^4 \rfloor & \text{buy (asks)} \end{cases}$$

where:

- o_i is the level's offset in bps and ρ is `Quote.retailDeltaBps`;
- m_q is `Quote.mid` in micro-USDC per SOL;
- P_i is the level price in micro-USDC per SOL.

The quantity and output at the level then depend on the side:

$$\begin{aligned} \text{sell: } q_i &= \min\left(R, z_i, \left\lfloor \frac{(Q+1) \cdot 10^9 - 1}{P_i} \right\rfloor\right), & u_i &= \left\lfloor \frac{q_i P_i}{10^9} \right\rfloor \\ \text{buy: } q_i &= \min\left(\left\lfloor \frac{R \cdot 10^9}{P_i} \right\rfloor, z_i, B\right), & u_i &= \left\lceil \frac{q_i P_i}{10^9} \right\rceil \end{aligned}$$

where:

- R is the input not yet consumed: lamports for a sell, micro-USDC for a buy;
- z_i is the level size in lamports;
- B, Q are the book's SOL and USDC after the previous levels;
- q_i is the SOL that changes hands at level i , in lamports;
- u_i is the USDC that changes hands: paid to the taker on a sell, charged to the taker on a buy.

On a sell, the third term caps q_i so that u_i never exceeds the book's USDC. On a buy, $q_i \leq B$ caps it at the book's SOL. The proposed inventory is then tested against the skew cap, and the quantity is trimmed by binary search if it fails (§5.7). The engine applies the level, $B \pm q_i$ and $Q \mp u_i$, records a level fill (level, price, input, output, lamports) and reduces R by the input spent: q_i on a sell, u_i on a buy. The walk continues until $R = 0$ or a stop condition ends it:

Condition	Stop reason	Result
input consumed ($R = 0$)		complete fill (status 1)
sell: the book's USDC cannot pay for the full level	INSUFFICIENT_LIQUIDITY	partial fill, or rejection if nothing filled
buy: the book's SOL is exhausted	INSUFFICIENT_LIQUIDITY	partial fill, or rejection if nothing filled
skew cap reached	SKEW_LIMIT	partial fill, or rejection if nothing fits
a level's USDC amount rounds to zero after an earlier fill	INSUFFICIENT_LIQUIDITY	partial fill
all five levels used with $R > 0$	INSUFFICIENT_LIQUIDITY	partial fill

A sell too small to earn one micro-USDC at the first level, such as 1 lamport, is rejected with `INVALID_AMOUNT`. The per-level fills are not logged on chain. Clients recompute them with the same rules crate.

Rounding

Every rounding step is fixed in the code:

Quantity	Rounding	In whose favour
bid price P_i	down	the book

Quantity	Rounding	In whose favour
ask price P_i	down	the taker, by less than one micro-USDC per SOL
USDC paid to a seller, u_i	down	the book
SOL a buyer can afford, $\lfloor R \cdot 10^9 / P_i \rfloor$	down	the book
USDC charged to a buyer, u_i	up	the book
inventory mark $V(B)$ in skew and loss checks	down	neither; used for limits, not for payment

At integer bps offsets on a mid that is a multiple of 10,000 micro-USDC, such as 200.00, both price formulas are exact. The amount rounding then decides every residual atom, and the book keeps it.

6.3 Partial fills and the minimum output

A partial fill (status 2) is a successful swap. The program transfers only the consumed input, and the rest never leaves the taker's account. The output is still checked against `min_amount_out` (check 9), so a taker that needs the whole amount filled sets a minimum the partial output cannot reach. The Jupiter adapter evaluates the same rules on its account snapshot and returns no quote for an amount that would fill only partially. A routed swap is therefore complete at quote time, and `min_amount_out` governs if the book changes before execution.

Selling 40 SOL into the sample bids exhausts the ladder. All five levels fill, 31 SOL for 6,194.46 USDC at an average of 199.82, and the walk ends with $R = 9$ SOL. The result is a partial fill with reason `INSUFFICIENT_LIQUIDITY`, and the 9 SOL stay with the taker. The result also requires `skewBps` to admit the post-fill skew of 344.29 bps.

6.4 Worked example: the sample ladder

The examples use the contract's sample book: quote mid 200.00 equal to the reference, levels at 1, 2, 4, 8 and 12 bps of 1, 2, 4, 8 and 16 SOL on each side, a retail delta of 1 bps, and a book holding 900 SOL and 180,000 USDC at its epoch start. They assume a fresh quote and reference, a `maxLevelBaseAtoms` of at least 16 SOL so the ladder is valid, and a `skewBps` of at least 56. The figures were produced by the shared rules engine.

Selling 5 SOL

Level	Offset	Price (USDC)	Size (SOL)	Filled (SOL)	Output (USDC)	Remaining (SOL)
1	1 bps	199.98	1	1	199.980000	4
2	2 bps	199.96	2	2	399.920000	2
3	4 bps	199.92	4	2	399.840000	0
Total				5	999.740000	

The taker receives 999.74 USDC, an average of 199.948 USDC per SOL, 2.60 bps below the mid. The book ends at 905 SOL and 179,000.26 USDC. Marked at 200.00, its excess over hold rises by 0.26 USDC, the spread it captured, and its skew moves from 0 to 55.55 bps.

Buying with 1,000 USDC

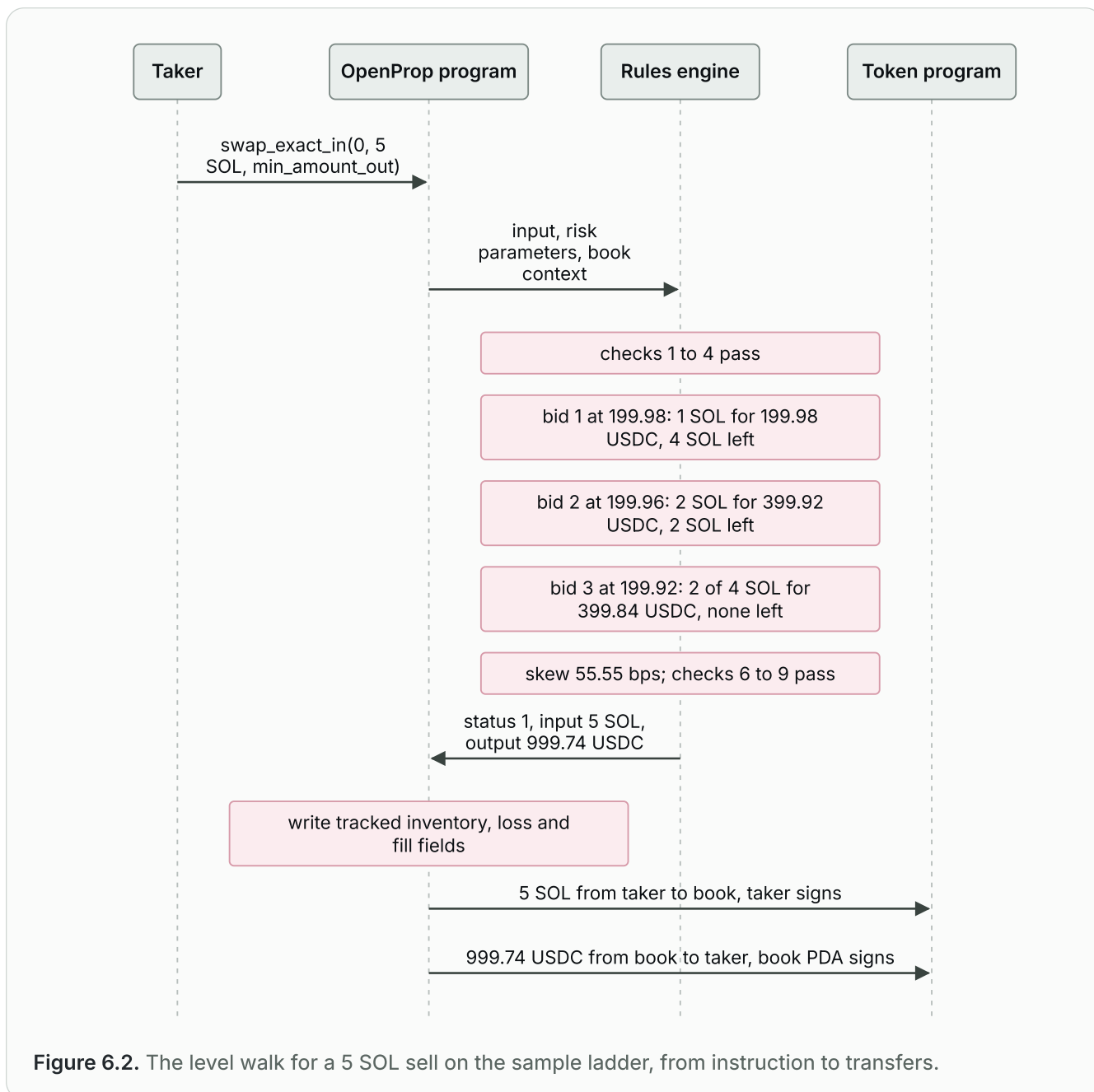
Level	Offset	Price (USDC)	Affordable (SOL)	Filled (SOL)	Charged (USDC)	Remaining (USDC)
1	1 bps	200.02	4.999500049	1.000000000	200.020000	799.980000
2	2 bps	200.04	3.999100179	2.000000000	400.080000	399.900000
3	4 bps	200.08	1.998700519	1.998700519	399.900000	0
Total				4.998700519	1,000.000000	

At level 3 the remaining 399.90 USDC buys $\lfloor 399,900,000 \cdot 10^9 / 200,080,000 \rfloor = 1,998,700,519$ lamports. Their exact cost is 399.89999984 USDC, which the ceiling charges as 399.900000. The input is consumed exactly. The taker receives 4.998700519 SOL for 1,000 USDC, an average of 200.052 USDC per SOL, 2.60 bps above the mid. The book ends at 895.001299481 SOL and 181,000 USDC, an excess gain of 0.259896 USDC at the reference, with skew again 55.55 bps.

The same swaps as retail flow

With the configured retail signer present, each offset shrinks by the quote's 1 bps retail delta, so level 1 trades at the mid:

Swap	Level prices used (USDC)	Output	Average price
Sell 5 SOL, retail	200.00, 199.98, 199.94	999.840000 USDC	199.968
Buy with 1,000 USDC, retail	200.00, 200.02, 200.06	4.999200239 SOL	200.032



6.5 The two token transfers

When the result status is nonzero, the program writes the book (§6.6), releases its account borrows and makes exactly two SPL Token Transfer calls:

Side	Transfer 1: input	Transfer 2: output
Sell SOL	input_consumed lamports, taker SOL → book SOL, signed by the taker	output_amount micro-USDC, book USDC → taker USDC, signed by the book PDA
Buy SOL	input_consumed micro-USDC, taker USDC → book USDC, signed by the taker	output_amount lamports, book SOL → taker SOL, signed by the book PDA

The book PDA signs with seeds ["book", quoter, pair, bump]. Only swap_exact_in and epoch settlement can make the book sign. The quoter's keys cannot move book tokens. The design estimates the two transfers at about 10k CU of the swap's cost.

6.6 Inventory tracking on the book

The rules engine never reads token balances. It works on the book's tracked inventory, trackedBase and trackedQuote, which only swaps and settlement change. The program requires the real balances of the book's token accounts to be at least the tracked amounts. Tokens sent to a book unsolicited are therefore excluded from its performance, and settlement sweeps them to the LP vault. On success the program writes:

Fields	Value written
trackedBase, trackedQuote	post-fill inventory from the walk
hourlyStartSlot, hourlyStartExcess	the loss window, reset if it expired (\$5.8)
hourlyLoss, epochLoss, epochExcess	the post-fill marks
epochFillCount	incremented
lastFillSlot, lastFillSequence	Clock.slot and the Quote.sequence that was filled
lastFillInput, lastFillOutput, lastFillSide	consumed input, output, side
scoreBaseDelta, scoreQuoteDelta	signed inventory change, added only for external takers
epochScoredFillCount, epochFillNotional	incremented, and filled SOL marked at RefPrice.mid, only for external takers

A taker is external when it is neither the book's cold owner nor one of the hot authorities cached in the Quote. A quoter's own fills change the real inventory and count fully toward losses, excess and settlement, but they add nothing to the scoring statistics or the turnover that allocation reads.

6.7 The retail flow class and retail delta

Each quote carries a retail delta, Quote.retailDeltaBps (ρ), that tightens every offset for flow the program can attribute to retail users. The attributing credential is a signature from the aggregator frontend signer recorded in Config.retailSigner. A swap is priced as retail when an eleventh account is present, has signed, and equals Config.retailSigner. Any other eleventh account is priced as standard flow without error. A zero retailSigner disables the class, and the key changes only through rotate_identities after the timelock. The delta is bounded per quote by Config.retailDeltaLimitBps, and a quote above it fails check 2 with QUOTE_LADDER_INVALID. An offset smaller than the delta is floored at zero, so retail pricing never crosses the mid. The Jupiter adapter quotes standard pricing, because only the on-chain signer check unlocks the retail offsets.

6.8 Compute cost

The program's compute use is measured by LiteSVM 0.6.1 executing the SBF release artifact, and recorded in `program/cu-measurements.json`:

Path	Measured CU	Budget
update_quote, indexed authority	178	200
swap_exact_in, representative three-level swap	17,934	40,000
swap_exact_in, five-level sell	20,911	40,000
swap_exact_in, five-level buy	20,874	40,000
swap_exact_in, skew partial on a small book	30,974	40,000
swap_exact_in, skew partial on a large book	37,497	40,000

The highest observation, 37,497 CU, is the figure the measurement file reports for `swap_exact_in`. The skew-partial cases cost more than full five-level walks because of the binary search for the largest admissible quantity at the cap. Every swap writes its Book, so the 12M CU cap per writable account limits fills per book per block: the design estimates about 300 fills per book per block at 40k CU per swap. Liquidity spread across several books multiplies that capacity.

7. The vault and LP shares

Each pair has one liquidity vault. The `LpVault` account records the share supply, the withdrawal queue and the registry of books. Two SPL token accounts under the vault's authority hold the SOL and USDC that are not allocated to a book, and a classic SPL Token mint issues the vault's shares, OPLP. This chapter gives the account layout, the valuation the program uses to convert between tokens and shares, and the arithmetic of deposit, `request_withdraw` and `claim`. How vault inventory is divided among books is the subject of chapter 8.

7.1 Accounts

`initialize_pair`, signed by the admin, creates the vault, its two token accounts and the share mint in the same transaction as the pair's `PairConfig` and `RefPrice`. All of them are program-derived addresses.

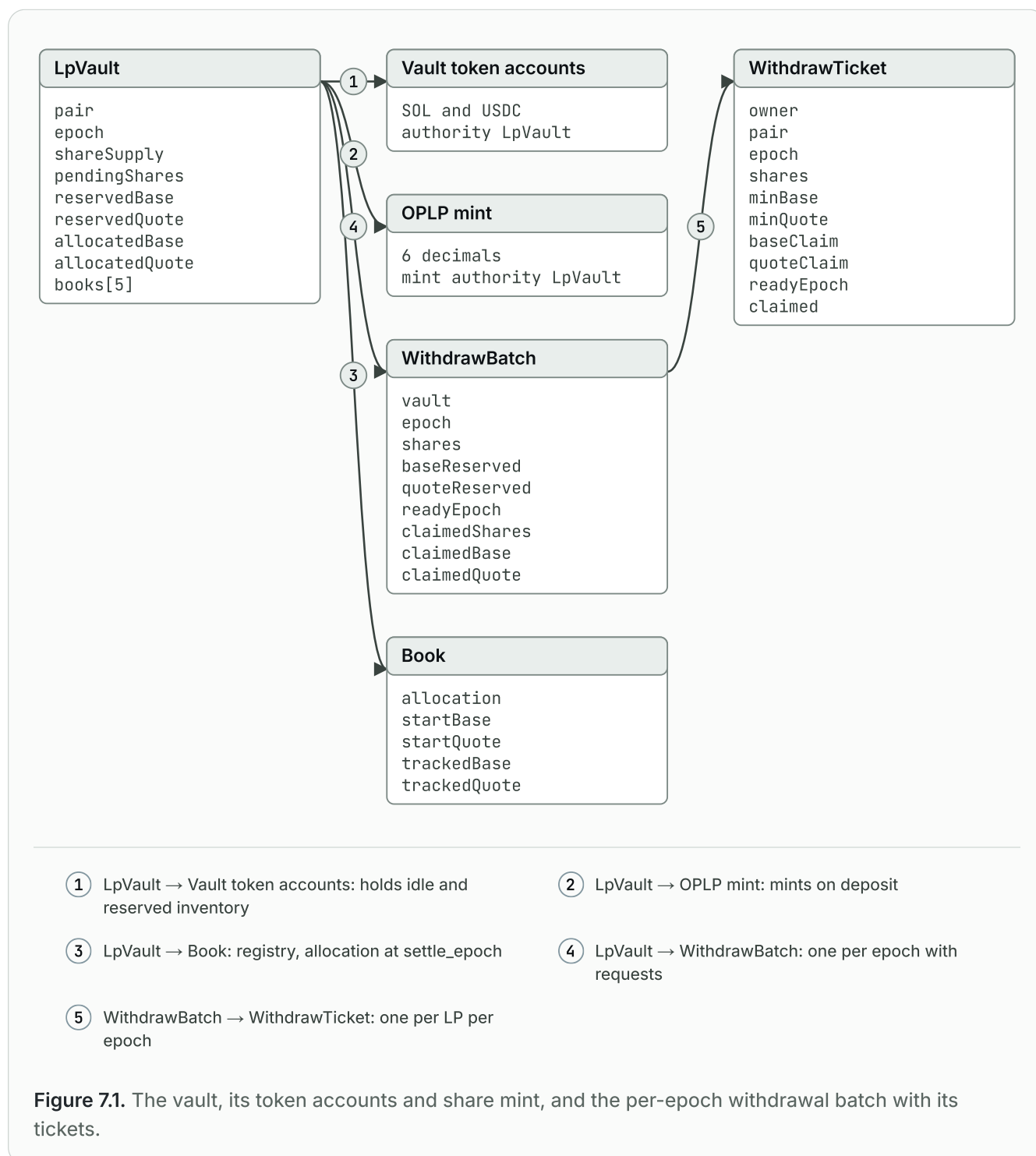
Account	Seeds	Owning program	Authority
LpVault	["vault", pair]	OpenProp	—
Vault SOL account	["vault_base", pair]	SPL Token	LpVault
Vault USDC account	["vault_quote", pair]	SPL Token	LpVault
OPLP share mint	["shares", pair]	SPL Token	mint authority LpVault, 6 decimals
WithdrawBatch	["withdraw_batch", vault, epoch]	OpenProp	—
WithdrawTicket	["wd", owner, epoch]	OpenProp	—
OperationReceipt	["receipt", owner, requestId]	OpenProp	—

The `LpVault` account is 408 bytes:

Field	Content
pair, baseToken, quoteToken, shareMint	Bindings to the pair and to the three SPL accounts above
epoch, epochStartSlot, lastSettledEpoch	The epoch clock. Epoch E runs from <code>epochStartSlot</code> for <code>epochSlots</code> slots (432,000 slots, about 24 hours at 200 ms)
shareSupply	OPLP in circulation
pendingShares	Shares burned by withdrawal requests of the current epoch and not yet reserved
reservedBase, reservedQuote	Tokens set aside for settled withdrawal batches and not yet claimed
allocatedBase, allocatedQuote	Tokens sent to books at the last settlement
bookCount, books	The registry: up to five books in canonical order
lastPrice, lastNav	Valuation price and free NAV of the last settlement

Field	Content
deferredQuoterFees, deferredProtocolFees	Performance fees owed and not yet paid

The vault's token accounts are debited by two instructions only. `claim` pays the owner of a withdrawal ticket. `settle_epoch` moves inventory to book token accounts and pays performance fees to quoter bond accounts and the protocol USDC account. No admin, guardian or quoter instruction transfers tokens out of these accounts.



7.2 Units and valuation

Amounts are integers in token atoms: SOL in lamports (10^{-9} SOL), USDC in micros (10^{-6} USDC), OPLP in share atoms (10^{-6} OPLP). The reference price P is `RefPrice.mid`, in USDC micros per whole SOL. The program values an inventory of b lamports and q micros as

$$V(b, q; P) = \left\lfloor \frac{b \cdot P}{10^9} \right\rfloor + q$$

where V is the value in USDC micros, rounded down on the SOL leg.

`deposit` accepts the reference when its mid is positive, its source slot is at most `refStalenessSlots` behind the current slot (10 slots in `v0`) and the pair is not halted. `settle_epoch` additionally requires the confidence and the disagreement between the two oracle sources to be inside `confidenceBps` and `divergenceBps`; it can value inventory while swaps are halted by a price-move breaker.

7.3 NAV and share price

Free physical basket

The vault's inventory is spread over its own token accounts and the token accounts of every registered book. The free basket counts all of it except tokens already reserved for withdrawal claims:

$$B = B_v + \sum_{i=1}^n B_i - R_B, \quad Q = Q_v + \sum_{i=1}^n Q_i - R_Q$$

where B_v, Q_v are the vault's SOL and USDC token balances, B_i, Q_i the token balances of book i , n the `bookCount`, and R_B, R_Q the vault's `reservedBase` and `reservedQuote`.

Economic NAV

Between settlements, books carry gains and losses that settlement will turn into bond charges and fees. `deposit` prices them as if the epoch settled now, so that a new LP neither buys into a loss the bond has yet to cover nor acquires fees owed to quoters and the protocol:

$$\text{NAV}^{\text{econ}} = V(B, Q; P) + \sum_{i=1}^n (c_i - f_i) - D$$

where

- $x_i = V(\hat{b}_i, \hat{q}_i; P) - V(b_i^0, q_i^0; P)$ is book i 's excess over hold this epoch, with tracked inventory $\hat{b}_i, \hat{q}_i$ (`trackedBase`, `trackedQuote`) and starting inventory b_i^0, q_i^0 (`startBase`, `startQuote`);
- $c_i = \min(\max(0, -x_i), \beta_i + d_i)$ is the first-loss coverage the book's quoter would supply, with β_i the quoter's bond and d_i the book's unpaid quoter credit (`deferredQuoterFees`);
- $e_i = \min(x_i, \max(0, X_i + x_i - H_i))$ for $x_i \geq 0$, else 0, is the new excess above the high-water mark, with X_i the book's `cumulativeExcess` and H_i its `highWaterMark`;
- $f_i = e_i - \lfloor e_i \cdot \text{lpShareBps} / 10^4 \rfloor$ is the quoter and protocol part of e_i (50% of it at the `v0` 50/40/10 split);
- D is the sum of the vault's `deferredQuoterFees` and `deferredProtocolFees`.

The LP NAV is $NAV = \max(0, NAV^{\text{econ}})$. While $NAV^{\text{econ}} < 0$, deposits fail with FEE_LIABILITY_SHORTFALL; withdrawals and settlement continue.

Economic supply and virtual offsets

Shares burned by a withdrawal request still own part of the free basket until settlement reserves their tokens, so the program prices against the economic supply

$$S = S_{\text{act}} + S_{\text{pend}}$$

where S_{act} is shareSupply and S_{pend} is pendingShares.

Two configuration values, virtualShares (V_s , share atoms) and virtualQuoteAtoms (V_q , USDC micros), are added to the supply and to the NAV in every conversion. Configuration validation requires

$$V_q \cdot 10^6 = \pi_0 \cdot V_s$$

where π_0 is initialSharePrice, in USDC micros per whole OPLP. The offsets fix the conversion rate of an empty vault at π_0 and keep the rate of early deposits from being set by tokens sent to the vault accounts outside deposit. No virtual asset is ever transferred; the offsets exist only in the arithmetic.

Share prices

The program and the API use two prices, in USDC per OPLP:

$$p_r = \frac{NAV}{S + V_s}, \quad p_d = \frac{NAV + V_q}{S + V_s}$$

where p_r is the redemption value of one share (sharePriceMicros in the API) and p_d the price at which deposits are converted (depositSharePriceMicros). Writing $p = NAV/S$ for the vault's own per-share value,

$$p_d = \frac{S}{S + V_s} p + \frac{V_s}{S + V_s} \cdot \frac{\pi_0}{10^6}$$

so the deposit price is a weighted mean of the vault's per-share value and the initial price, with weight $V_s/(S + V_s)$ on the initial price. That weight is 1 for an empty vault and falls toward zero as the supply grows.

7.4 Deposits

The instruction

deposit takes four u64 values: requestId, maxBaseAtoms, maxQuoteAtoms and minShares. The two maxima are spending limits, not exact amounts. The accounts are the owner (signer), Config, RefPrice, the vault with its token accounts and share mint, the owner's SOL, USDC and OPLP token accounts, the receipt, the token and system programs, and then every registered book as a group of five accounts (book, book SOL, book USDC, quoter, bond) in registry order. A deposit that omits a book fails, so the program always sees the whole basket and every pending bond charge and fee.

The program processes it in this order:

1. **Receipt.** The `OperationReceipt` at `["receipt", owner, requestId]` binds the original payload, including both maxima. A retry with the same payload returns without effect; a different payload under the same `requestId` fails with `REPLAY_PAYLOAD_MISMATCH`.
2. **Epoch window.** The current slot must be below `epochStartSlot + epochSlots`, else `EPOCH_NOT_READY`. Deposits close at the nominal end of an epoch and reopen when `settle_epoch` advances the vault to the next one.
3. **Reference.** Fresh and not halted (section 7.2), else `REF_STALE`.
4. **Registry.** Each book's token balances must cover its tracked inventory, and each bond account must cover the recorded bond.
5. **Basket, NAV, cap and shares**, as below.
6. **Transfers.** The selected SOL and USDC move from the owner to the vault token accounts, the vault mints s OPLP to the owner, and `shareSupply` grows by s .

Basket selection

With $S > 0$, the program takes the largest basket in the free basket's physical SOL/USDC ratio that fits both maximums. With maxima $\bar{b}$ and $\bar{q}$:

$$(b, q) = \begin{cases} (\bar{b}, \lfloor \bar{b} Q / B \rfloor) & \text{if } \bar{b} Q \leq \bar{q} B \\ (\lfloor \bar{q} B / Q \rfloor, \bar{q}) & \text{otherwise} \end{cases}$$

where (b, q) is the basket actually transferred. The limiting maximum is consumed in full and its counterpart is rounded down to whole atoms; the unused tokens stay in the owner's accounts. The ratio comes from custody at execution, not from the snapshot the wallet signed against. The program requires $B > 0$ and $Q > 0$ (`NON_PROPORTIONAL_DEPOSIT`) and $b > 0, q > 0$ (`INVALID_AMOUNT`). Because a deposit adds both tokens in the vault's existing proportions, it leaves the SOL exposure per share of existing holders unchanged.

For the first deposit ($S = 0$) there is no ratio to follow. Both maximums are consumed in full and must have equal reference-marked value, exactly to the atom:

$$\left\lfloor \frac{\bar{b} \cdot P}{10^9} \right\rfloor = \bar{q}$$

otherwise the deposit fails with `NON_PROPORTIONAL_DEPOSIT`. At a 200.00 USDC reference, 10 SOL must be paired with exactly 2,000 USDC. The first basket therefore sets the vault at equal value in SOL and USDC.

Shares, minimum and cap

With $A = V(b, q; P)$, the value of the consumed basket:

$$s = \left\lfloor \frac{A \cdot (S + V_s)}{\text{NAV} + V_q} \right\rfloor$$

where s is the number of share atoms minted. The deposit requires

$$s \geq \max(1, \text{minShares}), \quad \text{NAV} + A \leq \text{vaultCapQuoteAtoms}$$

failing with `MINIMUM_SHARES` or `VAULT_CAP_EXCEEDED`. Only consumed value counts toward the cap, so unused maximums never cause a rejection. The v0 cap is 2,000,000 USDC. For the first deposit into an empty vault

(NAV = 0), the formula reduces to $s = A \cdot 10^6 / \pi_0$: the first LP buys at the initial share price.

Worked example

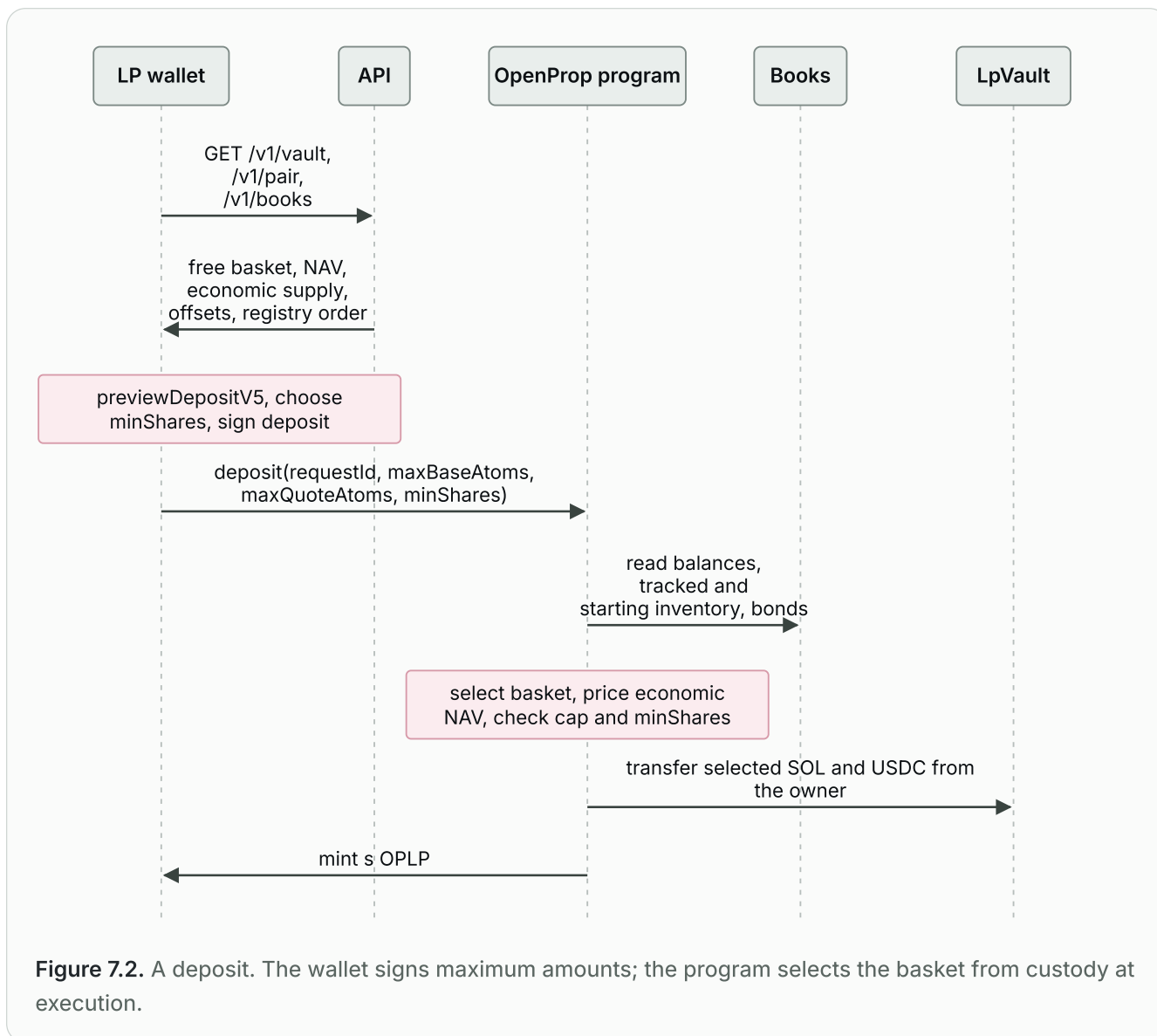
Take a vault with this state:

Quantity	Value
Reference P	200.00 USDC
Free basket B, Q	3,500 SOL, 500,000 USDC
$V(B, Q; P)$	$3,500 \times 200 + 500,000 = 1,200,000$ USDC
Pending charges and fees, D	none
NAV	1,200,000 USDC
Deposit price p_d (from S and the offsets)	1.200000 USDC per OPLP
<code>vaultCapQuoteAtoms</code>	2,000,000 USDC

An LP signs `maxBaseAtoms` = 10 SOL (10^{10} lamports) and `maxQuoteAtoms` = 1,000 USDC (10^9 micros).

1. Limiting side: $\bar{b}Q = 10 \times 500,000 = 5,000,000$ and $\bar{q}B = 1,000 \times 3,500 = 3,500,000$. Since $\bar{b}Q > \bar{q}B$, USDC is the limiting token.
2. Basket: $q = 1,000$ USDC and $b = \left\lfloor \frac{10^9 \cdot 3.5 \times 10^{12}}{5 \times 10^{11}} \right\rfloor = 7 \times 10^9$ lamports, so 7 SOL. The other 3 SOL stay in the wallet. The ratio used is the vault's physical 142.857 USDC per SOL, not the 200.00 reference.
3. Value: $A = 7 \times 200 + 1,000 = 2,400$ USDC.
4. Cap: $1,200,000 + 2,400 = 1,202,400 \leq 2,000,000$. The vault had 800,000 USDC of headroom before this deposit.
5. Shares: $s = 2,400 / 1.2 = 2,000$ OPLP, i.e. 2×10^9 share atoms.
6. Minimum: a wallet that previewed 2,000 OPLP and signed `minShares` = 1,990 OPLP accepts this result. If fills in the books had changed the free basket between preview and execution, the program would recompute b, q and s at execution and reject an issuance below 1,990 OPLP.

The app computes the same preview with `previewDepositV5` from the `/v1/vault` and `/v1/pair` reads, then simulates the transaction before the wallet signs.



7.5 Withdrawals

A withdrawal takes three steps across two epochs: a request in epoch E , a reservation inside $\text{settle_epoch}(E)$, and a claim in epoch $E + 1$ or later.

Request

`request_withdraw` takes four u64 values: epoch, shares, minBase and minQuote. The epoch must equal the vault's current epoch. The program:

1. creates the `WithdrawTicket` at `["wd", owner, epoch]` with the shares, the two minimums and `readyEpoch = E + 1`;
2. creates the epoch's `WithdrawBatch` at `["withdraw_batch", vault, epoch]` if this is the epoch's first request;
3. burns the shares from the owner's OPLP account;
4. moves them from `shareSupply` to `pendingShares` and adds them to the batch's shares.

The shares stop circulating at once, but they stay in the economic supply S and keep their claim on the whole free basket until settlement. One ticket exists per LP per epoch; repeating a request with the same payload is a no-op, and a different payload fails with `REPLAY_PAYLOAD_MISMATCH`. Requests need no reference price, so they are accepted while the pair is halted.

Reservation at settlement

`settle_epoch(E)` first recalls all book inventory to the vault and pays the epoch's fees (chapter 8). It then sets aside the pending shares' part of what is free:

$$\Delta R_B = \left\lfloor \frac{B^{\text{free}} \cdot S_{\text{pend}}}{S_{\text{act}} + S_{\text{pend}} + V_s} \right\rfloor, \quad \Delta R_Q = \left\lfloor \frac{Q^{\text{free}} \cdot S_{\text{pend}}}{S_{\text{act}} + S_{\text{pend}} + V_s} \right\rfloor$$

where

- $B^{\text{free}} = B_v - R_B - B^{\text{held}}$ and $Q^{\text{free}} = Q_v - R_Q$, with the vault balances B_v, Q_v measured after the recall and fee payments;
- D' is the fees this settlement could not pay from unreserved USDC; they become the vault's new `deferredQuoterFees` and `deferredProtocolFees`;
- $B^{\text{held}} = \min(B_v - R_B, \lceil D' \cdot 10^9 / P \rceil)$, or 0 when $D' = 0$, is SOL of equal reference value kept idle against those fees;
- $\Delta R_B, \Delta R_Q$ are the batch's `baseReserved` and `quoteReserved`.

The program adds ΔR_B and ΔR_Q to the vault's `reservedBase` and `reservedQuote`, sets the batch's `readyEpoch` to $E + 1$ and resets `pendingShares` to zero. From then on the batch's payout is fixed in tokens: it no longer takes part in book performance, allocation or NAV. The reservation is in kind, in the free basket's SOL/USDC proportions, and the virtual shares' fraction $V_s / (S + V_s)$ stays in the vault with the remaining holders.

Claim

`claim` takes the ticket's epoch and, optionally, two replacement minimums signed by the owner. It requires the vault's epoch to be at least the batch's `readyEpoch`, else `WITHDRAW_NOT_READY`. A ticket with s_j shares in a batch of s_T shares receives

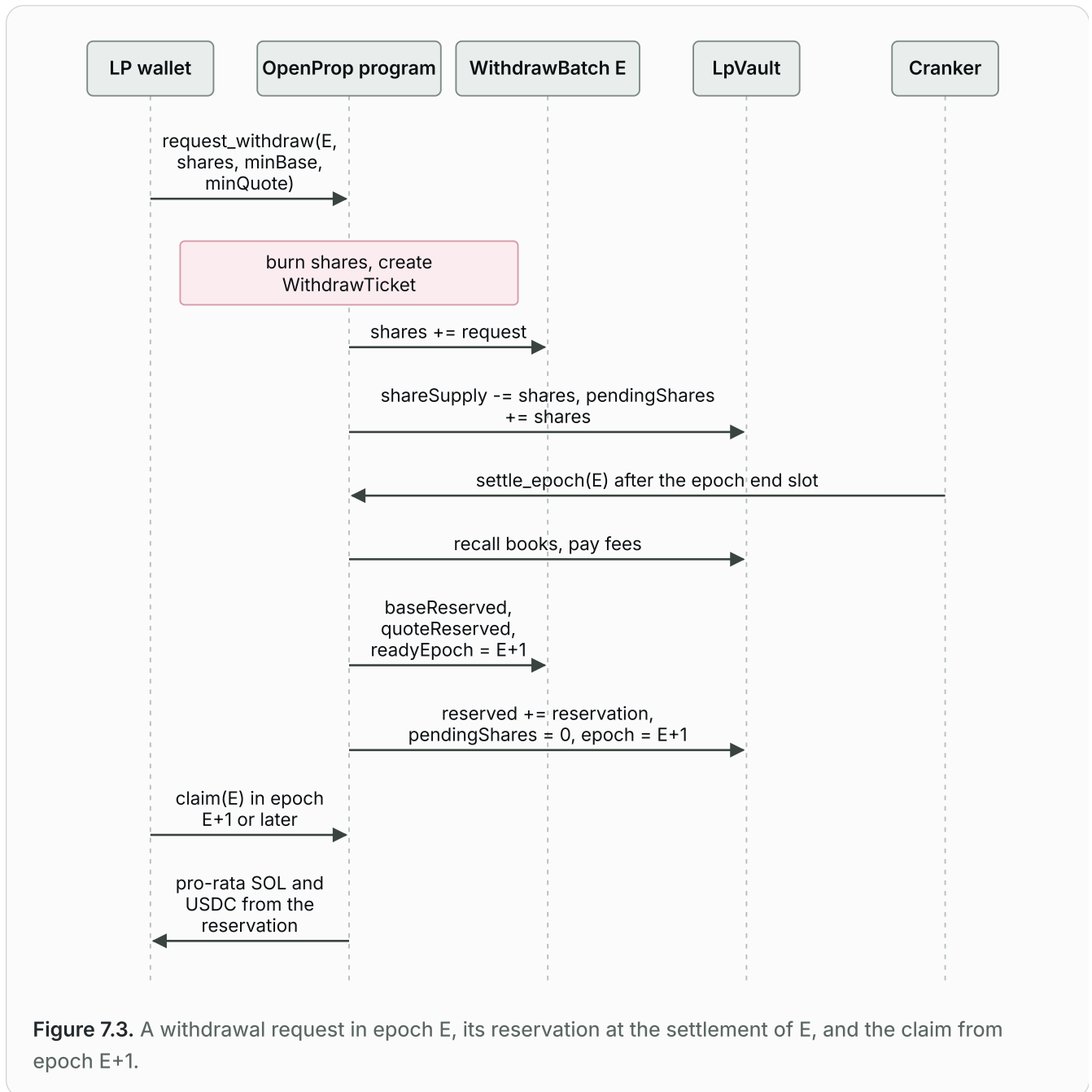
$$b_j = \left\lfloor \frac{\Delta R_B \cdot s_j}{s_T} \right\rfloor, \quad q_j = \left\lfloor \frac{\Delta R_Q \cdot s_j}{s_T} \right\rfloor$$

except that the claim completing the batch receives `baseReserved - claimedBase` and `quoteReserved - claimedQuote`, so rounding remainders are paid out and the batch closes at zero. Both amounts must meet the minimums (`SLIPPAGE_EXCEEDED` otherwise). The program transfers them from the vault's token accounts, lowers `reservedBase` and `reservedQuote`, and marks the ticket claimed; a second claim is a no-op.

Maximum delay

Every request is served by the first settlement after it: a request in epoch E is reserved by `settle_epoch(E)` and claimable from epoch $E + 1$. The wait is therefore at most one epoch (432,000 slots) plus the time between the epoch's end slot and the settlement transaction, which any cranker may send. Withdrawals never wait for books to reach a particular inventory, because settlement recalls every book before reserving.

The v0 design bounds the queue at two epochs. The deployment records its bound as `withdrawEpochs`, which is fixed for a deployed pair and must be positive and smaller than `timeLockEpochs` (3 epochs in v0). A parameter change that loosens a limit can activate only after both $\text{slot} \geq \text{proposal slot} + \text{timeLockEpochs} \cdot \text{epochSlots}$ and $\text{vault epoch} \geq \text{proposal epoch} + \max(\text{withdrawEpochs}_{\text{current}}, \text{withdrawEpochs}_{\text{proposed}}) + 1$. An LP who requests a withdrawal when a loosening is proposed is reserved and claimable before the change takes effect.



7.6 Idle and allocated inventory

At any slot the pair's LP inventory sits in three places:

Place	Contents	Changes during the epoch
Vault token accounts, reserved part	Unclaimed baskets of settled batches (R_B, R_Q)	Falls with each claim
Vault token accounts, idle part	Inventory not allocated at the last settlement, plus deposits since	Rises with each deposit
Book token accounts	Allocations made at the last settlement	Changes with each fill in that book

Allocation happens only inside `settle_epoch`. The settlement recalls every book's entire token balances to the vault, including any tokens sent to a book outside the program, which become LP assets and never count as book performance. It charges bonds, pays fees and reserves the withdrawal batch, then transfers to each book a slice of the free basket in the free basket's own ratio:

$$b_i = \left\lfloor \frac{B^{\text{alloc}} \cdot t_i}{\text{NAV}^{\text{free}}} \right\rfloor, \quad q_i = \left\lfloor \frac{Q^{\text{alloc}} \cdot t_i}{\text{NAV}^{\text{free}}} \right\rfloor$$

where $B^{\text{alloc}} = B^{\text{free}} - \Delta R_B$ and $Q^{\text{alloc}} = Q^{\text{free}} - \Delta R_Q$ are the tokens left after the reservation, $\text{NAV}^{\text{free}} = V(B^{\text{alloc}}, Q^{\text{alloc}}; P)$ is their value, recorded as `lastNav`, and t_i is book i 's target allocation in USDC micros (chapter 8). The sums of b_i and q_i become `allocatedBase` and `allocatedQuote`, and each book's starting and tracked inventory are set to its (b_i, q_i) . What is not allocated stays idle in the vault: the sum of targets never exceeds NAV^{free} , and the caps of chapter 8 often keep it below.

Idle inventory does not trade during the epoch; it is the part of the basket that new deposits join and that the next settlement can allocate. Every share carries the same pro-rata slice of idle, allocated and pending-settlement value, because deposits are priced over the whole free basket across vault and books, and reservations are taken after every book has been recalled.

8. Bonds, tiers and allocation

A quoter's access to LP inventory is bounded by its USDC bond and its tier, and set inside that bound by the score of its book. All three live in program accounts, and `settle_epoch` turns them into the next epoch's allocations with integer arithmetic that anyone can recompute from chain state. This chapter covers the bond's lifecycle and its role as first loss, the tier rules, the score, and the allocation step, and ends with a three-book example.

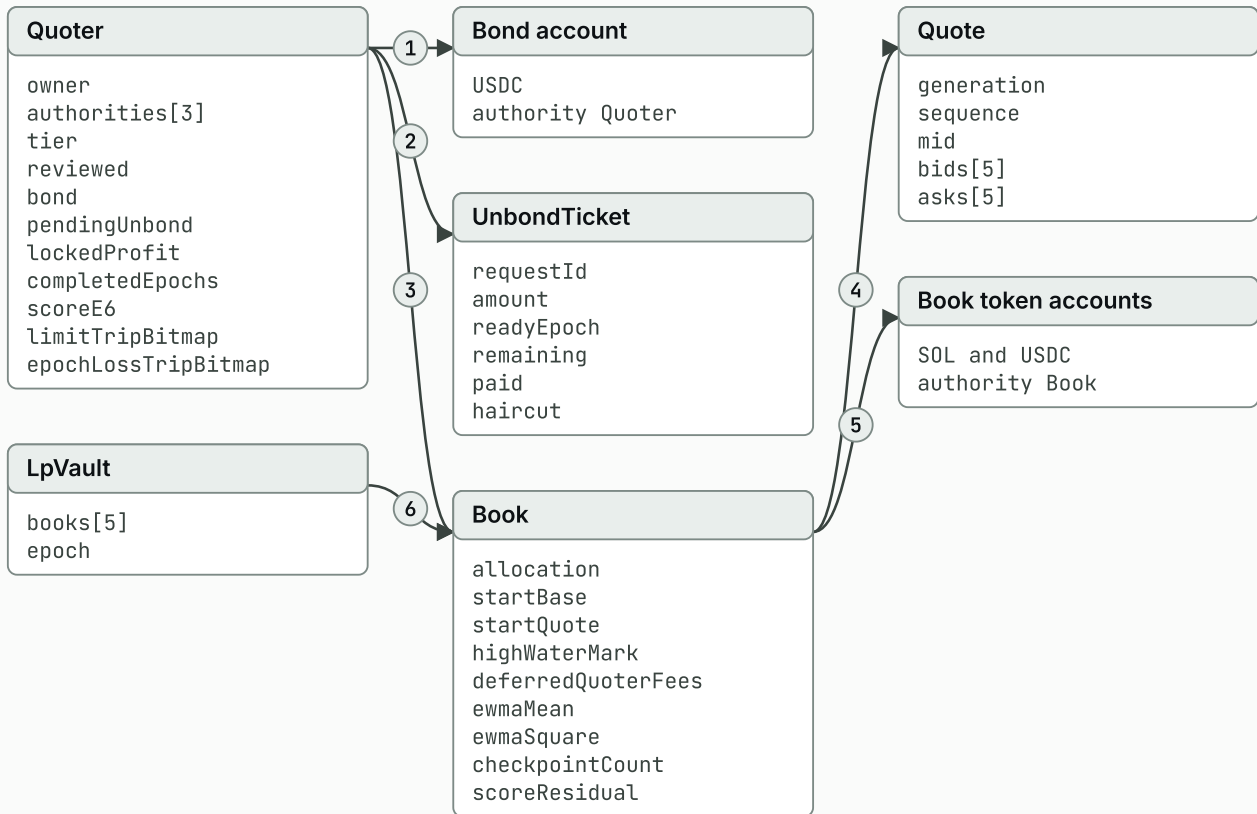
8.1 Registration and the bond

Registration

`register_quoter` takes a `requestId` and a bond amount in USDC micros and is signed by the quoter's cold owner key. It requires the bond to be at least `minBondQuoteAtoms` (else `INVALID_AMOUNT`) and the registry to have a free seat (`bookCount` below `maxBooks`, at most five; else `REGISTRY_FULL`). In one transaction it creates:

Account	Seeds	Size	Content
Quoter	["quoter", owner]	1,304 B	Owner, hot authorities, tier, bond, unbond and profit-lock state, trip history
Bond account	["bond", quoter]	SPL token	USDC, authority Quoter
Book	["book", quoter, pair]	576 B	Allocation, inventory, excess, loss windows, scoring state
Quote	["quote", book]	304 B	The ladder written by <code>update_quote</code>
Book token accounts	["book_base", book], ["book_quote", book]	SPL token	Allocated SOL and USDC, authority Book

It appends the book to the vault's books registry, transfers the bond from the owner's USDC account and records it in the quoter's bond field. The quoter starts at tier 0 with zero allocation; its first allocation arrives at the next settlement. The owner then names up to three hot keys with `set_quote_authority`. Each newly added key co-signs its addition, which matters for the self-flow rule in section 8.3.



- ① Quoter → Bond account: first-loss collateral
- ② Quoter → UnbondTicket: one per unbond request
- ③ Quoter → Book: one book in the pair
- ④ Book → Quote: current ladder
- ⑤ Book → Book token accounts: allocated inventory
- ⑥ LpVault → Book: registry entry, allocation at settle_epoch

Figure 8.1. The accounts of one quoter seat. The cold owner manages the bond and the hot keys; only the program moves book and vault tokens.

Adding bond

`add_bond` takes a `requestId` and a positive amount, transfers it to the bond account and adds it to `bond`. Bond added during an epoch raises the leverage cap at the next settlement. Both `register_quoter` and `add_bond` write an `OperationReceipt`, so a retried transaction never transfers twice.

Collateral in use

At tier τ , a book with allocation a ties up

$$\kappa = \left\lceil \frac{a}{L(\tau)} \right\rceil$$

where a is the book's allocation in USDC micros, $L(\tau)$ is `tier0Leverage`, `tier1Leverage` or `tier2Leverage` (1, 5 and 15 in v0) and κ is the bond the allocation requires.

Requesting an unbond

request_unbond takes a requestId and an amount n . With the vault in epoch E , it accepts

$$n \leq \max(0, \beta - u - \kappa - \Lambda(E + U))$$

where β is bond, u is pendingUnbond, U is unbondEpochs (at least 2, enforced by configuration), and $\Lambda(e)$ is the sum of profit tranches whose unlock epoch is later than e . A larger request fails with UNBOND_EXCEEDS_AVAILABLE. The program creates an UnbondTicket at ["unbond", quoter, requestId] with amount = n , remaining = n and readyEpoch = $E + U$, and adds n to pendingUnbond. From the next settlement on, the pending amount no longer counts toward the leverage cap (section 8.4), so the book's allocation shrinks to fit the bond that stays.

Claiming an unbond

claim_unbond takes the requestId and requires the vault's epoch E' to be at least readyEpoch (else UNBOND_NOT_READY). With n_{rem} the ticket's remaining:

$$\phi = \max(0, \beta - \kappa - \Lambda(E')), \quad \text{payout} = \min\left(n_{\text{rem}}, \left\lfloor \frac{\phi \cdot n_{\text{rem}}}{u} \right\rfloor\right)$$

where ϕ is the bond free of allocation collateral and locked profit. Free bond is shared pro rata among all open tickets through the factor n_{rem}/u . A payout below n_{rem} is a partial claim: the ticket stays open for the rest and can be claimed again once collateral or locks release more bond. A remainder is written off as a haircut only when the bond has truly fallen short, that is when $\kappa = 0$, $\Lambda(E') = 0$ and $\beta < u$. Every ticket keeps amount = paid + haircut + remaining, and each claim emits an unbondClaim event with the four values.

Example at tier 1: bond 50,000 USDC, allocation 200,000 USDC, so $\kappa = 40,000$; a profit tranche of 1,500 USDC unlocks after $E + U$. The available amount is $50,000 - 0 - 40,000 - 1,500 = 8,500$ USDC. A request for 8,500 USDC is accepted; 9,000 USDC fails. At the next settlement the leverage cap becomes $(50,000 - 8,500) \times 5 = 207,500$ USDC.

A quoter that is fully unbonded and flat (zero bond, pending unbond, locked profit, allocation, tracked inventory and unpaid fees) can call deregister_quoter, which frees its registry seat without moving tokens; otherwise it fails with BOND_IN_USE.

The bond as first loss

At settlement, each book's epoch excess over hold x is applied to its bond before it reaches LPs. With X the book's cumulativeExcess, H its highWaterMark, β the quoter's bond and d the book's unpaid quoter credit (deferredQuoterFees):

$$x < 0: \quad c = \min(-x, \beta + d), \quad c_{\text{tok}} = \min(\beta, c), \quad \ell = -x - c, \quad X' = X + x + c$$

$$x \geq 0: \quad e = \min(x, \max(0, X + x - H)), \quad X' = X + x$$

and in both cases $H' = \max(H, X')$, where c is the first-loss charge, c_{tok} the part paid in USDC from the bond account to the vault (the rest cancels unpaid quoter credit), ℓ the loss that falls on LPs, and e the new excess above the high-water mark. The charge restores cumulative excess, so a quoter whose bond paid for a loss does not have to earn it back a second time before its high-water mark is reached. Positive e is

split by `lpShareBps` and `quoterShareBps` with the protocol taking the remainder (50/40/10 in v0). The quoter's share is paid from unreserved vault USDC into the bond account and locked as a profit tranche for `profitLockEpochs`; a share that cannot be paid stays as `deferredQuoterFees`, which absorbs losses like bond but does not raise the leverage cap. Profit therefore adds first-loss capacity before it becomes withdrawable.

8.2 Tiers

Tier	Leverage	Bond per unit of allocation	Entry rule, checked at settlement
0	1	100%	Registration
1	5	20%	<code>completedEpochs</code> $\geq$ <code>tier1Epochs</code> (14), score above zero, no limit trip in the last <code>downgradeWindowEpochs</code> (7) epochs
2	15	6.7%	<code>completedEpochs</code> $\geq$ <code>tier2Epochs</code> (45), score at or above <code>tier2ScoreE6</code> , guardian review recorded

`completedEpochs` grows by one at a settlement only if the book was live, held a nonzero allocation during the epoch and received at least one eligible (non-self) fill. Idle or self-traded epochs do not count. Promotion also requires an eligible fill in the settled epoch and moves at most one tier per settlement. The tier-2 review is a flag, reviewed, set by `review_tier` and signed by the guardian key; the separate halt-only delegate cannot set it.

The quoter's trip history is kept in two 64-bit bitmaps whose bit 0 is the current epoch; each settlement shifts them left by one after the tier decision. A checkpoint that halts the book on the hourly or the epoch loss limit sets bit 0 of `limitTripBitmap`; an epoch-loss halt also sets bit 0 of `epochLossTripBitmap`. With T the `epochLossTripBitmap`, $W = \text{downgradeWindowEpochs}$ and the mask $2^W - 1$ selecting the last W epochs, the drop rule is

$$\text{drop one tier if } \text{bit}_0(T) = 1 \text{ and } \text{popcount}(T \& (2^W - 1)) \geq \text{downgradeTrips}$$

With the v0 values, the second epoch-loss trip within 7 epochs drops the quoter one tier at the settlement of the epoch in which it happens. A drop takes precedence over promotion, and tier 0 is the floor.

The tier's leverage fixes the bond per unit of allocation at $1/L$: 100%, 20% and 6.7%. Fills stop once a book's loss in the epoch exceeds 2% of its allocation (`epochLossBps`) or its loss in the hourly window exceeds 0.5% (`hourlyLossBps`). At tier 2 the bond is therefore 3.3 times the epoch loss limit, and at the lower tiers a larger multiple.

8.3 The score

The score measures a book's excess over hold on flow it did not originate, per unit of allocation, as a mean-to-volatility ratio with a minimum hurdle and a shrinkage for short histories. It is updated by `checkpoint_book`, which cranks call about every 1,500 slots (at most once per `checkpointSlots`), and once more inside `settle_epoch`.

Eligible flow

On every fill, `swap_exact_in` compares the taker with the book's cold owner and with the hot authorities cached in the `Quote`. A fill by any of them is self-flow: it changes tracked inventory, excess, loss windows and settlement like any fill, but adds nothing to the scoring fields. Every other fill adds its signed inventory changes to `scoreBaseDelta` and `scoreQuoteDelta`, increments `epochScoredFillCount` and adds its reference-marked notional to `epochFillNotional`. Because new hot keys must co-sign their addition, an owner cannot mark an unrelated taker as self-flow. Fills from other wallets are scored as external flow; the mean-return hurdle and the turnover gate below set the cost of producing score with such flow.

Return per observation

At checkpoint k (each `checkpoint_book`, and the final one inside `settle_epoch`) with valuation price P_k , the mark of eligible flow is

$$M_k = \left\lfloor \frac{\Delta b_k \cdot P_k}{10^9} \right\rfloor + \Delta q_k$$

where Δb_k and Δq_k are `scoreBaseDelta` (lamports) and `scoreQuoteDelta` (USDC micros) accumulated since the start of the epoch, and the floor rounds toward $-\infty$. M_k revalues all inventory taken from eligible flow at the current reference, so it contains the markup of that inventory as well as the spread at the fill. The return since the previous checkpoint, normalized by the allocation that earned it, is

$$r_k = \text{trunc} \left(\frac{(M_k - M_{k-1}) \cdot 10^9}{a} \right), \quad \rho_k = \rho_{k-1} + r_k$$

where M_{k-1} is `scoreBaselineExcess` (zero at the start of an epoch), r_k is in units of 10^{-9} of allocation, and ρ_k is the carried residual `scoreResidual`. A return is normalized once, at the allocation in force when it was earned; a later allocation never rescales it.

Gates

The residual becomes an observation when

$$\rho_k < 0 \quad \text{or} \quad \left(\text{live} \wedge a > 0 \wedge F_k > 0 \wedge N_k \cdot 10^4 \geq a \cdot \text{minScoredNotionalBps} \right)$$

where F_k is the number of eligible fills and N_k the eligible notional since the last observation. A positive residual waits, carried across checkpoints and settlements, until the book has traded enough external flow to count. A negative residual is observed at once, including while the book is paused or halted.

EWMA with half-life

An observation $y = \rho_k$ at Δt slots after the previous one (`lastScoreSlot`) updates the decayed first and second moments:

$$\lambda(\Delta t) = 2^{-j} \cdot \frac{2h - (\Delta t - jh)}{2h}, \quad j = \left\lfloor \frac{\Delta t}{h} \right\rfloor \quad (\lambda = 0 \text{ for } j \geq 30)$$

$$\mu_k = \lambda \mu_{k-1} + (1 - \lambda) y, \quad m_k = \lambda m_{k-1} + (1 - \lambda) y^2, \quad n_k = n_{k-1} + 1$$

where h is `scoreHalfLifeSlots`, μ is `ewmaMean`, m is `ewmaSquare`, n is `checkpointCount`, and the first observation sets $\mu = y$, $m = y^2$. λ halves at each whole half-life and is interpolated linearly in between. After an observation the residual returns to zero. The design's 7-day half-life is $h = 7 \times 432,000 = 3,024,000$ slots; at the 1,500-slot checkpoint spacing $\lambda = 1 - 1,500/6,048,000$, so each new observation carries a weight of $1/4,032$.

Score from the moments

$$\sigma_k = \max\left(\sqrt{m_k - \mu_k^2}, \sigma_{\min}\right), \quad \varepsilon_k = \begin{cases} \mu_k - \theta & \mu_k > \theta \\ \min(\mu_k, 0) & \text{otherwise} \end{cases}$$

$$S_k = \frac{\varepsilon_k \cdot 10^6}{\sigma_k} \cdot \frac{n_k}{n_k + n_0}$$

where $\sigma_{\min}$ is `scoreStdevFloorE9`, θ is `scoreMinMeanReturnE9`, n_0 is `scoreShrinkageN0`, and S_k is the book's `scoreE6` in units of 10^{-6} . The floor keeps a near-constant return stream from producing an unbounded ratio. The hurdle gives a mean return at or below θ no positive score, while a negative mean scores in full. The factor $n/(n + n_0)$ discounts short histories. When $y < 0$ and the previous score was positive, S_k is capped at the previous score, so a loss can never raise the score through the growth of n .

Score at settlement

The allocator uses a view of the stored history aged to the settlement slot: μ and m are both multiplied by $\lambda(\text{now} - \text{lastScoreSlot})$ and the formula above is applied with n unchanged. A positive view is capped at the stored score. A book that is halted or paused at settlement has score 0 for allocation. The view is written to the quoter's `scoreE6` and drives both the tier rules and the weights below.

Illustration

Take the sample book: mid 200.00 USDC, bids at 1, 2, 4, 8 and 12 bps for 1, 2, 4, 8 and 16 SOL, allocation 375,000 USDC. An external taker sells 3 SOL: 1 SOL fills at 199.98 and 2 SOL at 199.96, for 599.90 USDC. The deltas are $\Delta b = +3$ SOL and $\Delta q = -599.90$ USDC. Marked at a 200.00 reference, $M = 600.00 - 599.90 = 0.10$ USDC, so $r = \text{trunc}(0.10 \times 10^9 / 375,000) = 266$, a return of 2.66×10^{-7} . The fill adds 600 USDC of eligible notional, 16 bps of the allocation, toward the `minScoredNotionalBps` gate. Had the reference fallen to 199.90 before the checkpoint, $M = 599.70 - 599.90 = -0.20$ USDC, and with no positive residual carried in, the negative residual would be observed regardless of the gate. Had the taker been the book's own hot key, none of these scoring values would change.

8.4 From score to allocation

`settle_epoch(E)` computes every book's target from the free NAV left after fees and the withdrawal reservation (section 7.6). Let $\mathcal{A}$ be the books live at settlement, $\mathcal{Z} \subseteq \mathcal{A}$ the tier-0 books among them (after this settlement's tier decision), $\tilde{S}_i$ book i 's score view at settlement, $s_i = \max(\tilde{S}_i, 0)$ and $\Sigma = \sum_{i \in \mathcal{A}} s_i$.

Weight. For $i \in \mathcal{A}$, in basis points:

$$w_i = \left\lfloor \frac{s_i \cdot (10^4 - E_x)}{\Sigma} \right\rfloor + [i \in \mathcal{Z}] \left\lfloor \frac{E_x}{|\mathcal{Z}|} \right\rfloor$$

where E_x is explorationBps (1,000, the 10% exploration pool) and the first term is 0 when $\Sigma = 0$. Weights sum to at most 10^4 . When no tier-0 book is live, the exploration tenth stays idle. w_i is published as targetWeightBps in the settlement event.

Target. With NAV^{free} the free NAV and a_i the allocation of the ending epoch:

$$\delta_i = \left\lfloor \frac{\text{base}_i \cdot C}{10^4} \right\rfloor, \quad \text{cap}_i = (\beta_i - u_i) \cdot L(\tau_i)$$

$$t_i = \min \left(\text{clamp} \left(\left\lfloor \frac{\text{NAV}^{\text{free}} \cdot w_i}{10^4} \right\rfloor, a_i - \delta_i, a_i + \delta_i \right), \text{cap}_i, \left\lfloor \frac{\text{NAV}^{\text{free}} \cdot K}{10^4} \right\rfloor \right)$$

where C is allocationChangeCapBps (2,500, 25% per epoch), K is quoterAllocationCapBps (3,500, 35% of the vault), β_i is the bond after this settlement's charges and fee credits, u_i its pending unbond, τ_i the tier just decided, and the lower clamp bound saturates at zero. allocationChangeBasis selects base_i : with basis 1 it is the prior allocation a_i (or NAV^{free} when $a_i = 0$), with basis 2 it is NAV^{free} . The leverage and concentration caps apply after the change clamp, so they bind even when they require a larger step. Books outside $\mathcal{A}$ get $t_i = 0$.

Scaling. If $\sum_i t_i > \text{NAV}^{\text{free}}$, which the lower change bound can cause after large withdrawals, every target is scaled by $\text{NAV}^{\text{free}} / \sum_i t_i$.

Transfer. Each book receives the basket of section 7.6 for t_i , and its allocation becomes the reference value of what it received. Its starting and tracked inventory are reset to that basket, its loss windows and fill counters to zero, and its epoch to $E + 1$. A book halted by a loss limit is outside $\mathcal{A}$, so it receives no allocation at this settlement; it returns to live and competes again at the next one. A book its owner paused stays paused.

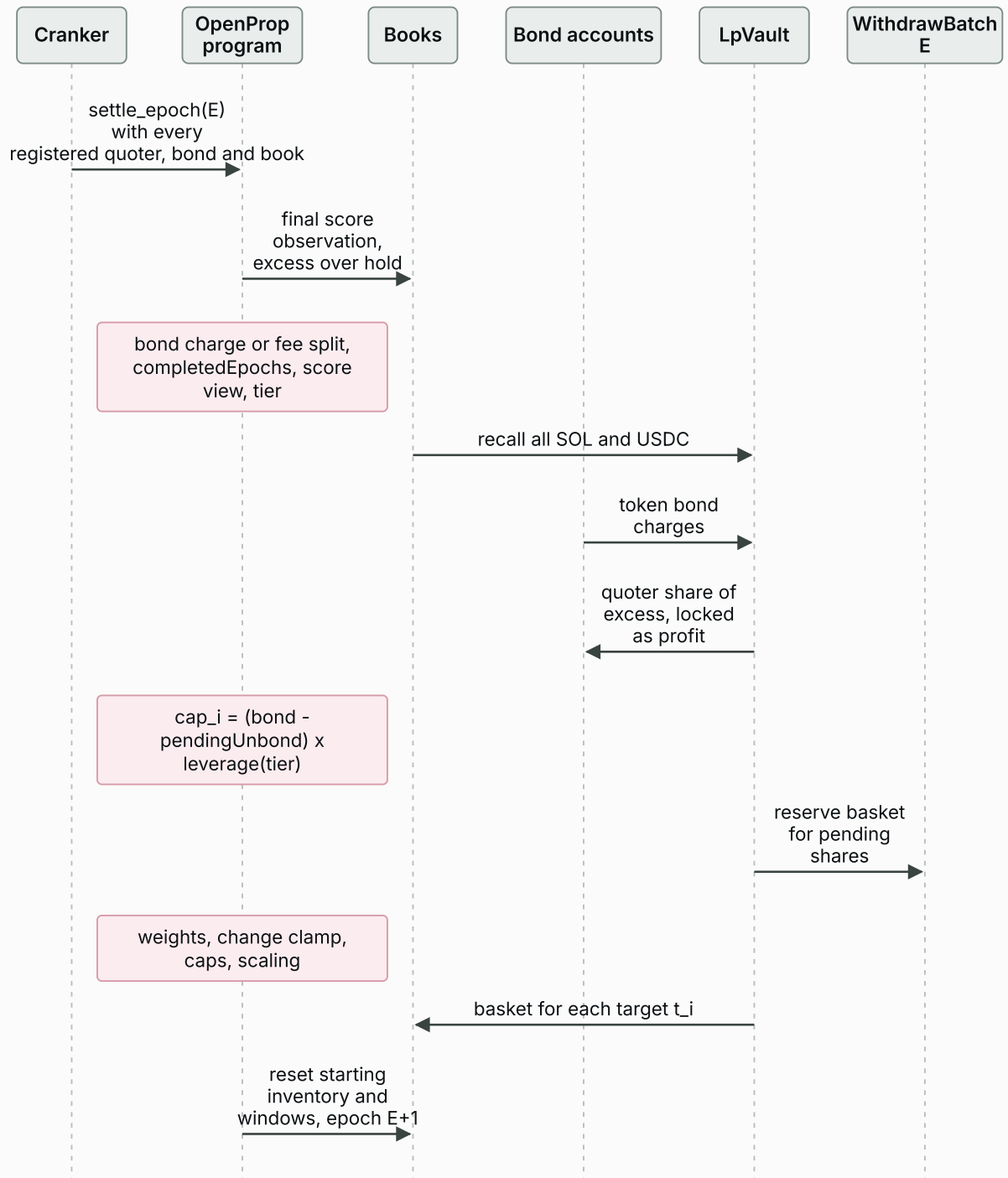


Figure 8.2. One epoch's allocation, in the order the program executes it inside a single `settle_epoch` instruction.

8.5 Worked example: three books

The free basket after fees and reservations is 4,500 SOL and 600,000 USDC at a 200.00 USDC reference, so $\text{NAV}^{\text{free}} = 1,500,000$ USDC. The parameters are the v_0 : $E_x = 1,000$, $C = 2,500$, $K = 3,500$, leverage 1, 5 and 15. Three books are live:

	Book A	Book B	Book C
Tier	2	1	0
Bond minus pending unbond (USDC)	50,000	100,000	30,000
Leverage cap (USDC)	750,000	500,000	30,000
Prior allocation a_i (USDC)	500,000	300,000	30,000
Score view s_i (scoreE6)	300,000	200,000	100,000

Weights. $\Sigma = 600,000$. The score part is $s_i \times 9,000/600,000$: 4,500, 3,000 and 1,500 bps. Book C is the only live tier-0 book and takes the whole exploration pool, so $w = (4,500, 3,000, 2,500)$ bps, summing to 10,000.

Targets. The raw targets are $1,500,000 \times w_i/10^4$; the concentration cap is $1,500,000 \times 0.35 = 525,000$ USDC. Both change bases are shown:

USDC	Book A	Book B	Book C
Raw target	675,000	450,000	375,000
Basis 1 window ($a_i \pm 25\%$ of a_i)	375,000 – 625,000	225,000 – 375,000	22,500 – 37,500
After clamp, basis 1	625,000	375,000	37,500
After leverage cap	625,000	375,000	30,000
After 35% cap: t_i , basis 1	525,000	375,000	30,000
Basis 2 window ($a_i \pm 375,000$)	125,000 – 875,000	0 – 675,000	0 – 405,000
t_i , basis 2	525,000	450,000	30,000

Each book is held by a different bound. Book A, the highest-scoring book, is held by the 35% concentration cap. Book C, at tier 0, is held at $1\times$ its bond, so its allocation is fully covered by its own collateral. Book B is held by the per-epoch change limit under basis 1 and receives its full raw target under basis 2. The targets sum to 930,000 USDC (basis 1) or 1,005,000 USDC (basis 2), below the free NAV, so no scaling applies; 570,000 or 495,000 USDC stays idle.

Baskets (basis 1). Each book receives $t_i/\text{NAV}^{\text{free}}$ of both tokens:

	Share of free basket	SOL	USDC	Value (USDC)
Book A	35%	1,575	210,000	525,000
Book B	25%	1,125	150,000	375,000
Book C	2%	90	12,000	30,000
Idle in vault	38%	1,710	228,000	570,000

The bond backing each new allocation is $\kappa_A = 35,000$, $\kappa_B = 75,000$ and $\kappa_C = 30,000$ USDC, inside each quoter's bond. These amounts are what `request_unbond` and `claim_unbond` hold back during epoch $E + 1$.

9. Checkpoints, settlement and fees

OpenProp measures every book against its own starting inventory, marked to the pair's reference price. Two permissionless instructions do the measuring. `checkpoint_book` marks one book during the epoch: it records the book's excess over hold, enforces the hourly and epoch loss windows and takes a score sample. `settle_epoch` closes the epoch for every registered book in one instruction: it charges losses to bonds, splits gains above each book's high-water mark, pays the quoter and protocol shares, reserves queued withdrawals and moves the next epoch's allocations into the books. Both instructions read the reference that `post_ref_price` stored in `RefPrice`. Neither accepts a price from its caller.

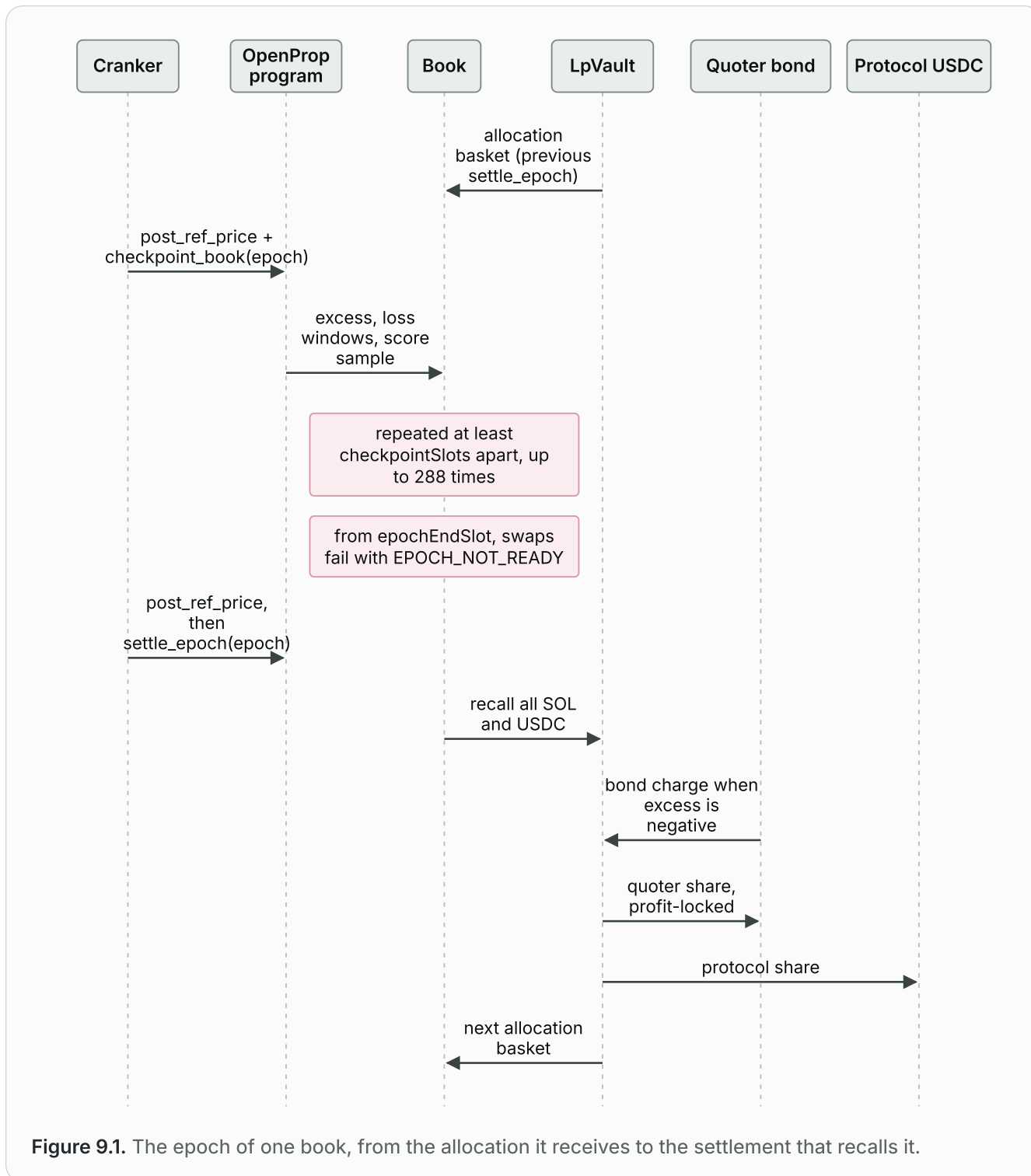
9.1 Epoch and checkpoint clocks

Every clock in this chapter counts `Clock.slot`. No instruction reads `Clock.unix_timestamp`.

Clock	Config field	Value	At 200 ms slots
Epoch length	<code>epochSlots</code>	432,000 slots	24 h
Minimum spacing between two checkpoints of a book	<code>checkpointSlots</code>	about 1,500 slots	5 min
Checkpoints per book per epoch	<code>epochSlots / checkpointSlots</code>	up to 288	—
Hourly loss window	<code>hourlyWindowSlots</code>	set at initialization	—
Score half-life	<code>scoreHalfLifeSlots</code>	7 days, expressed in slots at initialization	—
Withdrawal queue bound	<code>withdrawEpochs</code>	at most 2 epochs	—

`initialize_pair` writes the current slot s_0 into `LpVault.epochStartSlot`. Epoch e covers the slots $[s_0 + eE, s_0 + (e + 1)E)$, where E is `epochSlots`. Each settlement advances `epochStartSlot` by exactly E , not to the slot in which settlement ran, so the grid does not drift when a settlement lands late. If more than one boundary has passed, consecutive `settle_epoch` calls settle the missed epochs one at a time. Each Book stores the end of its epoch in `epochEndSlot`. From that slot until settlement, `swap_exact_in` and `deposit` fail with `EPOCH_NOT_READY` (6026). `request_withdraw` is still accepted and joins the epoch that is about to settle.

`checkpointSlots` is a minimum spacing, not a schedule. A `checkpoint_book` call that arrives earlier than `checkpointSlots` after the book's `lastCheckpointSlot` returns without effect. The protocol cranker plans a checkpoint for every allocated book once that spacing has elapsed, so a book receives up to 288 checkpoints per epoch.



9.2 checkpoint_book

checkpoint_book (tag 3) takes one argument, the book's expected epoch, and six accounts: Config, RefPrice (writable), Book (writable), the book's SOL and USDC token accounts, and the book's Quoter (writable). It requires no signer. The cranker sends it in one transaction after three Ed25519 verification instructions and a post_ref_price, so the book is marked against a reference written in the same transaction.

1. **Binding.** The book's epoch must equal the argument, its pair must match `RefPrice.pair`, and its quoter must be the supplied `Quoter`. A mismatch fails with `INVALID_ACCOUNT` (6001).
2. **Stale reference.** If the reference's source slot is older than `refStalenessSlots` (10 slots), the instruction clears any pending breaker recovery and returns without marking the book. If the pair breaker was not already tripped, it trips it with reason 2 and emits an incident record.
3. **Spacing.** Within `checkpointsSlots` of the last checkpoint, the instruction returns without effect.
4. **Custody.** The token accounts must be the book's own, and their balances must cover the book's tracked inventory.
5. **Valuation.** The reference must be fresh, its confidence within `confidenceBps` and its two sources within `divergenceBps` of the mid. A halt does not stop marking: a pair halted by its price-move breaker or by the guardian is still valued while its sources are fresh and agree.
6. **Excess, loss windows and score sample**, as defined below.
7. **Record.** A checkpoint record (kind 1) carries the excess, book value, hold value, `scoreE6`, reference price, hourly loss and epoch loss.

Excess over hold

$$V(b, q; P) = \left\lfloor \frac{bP}{10^9} \right\rfloor + q \quad X = V(B, Q; P) - V(B_0, Q_0; P)$$

Symbol	Meaning
P	Reference mid <code>RefPrice.mid</code> , in USDC micros (6 decimals) per whole SOL
B, Q	Tracked inventory, <code>Book.trackedBase</code> in lamports and <code>Book.trackedQuote</code> in USDC micros
B_0, Q_0	Starting inventory, <code>Book.startBase</code> and <code>Book.startQuote</code> , written by the previous settlement
V	Value of a SOL/USDC basket at P , in USDC micros
X	Excess over hold, signed, in USDC micros

X is the value of what the book holds now minus the value of what it was given, both at the current price. Equivalently, $X = \lfloor (B - B_0)P/10^9 \rfloor + (Q - Q_0)$ up to rounding, so a change in the reference moves X only through the SOL the book has taken on or given up since settlement. The starting basket is valued identically on both sides. A quoter therefore gains or loses excess through spread capture and inventory management, not through the direction of SOL. Only swaps and settlement change the tracked inventory. Tokens sent to a book's accounts from outside raise its token balance but not B or Q , so they never count as excess.

Loss windows

$$L_E = \max(0, -X) \quad L_H = \max(0, X_h - X)$$

X_h is `hourlyStartExcess`, the excess at the start of the current hourly window. The window restarts, with $X_h = X$, when `hourlyWindowSlots` have passed since `hourlyStartSlot`. These are the windows that `swap_exact_in` tests on every fill (Chapter 5). With A the book's allocation, the checkpoint halts the book (state 1) when

$$L_E \cdot 10^4 > A \cdot \ell_E \text{ (reason 7)} \quad \text{or else} \quad L_H \cdot 10^4 > A \cdot \ell_H \text{ (reason 6)}$$

where ℓ_E is epochLossBps (200, i.e. 2% of allocation) and ℓ_H is hourlyLossBps (50, i.e. 0.5%). A halt sets bit 0 of the quoter's limitTripBitmap, and an epoch-loss halt also sets bit 0 of epochLossTripBitmap. A risk halt overrides a cold-owner pause. swap_exact_in runs the same loss computation on the projected post-fill inventory, so a fill that would cross either limit never executes. The checkpoint catches losses that come from the reference moving while the book holds inventory.

Score samples

Each checkpoint also produces the book's score sample; Chapter 8 gives the score formulas. Fills by takers other than the book's cold owner and its current hot authorities accumulate signed inventory changes in scoreBaseDelta and scoreQuoteDelta. The checkpoint marks those changes at P to a value m , normalizes the change since the previous mark m_0 to a return per unit of allocation, and adds it to the carried residual R (scoreResidual):

$$r = \frac{(m - m_0) \cdot 10^9}{A} \quad R \leftarrow R + r$$

The residual becomes an observation when $R < 0$, or when the book is live and allocated, has new scored fills, and its new scored turnover reaches minScoredNotionalBps of A . An observation updates the decayed moments ewmaMean and ewmaSquare, increments checkpointCount, recomputes scoreE6 and resets R to zero. A residual that has not become an observation survives pauses, halts and settlement, so a loss cannot be dropped by stopping the book.

9.3 settle_epoch

settle_epoch (tag 4) takes the epoch number and $10 + 5n$ accounts: the caller (signer), Config, RefPrice, LpVault, the vault's SOL and USDC token accounts, the protocol USDC account, the epoch's WithdrawBatch, the token and system programs, then (Quoter, bond, Book, book SOL, book USDC) for each of the n registered books in registry order. The epoch must equal LpVault.epoch, the slot must have reached epochStartSlot + epochSlots, and the reference must pass the valuation gates of 9.2. A call for an epoch at or below lastSettledEpoch returns without effect. Every account, link and token balance is validated for all books before the first transfer.

Excess for the epoch and the high-water mark

For each book the program computes X at the settlement reference and updates its cumulative excess C (cumulativeExcess) and high-water mark H (highWaterMark). With $C = C_{\text{prev}} + X$:

$$X \geq 0 : \quad E^+ = \min(X, \max(0, C - H)) \quad C' = C \quad H' = \max(H, C)$$

$$X < 0, L = -X : \quad c = \min(L, b + d) \quad \ell = L - c \quad C' = C + c \quad H' = \max(H, C')$$

Symbol	Meaning
E^+	Positive excess above the mark, the amount that is split
b	The quoter's tracked USDC bond, Quoter.bond

Symbol	Meaning
d	Unpaid quoter credit from earlier epochs, <code>Book.deferredQuoterFees</code>
c	First-loss coverage charged for the epoch
ℓ	Loss that falls on LPs

Coverage restores cumulative excess. A loss fully covered by first-loss capital leaves C where it was, so $C = H$ for a book whose losses have always been covered. The two differ only after LPs have absorbed a loss ℓ . The book must then earn ℓ back for the LPs before any of its excess is split again. Suppose a loss left $C = H - 20.00$ USDC and the next epoch earns $X = 58.80$: $E^+ = 38.80$ is split, and the first 20.00 stays entirely with the LPs.

The split

$$S_{LP} = \left\lfloor \frac{E^+ \lambda_{LP}}{10^4} \right\rfloor \quad S_Q = \left\lfloor \frac{E^+ \lambda_Q}{10^4} \right\rfloor \quad S_P = E^+ - S_{LP} - S_Q$$

λ_{LP} is `lpShareBps` (5,000) and λ_Q is `quoterShareBps` (4,000). The protocol takes the remainder, 1,000 bps plus rounding dust. S_{LP} never moves: the book's whole inventory returns to the vault at settlement, and the LP share is the part of it that is not paid out.

Losses: bond first, then unpaid credit, then LPs

The coverage c comes first from tokens in the bond account, then from cancelling unpaid quoter credit:

$$c_t = \min(b, c) \quad c_d = c - c_t \quad b' = b - c_t \quad d' = d - c_d + S_Q$$

The program transfers c_t from the bond account (authority: the Quoter PDA) to the vault's USDC account. Cancelling c_d moves no tokens. Only the remainder ℓ reduces LP value. The quoter's new share S_Q joins d' and the protocol's share joins `deferredProtocolFees`; both are paid in the next step.

Fee payment, deferred fees and the profit lock

Before paying anything the program recalls every book: it transfers the full balances of each book's SOL and USDC accounts to the vault, including tokens that were never part of tracked inventory. Fees are then paid from unreserved vault USDC, $\$U = Q_{\text{vault}} - \$\text{reservedQuote}$, book by book in registry order, quoter credit before protocol credit:

$$\pi_Q = \min(U, d') \quad U \leftarrow U - \pi_Q \quad \pi_P = \min(U, p') \quad U \leftarrow U - \pi_P$$

p' is the book's `deferredProtocolFees` including S_P . π_Q goes to the quoter's bond account and is added to `Quoter.bond`, so a quoter's gains become first-loss capacity. π_P goes to the protocol USDC account. What cannot be paid stays in `Book.deferredQuoterFees` and `Book.deferredProtocolFees`, totalled in `LpVault.deferredQuoterFees` and `LpVault.deferredProtocolFees`. Reserved withdrawal assets never fund fees. While a liability D is outstanding, the vault holds back SOL worth that amount from the next allocation and from withdrawal reservations:

$$h_B = \min\left(B_{\text{free}}, \left\lceil \frac{D \cdot 10^9}{P} \right\rceil\right)$$

Each paid quoter amount opens a profit tranche in slot $e \bmod 64$ of `Quoter.profitTranches`, with unlock epoch $e + \kappa$, where κ is `profitLockEpochs` (set at initialization, at most 64). Tranches still locked in the next epoch are summed into `lockedProfit`. A tranche whose unlock epoch has not been reached is excluded from the bond that `request_unbond` and `claim_unbond` may release. Paid amounts count toward allocation capacity at once: the book's cap for the next epoch is $(b - u) \Lambda_t$, where u is `pendingUnbond` and Λ_t is the leverage of the quoter's tier (1, 5 or 15). Unpaid credit d counts as first-loss coverage but adds neither capacity nor withdrawable bond. Excess bond becomes withdrawable after 2 epochs.

Withdrawal reservation

With B_{free} and Q_{free} the vault's balances net of existing reservations and of h_B , the shares queued in the epoch, s_p (`pendingShares`), receive their proportional basket:

$$R_B = \left\lfloor \frac{B_{\text{free}} s_p}{s_a + s_p + s_v} \right\rfloor \quad R_Q = \left\lfloor \frac{Q_{\text{free}} s_p}{s_a + s_p + s_v} \right\rfloor$$

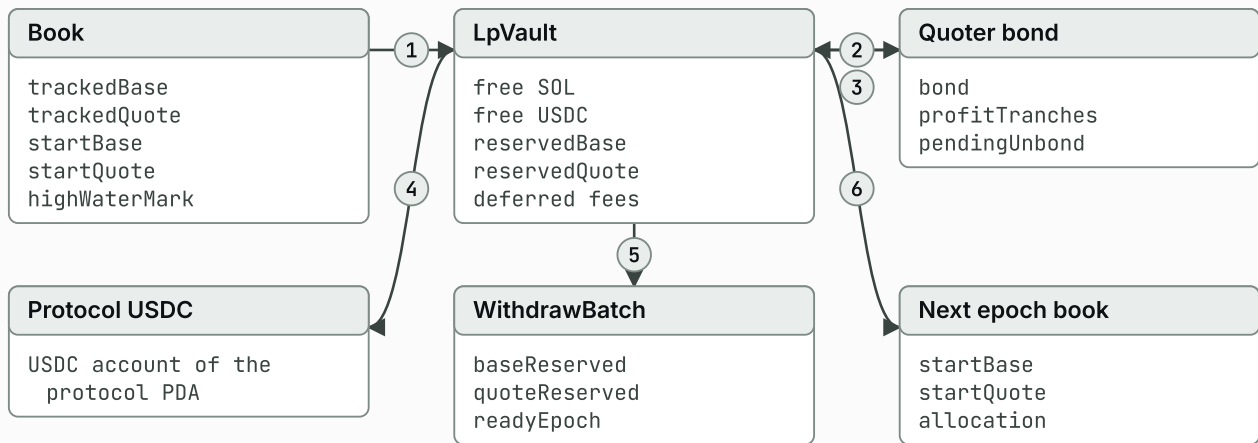
s_a is the active share supply and s_v is `virtualShares`. The amounts are written into the `WithdrawBatch`, added to `reservedBase` and `reservedQuote`, and become claimable from epoch $e + 1$ (Chapter 7). Including s_v in the denominator leaves the virtual-share fraction in the vault.

New allocations

What remains, with value $N = V(B_{\text{free}} - R_B, Q_{\text{free}} - R_Q; P)$, is allocated by the rule of Chapter 8. In order:

1. **Weight.** Each live book gets a weight w_i in bps: its share of the positive scores times $10^4 - \text{explorationBps}$, plus an equal part of the 1,000-bps exploration pool if it is at tier 0. A book paused or halted at settlement gets weight and target zero.
2. **Raw target.** $\lfloor N w_i / 10^4 \rfloor$.
3. **Change limit.** The target is clamped to $a_i \pm \Delta_i$, where a_i is the previous allocation and Δ_i is `allocationChangeCapBps` (2,500) of either a_i (`allocationChangeBasis` 1) or N (basis 2).
4. **Caps.** The target is cut to the bond capacity $(b_i - u_i) \Lambda_{t_i}$ and to `quoterAllocationCapBps` (3,500) of N . These caps apply after the change limit.
5. **Scaling.** If the targets sum to more than N , all are scaled down pro rata.
6. **Transfer.** Each book receives $\lfloor B' a_i^* / N \rfloor$ lamports and $\lfloor Q' a_i^* / N \rfloor$ USDC micros, where B' and Q' are the vault balances left after the reservation, so every book starts the epoch with the vault's own SOL/USDC ratio.

The book's `startBase`, `startQuote`, `trackedBase` and `trackedQuote` are set to the basket received, and its allocation to that basket's value. Loss windows and fill counters reset. A risk-halted book returns to the live state with the zero allocation it was given; a paused book stays paused. Settlement also applies the tier rules of Chapter 8: an epoch counts toward `completedEpochs` only if the book was live, allocated and had at least one scored fill.



- ① Book → LpVault: entire SOL and USDC balances
- ② Quoter bond → LpVault: bond charge c_t
- ③ LpVault → Quoter bond: paid quoter share
- ④ LpVault → Protocol USDC: paid protocol share
- ⑤ LpVault → WithdrawBatch: pro-rata reservation
- ⑥ LpVault → Next epoch book: proportional basket

Figure 9.2. Value moved by one `settle_epoch`. Every arrow is an SPL Token transfer signed by a program-derived authority.

9.4 A worked epoch

The reference was 200.00 USDC at the start of the epoch and is 202.00 at settlement. The vault held 50 SOL and 10,000.00 USDC idle, and no withdrawals are queued. Book A (tier 1) and book B (tier 0) end the epoch with the inventories below, net of all their fills.

	Book A	Book B
Bond	12,000.00	20,000.00
$C_{\text{prev}} = H$	400.00	0.00
Start basket (B_0, Q_0)	100 SOL, 20,000.00	50 SOL, 10,000.00
Allocation a	40,000.00	20,000.00
End inventory (B, Q)	98 SOL, 20,462.80	56 SOL, 8,500.00
Book value at 202.00	40,258.80	19,812.00
Hold value at 202.00	40,200.00	20,100.00
Excess X	+58.80	-288.00

B's loss is 1.44% of its allocation, under the 2% epoch limit, and the example assumes no hourly window lost more than 0.5%. Both books stayed live.

Settlement step	Book A	Book B
C, E^+	458.80, 58.80	—
$S_{LP} / S_Q / S_P$	29.40 / 23.52 / 5.88	—
Coverage c , LP loss ℓ	—	288.00, 0.00
C', H'	458.80, 458.80	0.00, 0.00
Bond after settlement	12,023.52 (23.52 in a profit tranche)	19,712.00
Capacity $(b - u)\Lambda_t$	60,117.60	19,712.00

After the recall, the bond charge and the two fee payments, the vault holds 204 SOL and 39,221.40 USDC, so $N = 204 \times 202 + 39,221.40 = 80,429.40$. That equals the hold value of everything the LPs owned (80,400.00) plus S_{LP} . Book B's loss reached them only through B's bond.

Only A has a positive score, so $w_A = 9,000$ bps. B is the only tier-0 book and takes the whole exploration pool, $w_B = 1,000$ bps. The 35% cap is 28,150.29.

Allocation step	Book A	Book B, basis 1	Book B, basis 2
Raw target $Nw_i/10^4$	72,386.46	8,042.94	8,042.94
Change limit Δ_i	10,000.00 (basis 1) / 20,107.35 (basis 2)	5,000.00	20,107.35
After change limits	50,000.00 / 60,107.35	15,000.00	8,042.94
After capacity and 35% caps	28,150.29	15,000.00	8,042.94
Basket transferred	71.4 SOL, 13,727.49 USDC	38.045789226 SOL, 7,314.750576 USDC	20.4 SOL, 3,922.14 USDC

The 35% cap binds for A under either basis. For B the result depends on `allocationChangeBasis`, which is set at initialization: basis 1 limits B's fall to 25% of its previous allocation, basis 2 to 25% of vault NAV. Under basis 1, B's recorded allocation is 14,999.999999 USDC, the value of the basket after rounding down to token atoms. The remainder of N stays idle in the vault.

9.5 The protocol's fee share

The protocol's income is S_P , 10% of positive excess above each book's high-water mark. It accrues per book in `Book.deferredProtocolFees` and is paid in USDC to one account: the associated USDC account of the program's own PDA ["protocol"], which `Config.treasuryToken` names. Settlement checks that the destination is exactly that address, a classic SPL Token account of the configured USDC mint with no delegate and no close authority. Identity rotation cannot change `treasuryToken`, so no key can redirect the share. No other instruction transfers to this account, and no admin instruction transfers from the vault. The program pays nothing to the caller of `checkpoint_book` or `settle_epoch`; the cranker pays its own transaction fees and account rent, and none of the protocol share goes to it.

Buy and burn

The protocol USDC has one way out, and it ends in a burn. \$OPENPROP is the protocol's token, with a fixed supply of 1,000,000,000, all of it sold in one public launch on equal terms: no presale, no team, investor or airdrop allocation, and no vesting. Nothing is minted later, so the supply only falls.

Three instructions move the USDC, and all three take the same 13 accounts in this order: the configured cranker (signer), Config, the Protocol account at ["protocol"], the protocol USDC account, the cranker's USDC account, the buyback mint, the cranker's account for that mint, the Token program, the mint's token program, the Instructions sysvar, the USDC mint, the Associated Token program and the System program. The mint and its token program are Config.buybackMint and Config.buybackTokenProgram, set at initialization; the token program is classic Token or Token-2022. One transaction carries the whole buyback:

Step	Instruction	What the program does
1	Compute budget	—
2	buyback_begin(amount, min_burn) (tag 23)	Reads the transaction through the Instructions sysvar and requires one buyback_burn and a buyback_finish of this program, with the same accounts, later in it; creates the cranker's two token accounts if missing; records both of the cranker's balances, the protocol balance and the mint's supply in Protocol; transfers amount USDC to the cranker under the PDA's signature
3	The Jupiter route	Setup, swap and cleanup instructions: USDC in, \$OPENPROP out
4	buyback_burn() (tag 24)	Burns the rise in the cranker's \$OPENPROP balance over its record: every token the route returned, positive slippage included
5	buyback_finish() (tag 25)	Must be the last instruction. Requires the mint's supply to have fallen by at least min_burn, neither of the cranker's balances to exceed its record and the protocol balance to have fallen by exactly amount; clears the guard and emits a buyback event (kind 5): USDC spent, tokens burned, slot

If any check fails, the transaction fails, and the claim of step 2 with it. buyback_begin runs only as a top-level instruction, and while a buyback is open the guard flags in Config and Protocol refuse a second one. No admin, guardian or treasury key can debit the protocol USDC account. While buybackMint is zero, buybacks are disabled and the USDC accrues; the mint, its token program and the cranker change only through the timelocked identity rotation of Chapter 10. The cranker sends the transaction once the protocol balance reaches its configured minimum, with min_burn set to the Jupiter quote's minimum output (13.6).

Because the protocol's share is taken from excess above the high-water mark, the burn follows the quality of the quoting, not the volume of fills. A fill earns the protocol nothing until its book's cumulative excess is above its mark at a settlement; a book that loses and then recovers pays nothing on the recovery. With the split above, every 1,000 USDC of new excess sends 100 USDC to buying and burning \$OPENPROP.

Unpaid fees

Unpaid fees are explicit liabilities. deposit prices new shares against economic NAV: gross free assets plus pending bond coverage, minus newly owed quoter and protocol shares, minus deferred fees. If liabilities exceed assets, LP NAV is zero, the shortfall stays recorded, settlement continues, and deposits fail with FEE_LIABILITY_SHORTFALL (6037). Each settlement records the outcome for every book in a settlement record (kind 2, 242 bytes): excess, LP share, quoter share, protocol share, bond charge, LP loss, new allocation, old allocation, high-water mark, reference price, target weight and cancelled quoter credit.

10. Parameters and governance

Every risk limit, split, clock and protocol identity of a deployment lives in one account, `Config`, at the program-derived address `["config"]`. The program compiles in no value that a deployment is expected to choose. A required value that is missing or out of range fails with `UNDECIDED_PARAMETER` (6003) rather than falling back to a default. Changes go through a small set of instructions signed by the admin, which the design specifies as a Squads multisig behind a timelock. The program bounds those instructions in three ways: validation rules that every candidate configuration must pass, an envelope of floors and ceilings fixed at initialization, and timelocks that make any loosening wait longer than the LP withdrawal queue. Parameters are not put to a vote: neither `$OPENPROP`, whose one role is the buy-and-burn of Chapter 9, nor LP shares (OPLP) carry governance rights.

10.1 The Config account

`Config` is 1,728 bytes, account kind 0, version 6. Fields are encoded one by one at the offsets published in `abi.json`, little-endian, never by casting Rust memory.

Offset	Bytes	Content
0	8	Header: version, kind, bump, flags, reason
8	296	admin, guardian, treasuryToken, baseMint, quoteMint, pythSigner, switchboardSigner, switchboardFeed, pythFeedId, retailSigner
304	368	46 active parameters, one u64 each
672	8	pendingActivationSlot
680	368	pendingParams, the proposed copy of the 46 parameters
1048	64	pythSlotAttestor, guardianHaltSigner
1112	8	identityActivationSlot
1120	296	pendingIdentities, the first nine keys and pythFeedId of a proposed identity vector
1416	24	scoreStdevFloorE9, scoreMinMeanReturnE9, minScoredNotionalBps
1440	24	pendingScoreParams
1464	64	The immutable envelope, eight u64 values
1528	8	pendingActivationEpoch
1536	96	cranker, buybackMint, buybackTokenProgram
1632	96	pendingBuybackIdentities, the proposed values of those three

USDC amounts and prices use 6 decimals, SOL amounts are lamports (9 decimals) and share amounts use 6 decimals. Basis-point fields are out of 10,000. A `Config` created by a build with the `local-dev` feature carries flag bit 0, which marks it as development-only; bit 1 is set only inside an open buyback transaction (9.5).

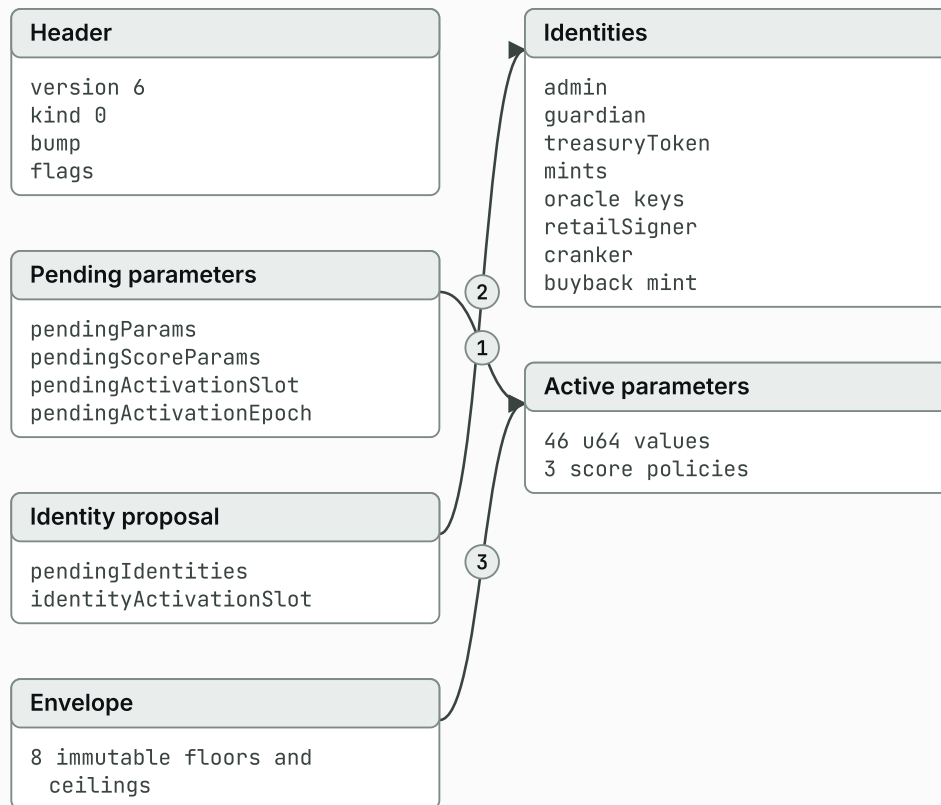


Figure 10.1. Regions of Config and the instructions that move values between them.

10.2 Parameters by purpose

The 49 parameters below are the full proposal vector of set_params, grouped by what they control. Values are the v0 set. "Set at initialization" marks a value v0 leaves open, which each deployment supplies. The last column gives the rule set_params applies to a change (10.4).

Parameter	Unit	Controls	Value	Change
Clocks				
epochSlots	slots	Epoch length and settlement boundaries	432,000	fixed
checkpointSlots	slots	Minimum spacing between checkpoints of a book	about 1,500	fixed
hourlyWindowSlots	slots	Length of the hourly loss window	set at initialization	fixed

Parameter	Unit	Controls	Value	Change
scoreHalfLifeSlots	slots	Half-life of the score's decayed moments	7 days, in slots	fixed
unbondEpochs	epochs	Delay from request_unbond to claim_unbond	2	fixed
withdrawEpochs	epochs	Withdrawal queue bound used by the loosening deadline	2	fixed
timeLockEpochs	epochs	Delay of a loosening and of an identity rotation	3	fixed
Book risk limits				
quoteStalenessSlots	slots	Maximum quote age at a fill	2	raise loosens
skewBps	bps of book value	Maximum gap between SOL value and USDC after a fill	set at initialization	raise loosens
hourlyLossBps	bps of allocation	Hourly loss limit	50	raise loosens
epochLossBps	bps of allocation	Epoch loss limit	200	raise loosens
maxLevelBaseAtoms	lamports	Maximum size of one ladder level	set at initialization	raise loosens
retailDeltaLimitBps	bps	Maximum offset tightening for the retail flow class	set at initialization	raise loosens
Reference gates				
bandBps	bps of reference	Maximum distance between quote mid and reference	set at initialization	raise loosens
confidenceBps	bps of reference	Maximum aggregate confidence of the reference	set at initialization	raise loosens
divergenceBps	bps of reference	Maximum gap between the Pyth and Switchboard prices	set at initialization	raise loosens
refStalenessSlots	slots	Maximum age of the reference source slot	10	raise loosens
oracleSlotAttestationPolicy	policy	Pyth payloads require a slot attestation	1, the only accepted value	pinned
oracleTimestampToleranceUs	reserved	No function; freshness is measured in slots	0, the only accepted value	pinned
Pair breaker				
breakerMoveBps	bps	Reference move inside the window that halts the pair	300	raise loosens

Parameter	Unit	Controls	Value	Change
breakerWindowSlots	slots	Rolling window of the move breaker	150	lower loosens
resumeDelaySlots	slots	Cooldown after a halt condition clears	50	lower loosens
Scoring				
scoreShrinkageN0	samples	n_0 in the shrinkage factor $n/(n + n_0)$	set at initialization	any change waits
scoreStddevFloorE9	return $\times 10^9$	Floor of the score's standard deviation	set at initialization	lower loosens
scoreMinMeanReturnE9	return $\times 10^9$	Mean-return hurdle subtracted before scoring	set at initialization	any change waits
minScoredNotionalBps	bps of allocation	Scored turnover needed for a positive sample	set at initialization	any change waits
tier2ScoreE6	score $\times 10^6$	Score needed for tier 2	set at initialization	lower loosens
Tiers				
tier1Epochs	epochs	Completed epochs needed for tier 1	14	lower loosens
tier2Epochs	epochs	Completed epochs needed for tier 2	45	lower loosens
tier0Leverage	$\times$ bond	Allocation capacity at tier 0	1	raise loosens
tier1Leverage	$\times$ bond	Allocation capacity at tier 1	5	raise loosens
tier2Leverage	$\times$ bond	Allocation capacity at tier 2	15	raise loosens
downgradeTrips	trips	Epoch-loss trips in the window that drop a tier	2	raise loosens
downgradeWindowEpochs	epochs	Trip window, also tier 1's no-trip window	7	lower loosens
Allocation				
explorationBps	bps of NAV	Exploration pool split across tier-0 books	1,000	raise loosens
allocationChangeCapBps	bps	Maximum change of a book's allocation per epoch	2,500	raise loosens

Parameter	Unit	Controls	Value	Change
allocationChangeBasis	1 or 2	Base of that change: previous allocation (1) or vault NAV (2)	set at initialization	any change waits
quoterAllocationCapBps	bps of NAV	Maximum allocation per quoter	3,500	raise loosens
Fees and splits				
lpShareBps	bps of positive excess	LP share above the high-water mark	5,000	any change waits
quoterShareBps	bps of positive excess	Quoter share; the protocol takes the remainder	4,000	any change waits
profitLockEpochs	epochs	Lock on quoter shares credited to the bond	set at initialization	any change waits
Capacity				
vaultCapQuoteAtoms	USDC micros	Maximum LP NAV after a deposit	2,000,000,000,000 (\$2M)	raise loosens
maxBooks	books	Maximum registered books	set at initialization	raise loosens
minBondQuoteAtoms	USDC micros	Minimum bond at registration	set at initialization	lower loosens
Share issuance				
initialSharePrice	USDC micros per whole share	Conversion price of the first deposit	set at initialization	any change waits
virtualShares	share atoms	Virtual shares added to every conversion	set at initialization	any change waits
virtualQuoteAtoms	USDC micros	Virtual assets paired with the virtual shares	set at initialization	any change waits
Policies				
settlementFallbackPolicy	policy	Settlement without a valid reference	0 (disabled), the only accepted value	pinned
evictionPolicy	policy	Forced removal of an inactive quoter	0 (disabled), the only accepted value	pinned

Validation rules

The program runs one check, `config_ok`, on every candidate configuration: at initialization, on a proposal, on its activation and on an identity rotation. A candidate passes only if all of the following hold.

- **Positive values.** `epochSlots`, `checkpointSlots`, `hourlyWindowSlots`, `scoreHalfLifeSlots`, `scoreShrinkageN0`, `maxLevelBaseAtoms`, `minBondQuoteAtoms`, `vaultCapQuoteAtoms`, the three leverages, `initialSharePrice`, `virtualShares`, `virtualQuoteAtoms` and the three score policies are greater than zero.
- **Basis points.** The band, confidence, divergence, skew, retail, loss, split, exploration and allocation-cap fields are at most 10,000, and $\text{lpShareBps} + \text{quoterShareBps} \leq 10,000$, so the protocol remainder is never negative.
- **Ranges.** $1 \leq \text{maxBooks} \leq 5$, the storage capacity of the book registry. $1 \leq \text{breakerWindowSlots} \leq 150$, the number of price buckets in `RefPrice`. $1 \leq \text{downgradeTrips} \leq \text{downgradeWindowEpochs} \leq 64$ and $1 \leq \text{profitLockEpochs} \leq 64$, the width of the trip bitmaps and tranche ring. `allocationChangeBasis` is 1 or 2, and `refStalenessSlots` ≥ 1 .
- **Clock order.** $\text{unbondEpochs} \geq 2$, $\text{withdrawEpochs} \geq 1$, $\text{timelockEpochs} \geq 3$, $\text{timelockEpochs} > \text{withdrawEpochs}$, $\text{tier1Epochs} \geq 1$ and $\text{tier2Epochs} \geq \text{tier1Epochs}$.
- **Share issuance.** $\text{virtualQuoteAtoms} \cdot 10^6 = \text{initialSharePrice} \cdot \text{virtualShares}$.
- **Pinned policies.** `oracleSlotAttestationPolicy` = 1, `settlementFallbackPolicy` = 0, `evictionPolicy` = 0 and `oracleTimestampToleranceUs` = 0.
- **Envelope.** The relations in 10.3.
- **Identities.** `admin`, `guardian`, both mints, `pythSigner`, `switchboardSigner`, `switchboardFeed`, `pythSlotAttestor` and `treasuryToken` are non-zero. `pythFeedId` fits in 32 bits. A non-zero `guardianHaltSigner` differs from `admin`, `guardian`, `pythSigner`, `switchboardSigner`, `retailSigner` and `pythSlotAttestor`.

10.3 The immutable envelope

Eight values are written once by `initialize_config` and lie outside every region that `set_params` and `rotate_identities` write. They bound all later values.

Envelope value	Bound
<code>minimumScoreStdevFloorE9</code>	<code>scoreStdevFloorE9</code> $\geq$ this value
<code>minimumScoreMeanReturnE9</code>	<code>scoreMinMeanReturnE9</code> $\geq$ this value
<code>minimumScoredNotionalBps</code>	<code>minScoredNotionalBps</code> $\geq$ this value
<code>minimumConfidenceBps</code>	<code>confidenceBps</code> $\geq$ this value
<code>minimumDivergenceBps</code>	<code>divergenceBps</code> $\geq$ this value
<code>minimumRefStalenessSlots</code>	<code>refStalenessSlots</code> $\geq$ this value
<code>maximumResumeDelaySlots</code>	<code>resumeDelaySlots</code> $\leq$ this value, itself at most $2^{63} - 1$
<code>minimumBreakerMoveBps</code>	<code>breakerMoveBps</code> $\geq$ this value

All eight are set at initialization and must be positive. The oracle bounds keep every gate usable. A tightening takes effect at once (10.4), and checkpoints, settlement and withdrawal reservations all depend on a reference that passes the confidence, divergence and freshness gates. The floors keep an immediate change from setting a gate so narrow that no reference passes, a breaker that trips on every update, or a cooldown that never ends. The scoring floors keep the score's denominator and hurdles away from zero, so near-constant returns or dust trading cannot produce a large score.

10.4 Proposing and activating a change

set_params (tag 14) is signed by admin and takes Config (writable) and LpVault. Its first byte selects the action.

Propose (0). The body carries all 49 values, 393 bytes in total. The program builds a candidate Config from the current one and the new vector and runs config_ok on it. It rejects any change to the seven clocks with UNDECIDED_PARAMETER, because epoch geometry and the queues are fixed for a deployed pair. It then classifies the change:

- a **loosening** if a "raise loosens" value rises, a "lower loosens" value falls, or an "any change waits" value differs;
- otherwise a **tightening**, which is written to the active parameters in the same instruction and clears any pending proposal.

A vector that loosens any one value waits as a whole, including the values it tightens. A loosening is stored in pendingParams and pendingScoreParams with two deadlines, both fixed when the proposal is made:

$$s_{\text{act}} = s_{\text{prop}} + T \cdot E \qquad e_{\text{act}} = e_{\text{prop}} + \max(W, W') + 1$$

Symbol	Meaning
$s_{\text{prop}}, e_{\text{prop}}$	Slot of the proposal and LpVault.epoch at that slot
T	timeLockEpochs (3)
E	epochSlots (432,000)
W, W'	Current and proposed withdrawEpochs (2; the field is fixed, so $W' = W$)
$s_{\text{act}}, e_{\text{act}}$	pendingActivationSlot, pendingActivationEpoch

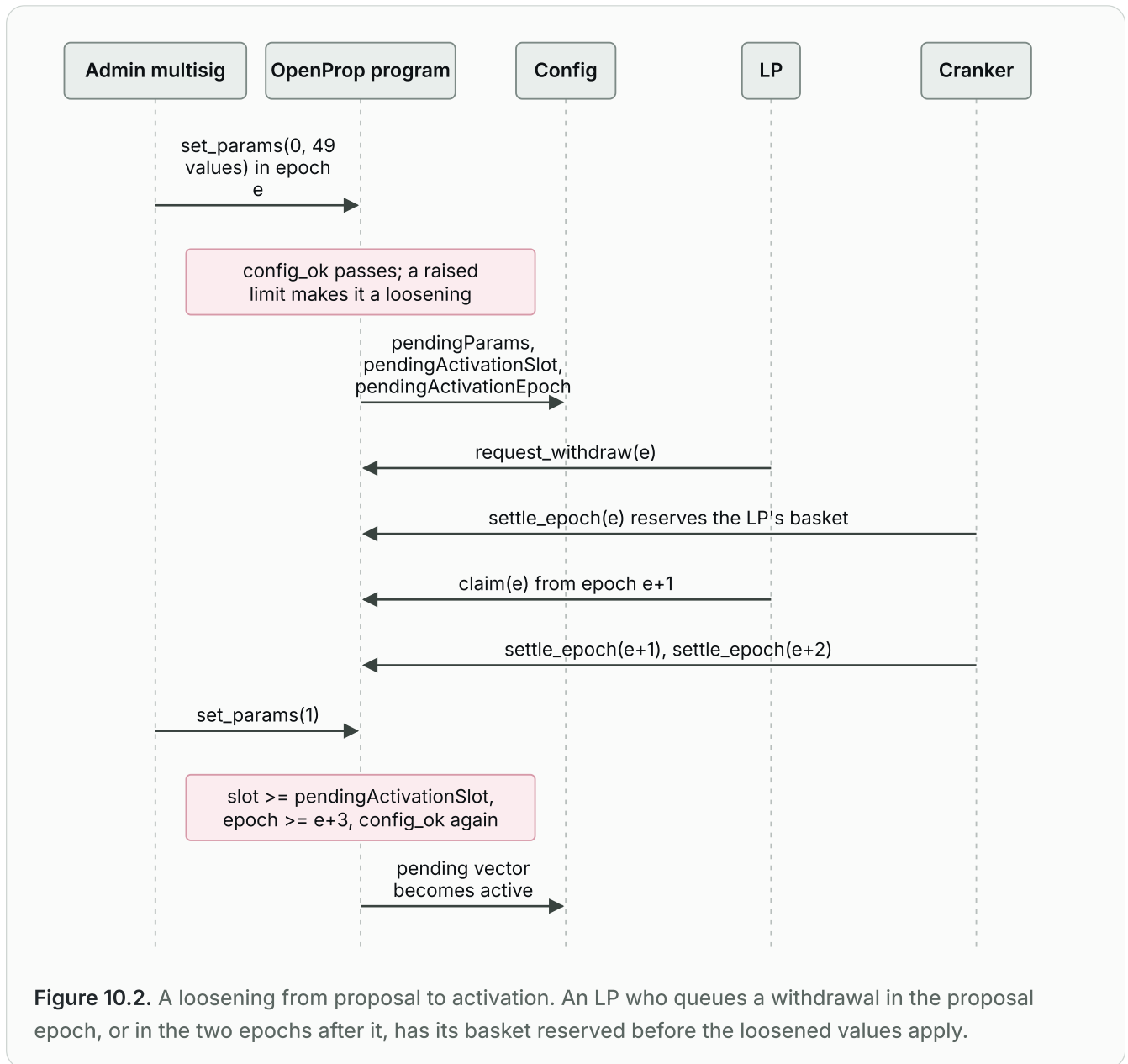
With the v0 values a loosening waits 1,296,000 slots, 72 hours at 200 ms, and until the vault reaches epoch $e_{\text{prop}} + 3$.

Apply (1). The admin signs again. The program requires a pending proposal, $\text{clock.slot} \geq s_{\text{act}}$ and $\text{LpVault.epoch} \geq e_{\text{act}}$, and fails with TIMELOCK_PENDING (6028) otherwise. It runs config_ok on the candidate once more, copies the pending vector into the active parameters and clears both deadlines.

Cancel (2). Clears both deadlines. A new proposal replaces a pending one and restarts both deadlines. A tightening also clears it, so an apply can never overwrite a later tightening.

The two deadlines protect the LP exit from opposite failures. An LP who requests a withdrawal in epoch e_{prop} , or in any of the W epochs after it, has a basket reserved at that epoch's settlement, before e_{act} arrives. If

settlement stalls, the slot deadline can pass while no reservation has been made; the epoch deadline then holds the change until the queued withdrawals have been served. If the vault is behind the epoch grid and a backlog of epochs is settled in quick succession, the epoch deadline can be reached early; the slot deadline then holds the change for the full $T \cdot E$ slots. For example, a proposal to raise skewBps at slot s in vault epoch 40 can be applied only from slot $s + 1,296,000$ and only once epochs 40, 41 and 42 have settled. A proposal that lowers hourlyLossBps from 50 to 40 takes effect in the transaction that carries it.



10.5 Identities and their rotation

Identity	Config field	Role
Admin	admin	Signs set_params, rotate_identities, resume_pair and initialize_pair. A Squads multisig
Guardian	guardian	Signs halt_pair and review_tier. A multisig

Identity	Config field	Role
Guardian halt signer	guardianHaltSigner	Signs halt_pair only. Zero disables it
Protocol USDC	treasuryToken	The USDC account of the ["protocol"] PDA that receives the protocol share. Fixed: a rotation must keep it
Cranker	cranker	Signs the buyback instructions, buyback_begin, buyback_burn and buyback_finish
Buyback mint	buybackMint, buybackTokenProgram	The token the protocol share buys and burns, \$OPENPROP, and its token program (classic Token or Token-2022). Zero disables buybacks
Pyth Pro signer	pythSigner, pythFeedId	Ed25519 key and feed identifier of accepted Pyth Pro payloads
Switchboard signer	switchboardSigner, switchboardFeed	Ed25519 key and feed hash of accepted Switchboard Surge quotes
Slot attester	pythSlotAttestor	Signs OPENPROP_PYTH_SLOT_V1 attestations that bind a Pyth payload to an authenticated Solana slot
Retail signer	retailSigner	Co-signer that selects the retail flow class, Jupiter's frontend signer per the design. Zero disables it
Mints	baseMint, quoteMint	SOL and USDC mints. Immutable

The cranker identity signs buybacks only. Any key may call `post_ref_price`, `checkpoint_book` and `settle_epoch`. Quoter keys belong to each Quoter account and are managed by its cold owner (Chapter 11).

`rotate_identities` (tag 21) is signed by admin and replaces the twelve keys and the Pyth feed identifier together. The proposal (action 0) carries a 392-byte vector: admin, guardian, treasuryToken, pythSigner, switchboardSigner, switchboardFeed, retailSigner, pythSlotAttestor, guardianHaltSigner, cranker, buybackMint, buybackTokenProgram, then pythFeedId as a u64. treasuryToken must equal the current value, so a rotation cannot redirect the protocol share. The program runs `config_ok` on the candidate, stores it in `pendingIdentities` and `pendingBuybackIdentities` and sets $\text{identityActivationSlot} = s_{\text{prop}} + T \cdot E$. Apply (action 1) requires that slot and validates the candidate again; cancel (action 2) clears it. The admin rotates its own key and the guardian's through the same path.

An identity rotation has the slot deadline only. Settlement needs a reference that the configured oracle keys signed, so a served-epoch deadline would let an oracle outage block the rotation that repairs it. A rotation changes no numeric limit, mint or envelope value, so it cannot loosen a risk limit.

The guardian's `review_tier` writes `Quoter.reviewed` (0 or 1). Promotion to tier 2 at settlement requires `reviewed = 1`, the v0 guardian review.

10.6 Initialization

`initialize_config` (tag 16) creates `Config` once. Its 912-byte body holds bytes 8 to 672 of the account (identities and the 46 parameters), then `pythSlotAttestor` and `guardianHaltSigner`, the three score policies, the eight envelope values, and `cranker`, `buybackMint` and `buybackTokenProgram`. `treasuryToken` must be the USDC account of the ["protocol"] PDA. The signer must be the program's upgrade authority, which the program reads from its canonical `Program` and `ProgramData` accounts under the upgradeable loader, and it

must equal the admin in the body. Only the program's upgrade authority can therefore create its configuration, and only once: the ["config"] address cannot be created a second time. `initialize_pair` (tag 17), signed by admin, then creates the Protocol account, PairConfig, RefPrice, LpVault, the vault's two token accounts and the OPLP share mint (6 decimals, mint authority LpVault), and records the first `epochStartSlot`.

10.7 The deployment's parameter file

A deployment's initialization values come from a parameter file. The file names a profile, the allowed clusters on which that profile may be used, and its purpose. It then holds three groups of values: the public catalog's values under their catalog names, such as `bandWidthBps`, `skewLimitBps` and `minimumBondMicros`; a chain block whose keys are the Config field names passed to `initialize_config`, including the eight envelope values; and a referenceEngine block for the quoter SDK's reference engine (`halfSpreadBps`, `levelStepBps`, `levelSizesBaseAtoms`, `inventorySkewBps`, `referenceMoveBps`, `refreshAfterSlots`, `retailDeltaBps`), which never reaches the chain.

The development profile, `dev-only`, lists only `localnet`, and services refuse it on `devnet` and `mainnet-beta`. Each public cluster is initialized from its own profile with the same names. No value that the design leaves open is copied from one profile to another. The API publishes two views: `parameters`, the catalog, in which each entry is either set to a value the design states or undecided, and `runtimeParameters`, read from the indexed on-chain Config. Once Config exists, the file has no further authority. Every later change is a `set_params` or `rotate_identities` transaction, visible on chain with its deadlines.

11. Security design

OpenProp places fund safety in the program. Every asset sits in a token account whose authority is a program-derived address, and the program signs a transfer out of such an account only inside a few instructions whose amounts and recipients it computes itself. Keys outside the program hold narrow powers: a quoter's keys manage its bond and its prices, the guardian can stop trading, and the admin can change parameters behind a timelock. None of them can sign a transfer out of the vault, a book or a bond. This chapter describes those controls as built: custody, the powers of each key, replay protection, the checks on every fill, timelocks, halts, compute bounds, inventory reconciliation, and how the indexer treats history it cannot prove.

11.1 Custody

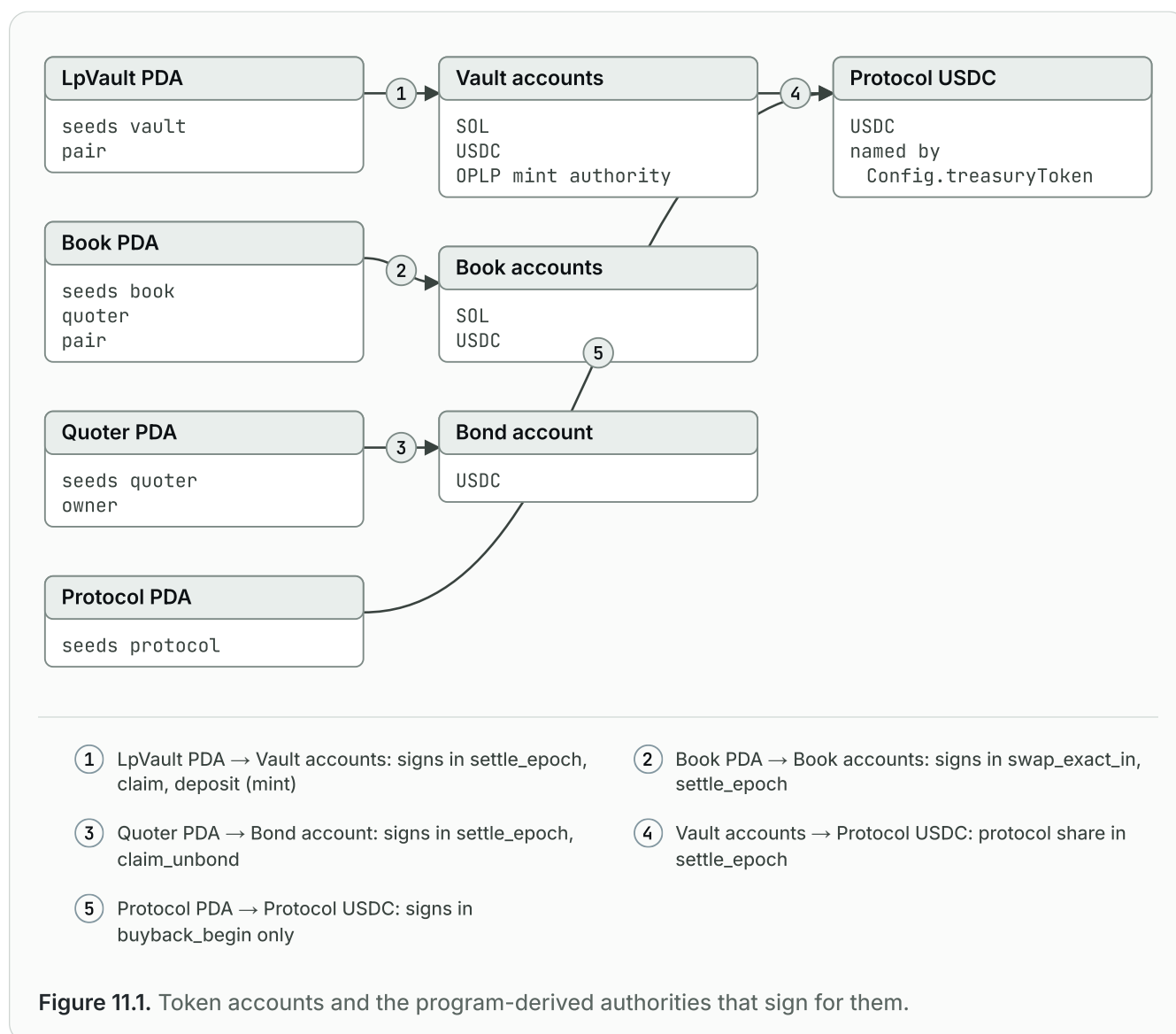
The vault and every book hold their assets in separate classic SPL Token accounts. The program creates each account at a program-derived address and initializes it with a program-derived authority. No instruction changes a token account's authority.

Token account	Address seeds	Authority	Instructions that debit it
Vault SOL	["vault_base", pair]	LpVault PDA ["vault", pair]	settle_epoch (allocations to books), claim (reserved basket to the LP)
Vault USDC	["vault_quote", pair]	LpVault PDA	settle_epoch (allocations, quoter and protocol shares), claim
Book SOL	["book_base", book]	Book PDA ["book", quoter, pair]	swap_exact_in (output to the taker), settle_epoch (recall to the vault)
Book USDC	["book_quote", book]	Book PDA	swap_exact_in, settle_epoch
Bond	["bond", quoter]	Quoter PDA ["quoter", owner]	settle_epoch (bond charge to the vault), claim_unbond (to the cold owner)
OPLP share mint	["shares", pair]	LpVault PDA, as mint authority	deposit mints; request_withdraw burns from the LP's own account under the LP's signature
Protocol USDC	the USDC account of ["protocol"], named by Config.treasuryToken	Protocol PDA ["protocol"]	buyback_begin only, inside a transaction that ends in buyback_finish (9.5); receives the protocol share in settle_epoch

Through the program, funds enter only under the signature of their owner: an LP signs the two transfers of deposit, a cold owner signs the bond transfer of register_quoter and add_bond, and a taker signs the input transfer of swap_exact_in. A swap makes exactly two SPL Token transfer CPIs and no other cross-program call. The admin has no instruction that moves tokens. The Protocol PDA signs one transfer, the claim of buyback_begin, and the same transaction must then burn at least the quoted amount of the buyback mint and leave the cranker no USDC or tokens from it, or it fails whole.

Every instruction checks each account it receives. A program account must be owned by the program, have its exact length, and carry version byte 6 and its kind byte; otherwise it fails with INVALID_ACCOUNT (6001). A

token account must be owned by the SPL Token program, be 165 bytes and initialized, and carry the configured mint and the expected authority; otherwise it fails with `INVALID_TOKEN_ACCOUNT` (6029). Links between accounts are compared byte for byte: a book's token accounts must be the ones it records, its pair must be the reference's pair, and its quoter must point back to it.



11.2 Signer roles

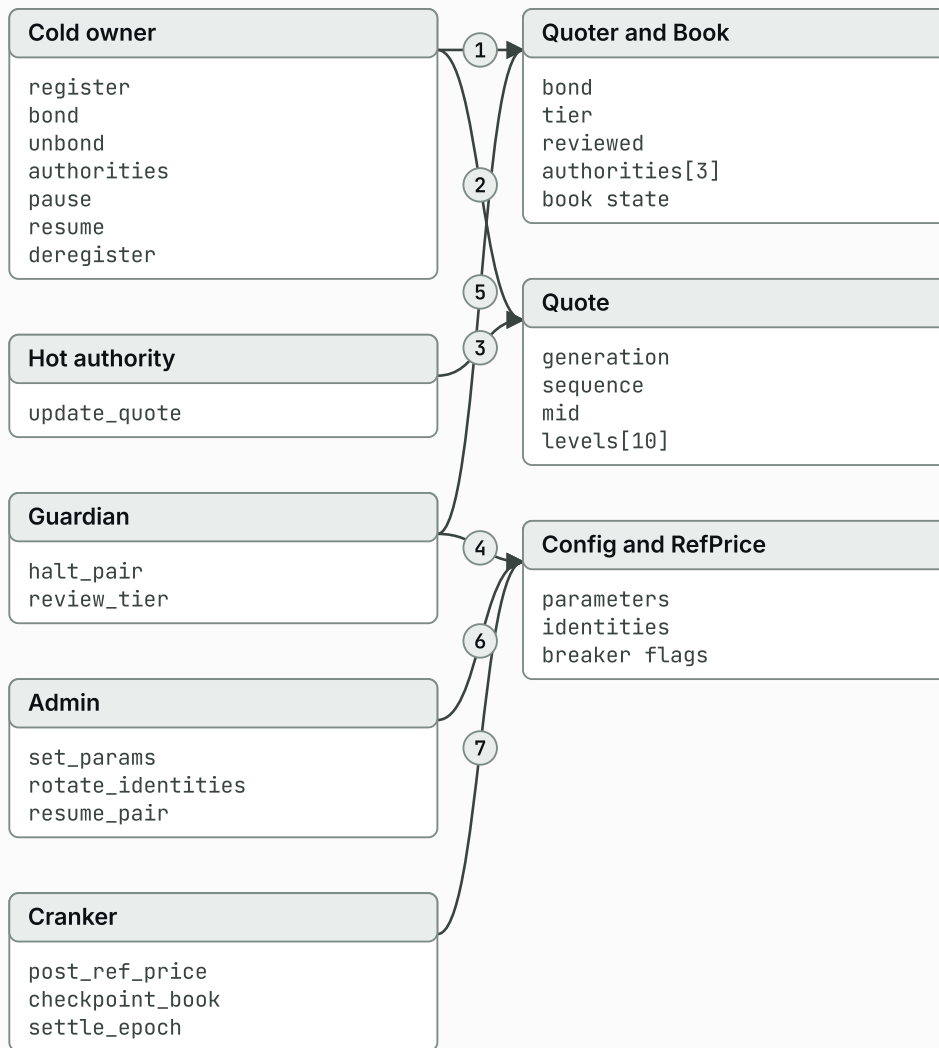
Key	Instructions it signs	What it cannot do
Cold owner, one per Quoter	<code>register_quoter</code> , <code>add_bond</code> , <code>request_unbond</code> , <code>claim_unbond</code> , <code>set_quote_authority</code> , <code>pause_book</code> , <code>resume_book</code> , <code>deregister_quoter</code>	Withdraw book or vault assets; resume a book halted by a loss limit; release bond still needed as collateral; install a hot key without that key's signature
Hot authority, up to three per quoter	<code>update_quote</code>	Write any account other than its Quote; move tokens; change authorities
Taker	<code>swap_exact_in</code>	Fill outside the checks of 11.4

Key	Instructions it signs	What it cannot do
LP	deposit, request_withdraw, claim	Claim before the ready epoch or claim another owner's ticket
Cranker, any key	post_ref_price, checkpoint_book, settle_epoch	Choose a price, an amount, a recipient or an epoch
Guardian	halt_pair, review_tier	Resume, change parameters, quote, move funds
Guardian halt signer	halt_pair	Anything else
Admin	set_params, rotate_identities, resume_pair, initialize_pair	Move funds; apply a loosening before both deadlines; change clocks, mints or the envelope
Upgrade authority	initialize_config, once	Anything after initialization
Oracle signers, slot attester	Off-chain messages verified through the Ed25519 precompile	Sign a program instruction
Retail signer	Co-signs a swap to select the retail offsets	Anything else

The cold owner's bond is encumbered three ways. `request_unbond` releases at most the bond minus pending unbonds, minus the collateral $\lceil A/\Lambda_t \rceil$ for allocation A at tier leverage Λ_t , minus profit tranches still locked at the ready epoch, and fails with `UNBOND_EXCEEDS_AVAILABLE` (6031) beyond that. `claim_unbond` pays pro rata from the bond that remains free when the ticket matures. A pause cannot overwrite a risk halt, and `resume_book` lifts only a pause. A risk halt ends at settlement.

A hot authority writes only the Quote account. It cannot reach a token account, and its prices fill only inside the band, skew and loss limits. Swaps by the cold owner or by a current hot authority still change the book's inventory, losses and settlement, but add nothing to its score or scored turnover. Every newly added hot key co-signs `set_quote_authority`, so a cold owner cannot attach an unrelated wallet to its book.

The protocol cranker refuses to run with a privileged key: the admin, guardian, halt signer, oracle signers, retail signer, slot attester or any quoter's cold owner. The guardian bot runs with the halt signer only, which Config requires to differ from the admin, guardian, oracle, retail and attester keys. Its adapter can build exactly one instruction, `halt_pair`.



- ① Cold owner → Quoter and Book: bond, authorities, pause state
- ② Cold owner → Quote: authority cache, generation
- ③ Hot authority → Quote: sequence, mid, levels
- ④ Guardian → Config and RefPrice: halt flags
- ⑤ Guardian → Quoter and Book: tier-two review
- ⑥ Admin → Config and RefPrice: parameters, identities, manual resume
- ⑦ Cranker → Config and RefPrice: reference price, breaker trips

Figure 11.2. Keys and the program accounts each can change. No key appears with a token account.

11.3 Replay protection

Operation	Replay key	A repeat
deposit, register_quoter, add_bond	OperationReceipt at ["receipt", signer, requestId], binding the instruction tag and the original payload	Same payload: success with no transfer. Different payload: <code>REPLAY_PAYLOAD_MISMATCH</code> (6024)

Operation	Replay key	A repeat
request_unbond	UnbondTicket at ["unbond", quoter, requestId]	Same amount: success with no effect. Different amount: 6024
request_withdraw	WithdrawTicket at ["wd", owner, epoch]	Same shares and minimums: success with no effect. Different: 6024
claim, claim_unbond	The ticket's claimed and remaining fields	A claimed ticket is a no-op; a payout never exceeds the remainder
checkpoint_book	Expected epoch and checkpointSlots spacing	No-op within the spacing; a wrong epoch fails
settle_epoch	LpVault.lastSettledEpoch	No-op for a settled epoch
post_ref_price	Pyth timestamp strictly increasing, source slot non-decreasing	Identical proof: no-op. Older proof: INVALID_ORACLE (6027)
update_quote	Quote.generation and a strictly increasing Quote.sequence	STALE_SEQUENCE (6014)

Monetary request IDs are u64 values chosen by the client. A client keeps one ID and its original payload for one logical operation, so a retry after a timeout cannot transfer twice.

A quote update names a generation and a sequence. The program accepts it only if the generation equals `Quote.generation` and the sequence is greater than `Quote.sequence`. Every change to the authority list increments the generation, resets the sequence to zero and clears the stored mid and slot, so the book has no live quote until a new update lands. Without the generation, a packet signed by a removed key could be accepted again after that key was re-added and the sequence restarted. With it, the old packet names an old generation and fails. Repeating an identical list is a no-op that keeps the generation, sequence and quote. An explicit reset is accepted only with a key set disjoint from the previous one. The quoter SDK persists each issued generation and sequence before it returns a signed packet and holds an exclusive lock on the Quote it signs for.

Reference proofs cannot be replayed into a different context. `post_ref_price` reads three Ed25519 verification instructions that precede it in the same transaction and must reference their own data. The keys must equal `pythSigner`, `switchboardSigner` and `pythSlotAttestor`. The Switchboard message's signed slot hash must appear in the `SlotHashes` sysvar at the claimed source slot, which must be no older than `refStalenessSlots`. The attestation binds the program ID, the `Config` address, that slot and slot hash, and the full Pyth payload. The cranker journals each signed packet before broadcast and, after a timeout, resends the identical bytes rather than producing a new proof.

11.4 The checks on every fill

`swap_exact_in` evaluates a fill with the shared rules engine, the same Rust source that the SDK, the aggregator adapter and the browser evaluator use. Before evaluation the program checks the accounts as in 11.1, requires each book token balance to cover the tracked inventory, and treats the swap as retail only if an eleventh account equal to `Config.retailSigner` signed it. The rules then run in order.

Check	Rule	Error
Epoch open	<code>Clock.slot < Book.epochEndSlot</code>	EPOCH_NOT_READY (6026)
Configuration	Risk parameters set and in range	UNDECIDED_PARAMETER (6003)
Book state	Live, neither halted nor paused	BOOK_HALTED (6009), BOOK_PAUSED (6010)
Ladder	Mid > 0; offsets below 10,000 bps and non-decreasing by level; sizes $\leq \text{maxLevelBaseAtoms}$; retail delta $\leq \text{retailDeltaLimitBps}$	QUOTE_LADDER_INVALID (6034)
Fresh quote	$t - t_q \leq \text{quoteStalenessSlots} (2)$	QUOTE_STALE (6004)
Reference	Every source slot within <code>refStalenessSlots</code> (10); $c \cdot 10^4 \leq P \kappa_c$; $ p_1 - p_2 \cdot 10^4 \leq P \kappa_d$	REF_STALE (6005), CONFIDENCE_TOO_WIDE (6007), REFERENCE_DIVERGENCE (6035)
Band	$ m - P \cdot 10^4 \leq P \kappa_b$	MID_OUTSIDE_BAND (6006)
Ladder walk and skew	$ V_B - Q \cdot 10^4 \leq (V_B + Q) \kappa_s$ after each level; partial fill at the limit	SKEW_LIMIT (6011), INSUFFICIENT_LIQUIDITY (6016)
Hourly loss	Projected $L_H \cdot 10^4 \leq A \cdot \text{hourlyLossBps}$	HOURLY_LOSS_LIMIT (6012)
Epoch loss	Projected $L_E \cdot 10^4 \leq A \cdot \text{epochLossBps}$	EPOCH_LOSS_LIMIT (6013)
Pair breaker	RefPrice halt bit clear	PAIR_HALTED (6008)
Slippage	Output $\geq$ the taker's minimum	SLIPPAGE_EXCEEDED (6015)

t is the current slot and t_q the quote's slot; m the quote mid and P the reference mid; c the reference confidence and p_1, p_2 the Pyth and Switchboard prices; $\kappa_b, \kappa_c, \kappa_d, \kappa_s$ are bandBps, confidenceBps, divergenceBps and skewBps; V_B is the book's SOL marked at P and Q its USDC; A is the allocation. Loss is projected on the post-fill inventory and compared at its larger value before and after the fill, so a profitable fill cannot reopen a window that is already breached. If the book starts the fill beyond the skew limit, each level may not worsen its starting ratio. A rejected swap writes no state and moves no tokens. A partial fill at the skew limit transfers only the input consumed and the output earned.

11.5 Timelocks

Change	Delay
Loosening a parameter	<code>timeLockEpochs</code> × <code>epochSlots</code> slots (3 epochs) and <code>withdrawEpochs</code> + 1 settled epochs (3), both fixed at proposal
Tightening a parameter	None, within the immutable envelope
Rotating an identity	<code>timeLockEpochs</code> × <code>epochSlots</code> slots
Clocks, mints, envelope	Cannot change after initialization
Unbonding	<code>unbondEpochs</code> epochs (2)
Releasing credited quoter profit	<code>profitLockEpochs</code> epochs

config_ok requires $\text{timelockEpochs} > \text{withdrawEpochs}$ and $\text{timelockEpochs} \geq 3$, and both clocks are fixed after initialization, so a loosening always waits longer than the LP withdrawal queue. Chapter 10 gives the deadlines and their derivation.

11.6 The guardian's halt and the program's breaker

RefPrice.flags carries the pair's state: bit 0 halts fills, bit 1 marks a manual halt. The program sets bit 0 itself through its breaker. post_ref_price trips it when the previous reference had gone stale (reason 2), the aggregate confidence exceeds confidenceBps (reason 3), the mid moved more than breakerMoveBps (300) against any price in the last breakerWindowSlots (150) slots (reason 4), or the two sources diverge beyond divergenceBps (reason 5). checkpoint_book trips it on a stale reference. Once a post shows no condition, the program records clearedSlot and sets resumeSlot = clearedSlot + resumeDelaySlots (50 slots). The first healthy post at or after resumeSlot clears bit 0. A new condition during the cooldown restarts it.

halt_pair, signed by the guardian or the halt signer, sets both bits with reason 1. The automatic recovery never clears a manual halt. Only resume_pair, signed by the admin, clears bit 1. It requires a reference no older than refStalenessSlots and starts the same cooldown, after which a healthy post lifts the halt. A repeated halt changes nothing and records no second incident.

A halt stops fills and deposits. It does not stop exits: request_withdraw, claim, request_unbond and claim_unbond do not read the halt bits. Checkpoints and settlement continue while the sources are fresh, confident and in agreement, so a price-move or manual halt still lets epochs settle and withdrawals be reserved. Books have their own halts: a checkpoint halts a book that crossed a loss limit (reasons 6 and 7), and the cold owner can pause its own book (reason 9).

Without an explicit policy the guardian bot only observes. A policy can authorize it to halt on sustained source divergence. The policy's threshold must exceed the on-chain divergenceBps, and its duration counts only advancing authenticated source and posted slots. A gap, restart or healed divergence resets the count, and the bot re-checks the condition immediately before broadcasting.

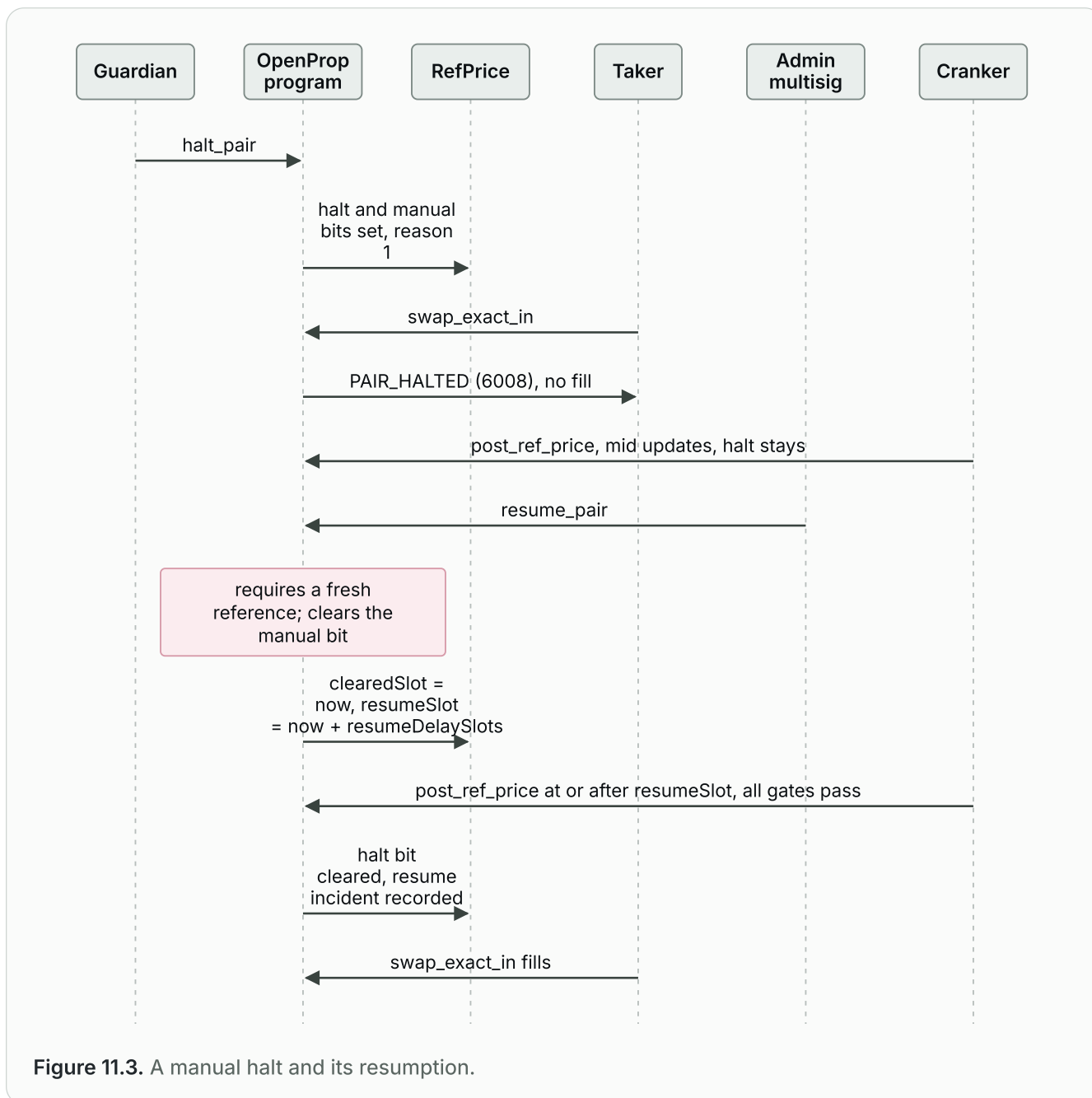


Figure 11.3. A manual halt and its resumption.

11.7 Compute and account-list bounds

The program has no heap allocator and writes no logs on the hot paths. Cold paths emit fixed-size binary records, 210 or 242 bytes. `update_quote` takes three accounts and has its own entrypoint path that deserializes only those three. It checks the signer against the authority cache in `Quote`, reads the `Clock` sysvar account directly, and accepts an optional index byte that names the authority's position. `swap_exact_in` walks at most five levels on one side, and its skew search is a binary search over a u64 quantity.

Measured on the release artifact executing in LiteSVM, against the design targets:

Path	Measured CU	Target
<code>update_quote</code> , indexed authority	178	200

Path	Measured CU	Target
update_quote, authority scan	183	200
Representative three-level swap	17,934	40,000
Five-level sell / buy	20,911 / 20,874	40,000
Largest skew partial fill	37,497	40,000

These are observations for the tested configurations, not upper bounds over every input. At about 40,000 CU per swap, the 12M CU cap per writable account allows roughly 300 fills per book per block (the v0 estimate).

Account lists are fixed by the registry. `settle_epoch` takes exactly $10 + 5n$ accounts and deposit exactly $13 + 5n$, where $n \leq 5$ is the number of registered books, supplied in registry order. A list that omits a book fails, so no book's inventory or liabilities can be left out of NAV or settlement. With five books a settlement uses a v0 transaction with an address lookup table. Every other structure is bounded: 150 breaker buckets, 64 profit tranches, 64-epoch trip bitmaps, at most 512 `SlotHashes` entries scanned, receipt payloads of at most 256 bytes. Monetary arithmetic is checked and fails with `ARITHMETIC_OVERFLOW` (6025). Score statistics use bounded, saturating arithmetic, so no statistic can stop a checkpoint, a settlement or a withdrawal.

11.8 Reconciliation of tracked inventory

Each book records its trading inventory in `trackedBase` and `trackedQuote`. Only `swap_exact_in`, by the exact amounts of the fill, and `settle_epoch`, which recalls the book and installs the next basket, change them. Every instruction that reads a book's token accounts requires the balances to cover the tracked amounts: `swap_exact_in`, `checkpoint_book`, `deposit` and `settle_epoch`. Every instruction that reads a bond account requires its balance to cover `Quoter.bond`: `deposit`, `settle_epoch` and `claim_unbond`. A shortfall fails with `INVALID_ACCOUNT`.

Excess, loss windows, skew and scoring use tracked amounts. Tokens sent to a book from outside raise its balance but never its excess or score, and settlement sweeps them into the vault, where they accrue to LPs. A transfer straight into a bond account does not raise `Quoter.bond`; only `register_quoter`, `add_bond` and a `quoter` share paid at settlement do. The indexer reports a positive difference between balance and tracked amount as `untrackedInventory` and fails the projection if a balance is ever below its tracked amount.

11.9 The indexer and unproven history

The indexer reads at `finalized` commitment. It takes every account page from one finalized bank and retries if the bank advances during pagination. Source instructions, the public snapshot and the cursor commit together in one PostgreSQL transaction under a per-cluster lock, so a slower poller cannot move the index backwards. The database never originates a balance or approves a movement of funds.

History must be contiguous. New signature pages must reconnect to a transaction the index already holds, and epoch transitions must run without a gap from initialization to the vault's current epoch. When an RPC node has pruned the history needed for that, indexing fails and requires an archival node. It does not fill the gap with zeros.

The indexer follows OpenProp instructions called directly and through CPI, using runtime log frames to decide each invocation's outcome. When logs are truncated, or an inner invocation's survival of its ancestors cannot be proven, the invocation is stored with status UNKNOWN and produces no fill, checkpoint or settlement record. The snapshot then reports `meta.historyEvidence.complete = false` with typed issues: LOGS_INCOMPLETE, INVOCATION_OUTCOME_UNKNOWN, SETTLEMENT_EVENTS_INCOMPLETE, REFERENCE_HISTORY_UNAVAILABLE, FILL_TRANSFERS_UNAVAILABLE, IDENTITY_HISTORY_UNAVAILABLE and TIMING_UNAVAILABLE. A fill is marked only against a reference proven by an accepted proof earlier in canonical order. An unproven settlement clears the current target-weight figures and the latest-settlement summary, while earlier proven epochs stay readable. A retail flow class that cannot be proven through a router is reported as unknown.

Current state does not depend on history. Balances, NAV, tickets and claimable amounts are read from finalized accounts and reconcile on their own. A request or activation slot that history cannot establish is flagged as unknown, never inferred from the latest bank.

12. Program architecture

OpenProp runs as one on-chain program, `openprop`, written against Pinocchio. The vault, the books, the quotes, the oracle gate, settlement and governance live in the same executable. A split into a vault program and an AMM program would put a cross-program invocation on every swap, so the program keeps them together. The crate is `#![no_std]` and declares `no_allocator!()`: no instruction can allocate heap memory, and a panic aborts the transaction. The hot paths emit no logs. `update_quote` makes no cross-program invocation at all, and `swap_exact_in` makes exactly two, the SPL Token transfers that move the input and the output.

The source is split by caller and cost: `hot.rs` holds `update_quote` and `swap_exact_in`; `oracle.rs` holds `post_ref_price`; `settlement.rs` holds `checkpoint_book` and `settle_epoch`; `cold.rs` holds LP, quoter and initialization instructions; `admin.rs` holds the guardian and admin instructions. `abi.rs` and `abi.json` are generated together by `generate-abi.mjs` and are the single description of every account offset, instruction tag, parameter order and event layout. Accounts are encoded field by field at explicit offsets, never by casting Rust memory. All integers are little-endian. Prices and USDC amounts use six decimals, SOL amounts use nine (lamports), and share amounts use six. Quote offsets are whole basis points.

12.1 Wire version and dispatch

Every instruction starts with two bytes, `[6, tag]`: the wire version, then the instruction tag. Instruction bodies have exact lengths; a body of the wrong length, an unknown tag or any version other than 5 fails with `INVALID_INSTRUCTION` (6000).

The entrypoint has two lanes. The loader input begins with the number of accounts. When that number is 3 and the instruction data is 150 or 151 bytes long and begins `[5, 0]`, the entrypoint deserializes only three account slots and calls `hot::update` directly. Every other instruction goes through the general lane, which deserializes up to the runtime's maximum account count and dispatches on the tag in `process_instruction`. The short lane exists because `update_quote` is the instruction a quoter sends many times a second; deserializing exactly three account slots keeps its fixed entry cost small.

The `update_quote` body is 148 bytes, or 149 with the optional authority index:

Offset in instruction data	Field	Type
0	wire version, 5	u8
1	tag, 0	u8
2	generation	u64
10	sequence	u64
18	mid	u64
26	ten levels, five bids then five asks, each (<code>offset_bps</code> u32, <code>size</code> u64)	120 bytes
146	<code>retailDeltaBps</code>	u32
150	authority index, optional	u8

The generation at offset 2 and the sequence at offset 10 are the two fields a landing service such as BAM's Maker Priority Plugin reads to deduplicate updates per market.

12.2 Accounts

Header, version and length

Every program-owned account starts with the same eight-byte header: version (u8), kind (u8), bump (u8), flags (u8), reason (u8) and three reserved bytes. Before reading any account, the program checks four things: the account is owned by the program, its data length equals the exact length of its kind, byte 0 is 6 and byte 1 is the expected kind. A mismatch fails with `INVALID_ACCOUNT` (6001). The hot update reads the first two bytes as one u16 and compares it against `kind << 8 | 6`, which costs one comparison.

The version byte and the exact length together select the decoder. Off-chain decoders in `@openprop/contract` dispatch on byte 0 and reject any account whose kind or length does not match the version-6 layout. Accounts of an earlier layout are never reinterpreted as version 6; a new layout version is a new deployment. `initialize_config` sets bit 0 of flags on Config when the program is built with the `local-dev` feature, which marks a development-only configuration that public-cluster setup refuses. Bit 1 of Config's flags is the open-buyback guard.

The flags and reason bytes carry state that the hot path must read cheaply. On Book, flags is the book state (0 live, 1 halted, 2 paused) and reason is the incident reason. On RefPrice, bit 0 of flags means the pair is halted and bit 1 means the halt was manual. On Quoter, bit 0 marks a deregistered seat.

The account set

Account (kind)	Seeds	Length	What it holds	Written by
Config (0)	["config"]	1,728	twelve identities, both mints, pythFeedId, the 49 active parameters, a pending parameter set with its activation slot and epoch, a pending identity vector with its activation slot, the eight immutable envelope values	initialize_config, set_params, rotate_identities; the buyback guard bit in buyback_begin and buyback_finish
PairConfig (1)	["pair", baseMint, quoteMint]	168	the config, both mints, the reference and the vault of the pair	initialize_pair
RefPrice (2)	["ref", pair]	3,776	mid, confidence, sourceSlot, postedSlot; each source's price, confidence, slot and timestamp; breaker trippedSlot, clearedSlot, resumeSlot; replay cursors; a 3,600-byte ring of 150 per-slot buckets for the move breaker	post_ref_price, checkpoint_book, halt_pair, resume_pair
LpVault (3)	["vault", pair]	408	token accounts, share mint, epoch, epochStartSlot, lastSettledEpoch, share supply, reserved and allocated inventory, bookCount, the registry of up to five books, deferred fee liabilities	LP instructions, registration, settlement

Account (kind)	Seeds	Length	What it holds	Written by
Quoter (4)	["quoter", owner]	1,304	cold owner, up to three hot authorities, bondToken, bond, pendingUnbond, tier, reviewed, completedEpochs, scoreE6, trip bitmaps, locked profit and its 1,024-byte tranche table	cold owner, checkpoint, settlement, guardian review
Book (5)	["book", quoter, pair]	576	tracked inventory, start inventory, allocation, epoch and cumulative excess, highWaterMark, hourly and epoch loss windows, EWMA moments and checkpoint count, last fill, fill counters and turnover, scoring deltas and residual	swap_exact_in, checkpoint, settlement, cold owner
Quote (6)	["quote", book]	304	sequence, slot, mid, retailDeltaBps, ten levels, the cached authority list and generation	update_quote; set_quote_authority for the cache
WithdrawTicket (7)	["wd", owner, epoch]	144	shares, minimum outputs, fixed claim amounts, readyEpoch, claim state	request_withdraw, claim
UnbondTicket (8)	["unbond", quoter, requestId]	136	requested amount, readyEpoch, remaining, cumulative paid and haircut	request_unbond, claim_unbond
WithdrawBatch (9)	["withdraw_batch", vault, epoch]	104	the epoch's queued shares, reserved SOL and USDC, claimed totals	request_withdraw, settle_epoch, claim
OperationReceipt (10)	["receipt", owner, requestId]	320	owner, request ID, instruction tag and the exact request payload	deposit, register_quoter, add_bond
Protocol (11)	["protocol"]	272	the config; during a buyback, the cranker, its two token accounts, the mint and token program, the begin and finish indices, the amount, min_burn, and the balances and supply recorded at the start	initialize_pair, buyback_begin, buyback_finish

Numeric seed components (epoch, requestId) are u64 little-endian. Lengths are in bytes. The field-by-field layouts are reproduced in the appendix and in `abi.json`.

Quote is deliberately a separate account from Book. An update write-locks only the 304-byte Quote; a swap reads Quote and writes Book. The quoter's update traffic therefore never write-locks the account that swaps mutate. The cached authority list and generation sit after the ladder: `update_quote` writes only the fields before them, and only `set_quote_authority` refreshes the cache, so a hot key can never widen its own permissions.

In wire 6 every risk parameter lives in Config. `PairConfig` binds the pair's mints to its reference account and its vault; it stores no limits of its own.

Token accounts and custody

The program creates six kinds of classic SPL Token account, all at program-derived addresses, and each owned by a program account that only the program can sign for:

Token account	Seeds	Mint	Token authority
vault SOL, vault USDC	["vault_base", pair], ["vault_quote", pair]	SOL (9 decimals), USDC (6 decimals)	LpVault
share mint (82 bytes)	["shares", pair]	6 decimals	LpVault
bond	["bond", quoter]	USDC	Quoter
book SOL, book USDC	["book_base", book], ["book_quote", book]	SOL, USDC	Book

`initialize_pair` checks the decimals of both mints. The Book PDA signs only inside `swap_exact_in` (output transfer) and `settle_epoch` (rebalancing). The Quoter PDA signs only bond payouts in `claim_unbond` and settlement. The LpVault PDA signs claims and settlement transfers. No instruction gives a quoter's cold key, hot key, the guardian or the admin a path to move these balances.

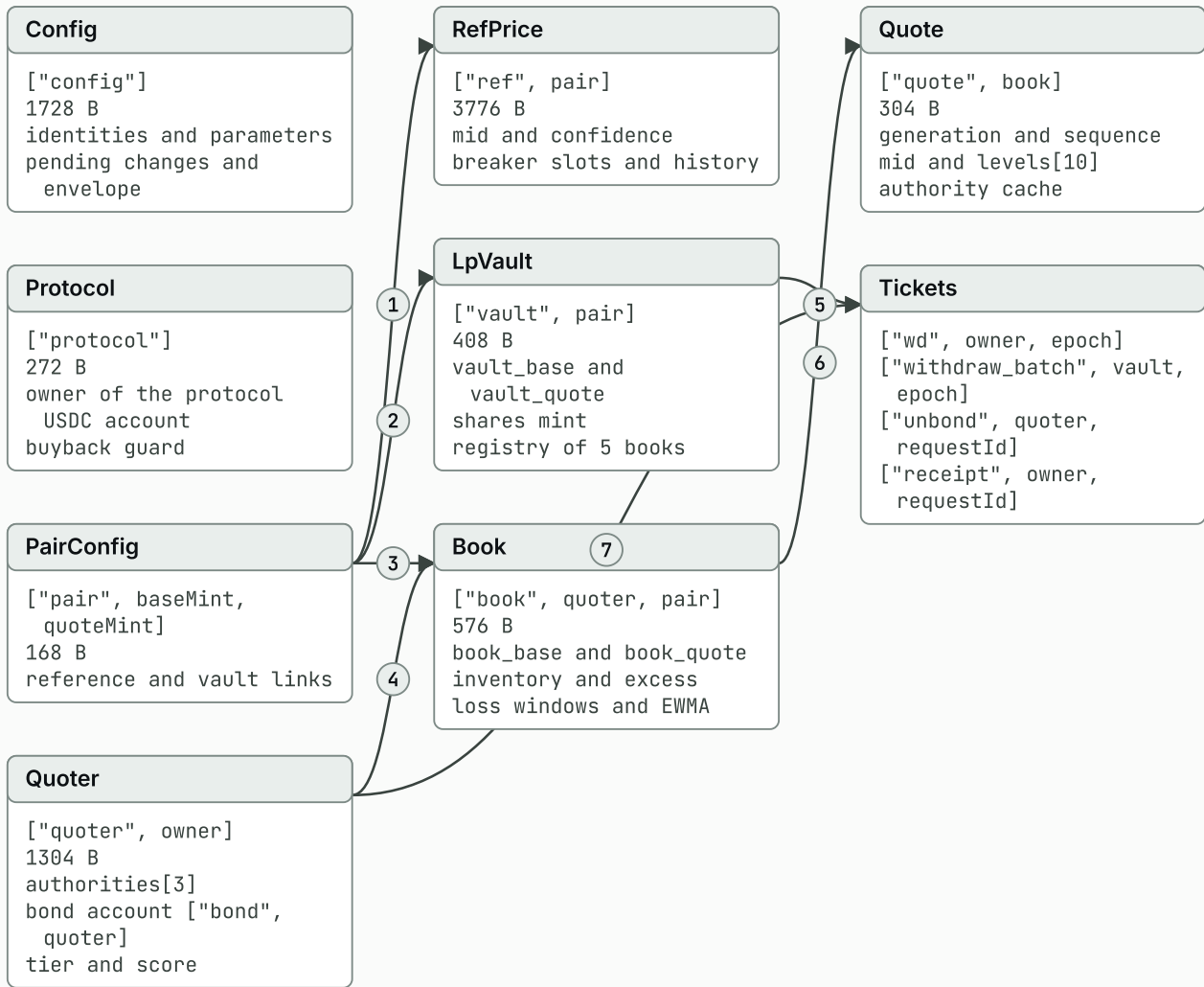


Figure 12.1. Program accounts and the seeds that derive them

12.3 Instructions by caller

The program has 23 instructions, tags 0 to 22. Grouped by the key that signs them, each group can reach only the accounts listed. In the account lists, S marks a signer and W a writable account; positions are fixed and ordered as shown.

Quoter hot key

Tag	Instruction	Body	Accounts
0	update_quote	see §12.1	hot S, Quote W, Clock sysvar

The hot key must appear in the cached list on Quote. With the authority index, the program compares one 32-byte slot; without it, it scans up to three. The body's generation must equal the stored one, and sequence must be strictly greater than the stored sequence (STALE_SEQUENCE, 6014). The program verifies the Clock sysvar's address and 40-byte length and writes `Clock.slot` as the quote slot. The ladder itself is validated on every fill, not on every update.

Quoter cold key

Tag	Instruction	Body	Key accounts
8	register_quoter	request ID, bond (2 × u64)	owner SW, config, pair, vault W, quoter, bond, book, quote, book SOL and USDC (all W, created), owner USDC W, both mints, receipt W
9	add_bond	request ID, amount	owner SW, config, quoter W, bond W, owner USDC W, receipt W, book
10	request_unbond	request ID, amount	owner SW, config, vault, quoter W, book, unbond ticket W, bond W, owner USDC W
15	claim_unbond	request ID	as request_unbond, with the token program in place of the system program
11	set_quote_authority	count u8, 0–3 keys, optional reset byte	owner S, quoter W, quote W, then each newly added hot key S
12, 19	pause_book, resume_book	empty	owner S, quoter, book W
22	deregister_quoter	empty	owner S, config, vault W, quoter W, book W, quote W, book SOL, book USDC

Every newly added hot key co-signs its own installation. Each actual change of the list writes Quoter and Quote together, increments generation, resets sequence to zero and clears the live quote, so a packet signed under an earlier list cannot land. Registration fails with REGISTRY_FULL (6038) when the vault already holds `maxBooks` books.

LP

Tag	Instruction	Body	Key accounts
5	deposit	request ID, maxBaseAtoms, maxQuoteAtoms, minShares	owner SW, config, reference, vault W, vault SOL W, vault USDC W, share mint W, owner SOL, USDC and share accounts W, receipt W; then every registered (book, book SOL, book USDC, quoter, bond)
6	request_withdraw	epoch, shares, minimum SOL, minimum USDC	owner SW, config, vault W, share mint W, owner shares W, ticket W, batch W

Tag	Instruction	Body	Key accounts
7	claim	epoch, optional replacement minima	owner S, config, vault W, vault SOL W, vault USDC W, ticket W, batch W, owner SOL W, owner USDC W

deposit takes every registered book in the vault's registry order, including halted and paused books, so the share price includes all inventory and all pending first-loss coverage.

Taker

Tag	Instruction	Body	Accounts
1	swap_exact_in	side u8 (0 sell SOL, 1 buy SOL), input u64, minimum output u64	taker S, config, reference, book W, quote, book SOL W, book USDC W, taker SOL W, taker USDC W, token program; optional retail signer S

The optional eleventh account must equal `Config.retailSigner` and must sign for the retail offsets to apply.

Cranker (any signer)

Tag	Instruction	Body	Accounts
2	post_ref_price	Pyth index u16, Surge index u16, source slot u64, attestation index u16	config, reference W, Instructions sysvar, SlotHashes sysvar
3	checkpoint_book	expected book epoch u64	config, reference W, book W, book SOL, book USDC, quoter W
4	settle_epoch	epoch u64	caller SW, config, reference, vault W, vault SOL W, vault USDC W, protocol USDC W, withdrawal batch W; then every registered (quoter, bond, book, book SOL, book USDC), all W

post_ref_price takes no signer account: its authority comes from three earlier Ed25519 verification instructions in the same transaction, whose public keys must equal the configured Pyth signer, Switchboard signer and slot attestor.

Buyback (the configured cranker)

Tag	Instruction	Body	Accounts
23	buyback_begin	amount u64, min_burn u64	cranker SW, config W, protocol W, protocol USDC W, cranker USDC W, buyback mint W, cranker buyback-token account W, Token program, buyback token program, Instructions sysvar, USDC mint, Associated Token program, System program
24	buyback_burn	empty	as buyback_begin
25	buyback_finish	empty	as buyback_begin; must be the transaction's last instruction

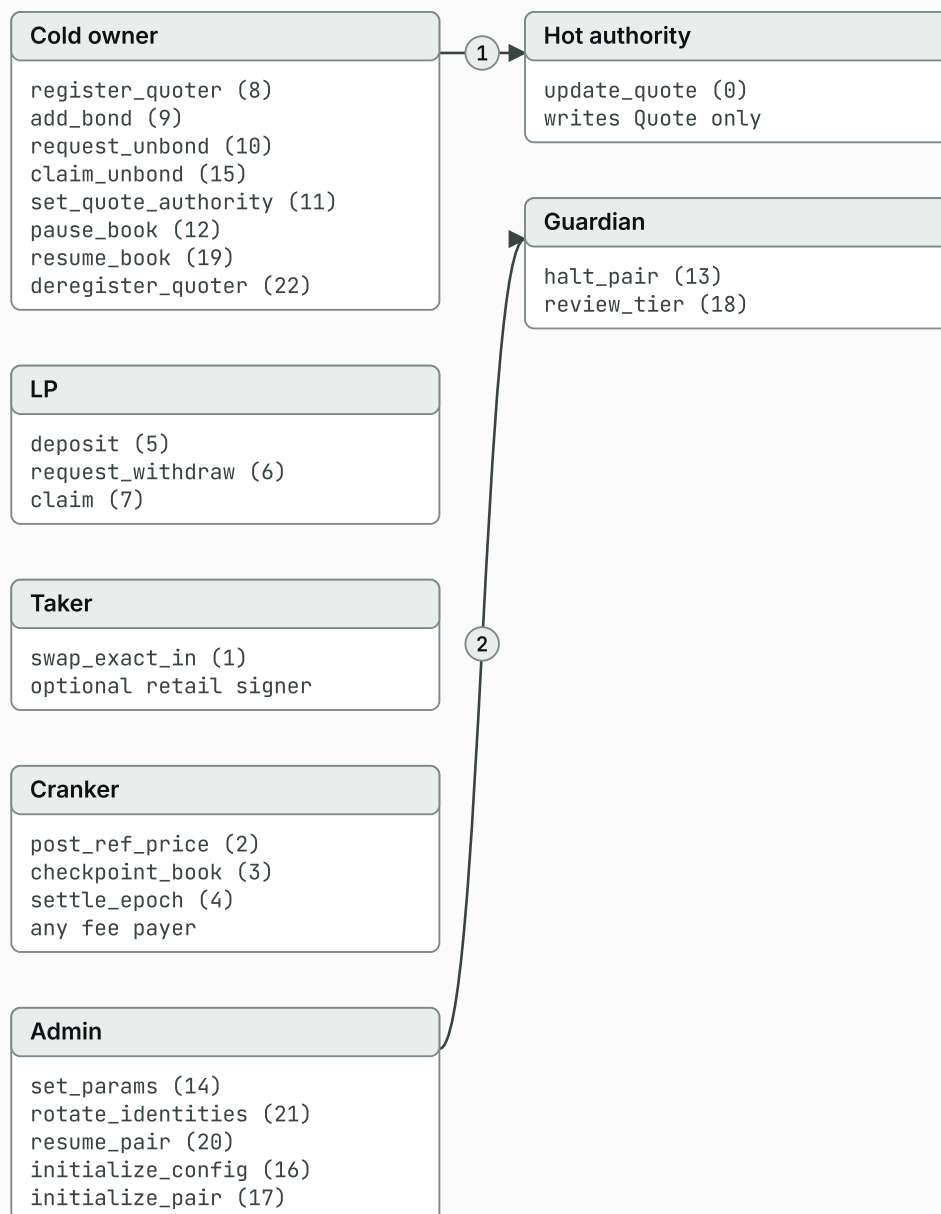
The three run in one transaction around a Jupiter route. buyback_begin must be a top-level instruction and checks the whole transaction through the Instructions sysvar before it moves anything: only compute-

budget instructions before it, then exactly one `buyback_burn` and a final `buyback_finish` with the same accounts. Section 9.5 gives the checks.

Guardian and admin

Tag	Instruction	Signer	Accounts
13	<code>halt_pair</code>	guardian or guardianHaltSigner	signer S, config, reference W
18	<code>review_tier</code>	guardian	guardian S, config, quoter W
14	<code>set_params</code>	admin	admin S, config W, vault
21	<code>rotate_identities</code>	admin	admin S, config W, vault
20	<code>resume_pair</code>	admin	admin S, config, reference W
16	<code>initialize_config</code>	upgrade authority, as admin	admin SW, config W, system program, program, ProgramData
17	<code>initialize_pair</code>	admin	admin SW, config, pair W, reference W, vault W, vault SOL W, vault USDC W, share mint W, both mints, token program, system program, protocol W

`set_params` and `rotate_identities` take an action byte: 0 proposes, 1 applies, 2 cancels. `initialize_config` verifies that the signer is the upgrade authority recorded in the program's canonical ProgramData account, so no first caller can capture the configuration.



① Cold owner → Hot authority: installs up to 3 keys

② Admin → Guardian: sets identity, timelocked

Figure 12.2. Instructions grouped by the key that signs them

12.4 Events

Cold paths record their outcomes with `sol_log_data` as fixed binary records; the hot paths emit none. Every record has the same header: version u8 at byte 0, kind u8 at 1, the subject public key at 2, epoch u64 at 34, slot u64 at 42, then signed i128 values from byte 50.

Version / kind	Bytes	Subject	Values, in order
1 / 1 checkpoint	210	book	excess, value, hold value, scoreE6, reference price, hourly loss, epoch loss

Version / kind	Bytes	Subject	Values, in order
3 / 2 settlement	242	book	excess, LP share, quoter share, protocol share, bond charge, LP loss, new allocation, old allocation, high-water mark, reference price, target weight (bps), cancelled unpaid quoter credit
1 / 3 incident	210	reference or book	reason, action, reference price
1 / 4 unbond claim	210	unbond ticket	requested amount, cumulative paid, cumulative haircut, remaining

checkpoint_book emits a checkpoint record and, when a limit trips, an incident. settle_epoch emits one settlement record per registered book. halt_pair, post_ref_price, checkpoint_book and the pause instructions emit incidents. Incident actions are 0 halt, 1 resume and 2 pause; reasons are 1 guardian, 2 stale reference, 3 confidence, 4 price move, 5 source divergence, 6 hourly loss, 7 epoch loss and 9 owner pause. Unused value positions are zero.

12.5 The address lookup table

Two instructions take one group of accounts per registered book. deposit takes 13 fixed accounts plus 5 per registered book; settle_epoch takes 10 fixed accounts plus 5 per book. With the registry at its capacity of five books, a deposit names 38 accounts and the program, 39 keys. At 32 bytes per key the account list alone is 1,248 bytes, above the 1,232-byte packet limit before any signature or instruction data. These transactions are therefore v0 transactions that reference the deployment's address lookup table, where each listed account costs a one-byte index.

The table is created by the deployment's initialization. It holds the program, the protocol addresses (config, pair, reference, vault, both vault token accounts, share mint), both mints, the protocol USDC account, the SPL Token and System programs, the Clock, Instructions and SlotHashes sysvars, and the admin and cranker keys. A client uses the table only after the finalized slot has passed the table's last extension slot and the table is not deactivated. Small packets, such as a reference post or a halt, compile without it. Aggregator routes also stay on v0: the 4,096-byte v1 transaction format has no lookup tables.

12.6 Compute budgets

The design sets two targets: at most 200 CU for update_quote and at most 40,000 CU for a swap. The measurements below come from the program's measurement file, program/cu-measurements.json and the update breakdown files, produced by LiteSVM 0.6.1 executing the SBF release artifact. The file records the SHA-256 of the measured program and of the rules source, so each number is tied to one build.

Path	Measured CU	Target
update_quote, indexed, each of the three authority positions	178	200
update_quote, authority scan, worst tested position	183	200
swap_exact_in, representative three-level fill	17,934	40,000
swap_exact_in, five-level sell	20,911	40,000

Path	Measured CU	Target
swap_exact_in, five-level buy	20,874	40,000
swap_exact_in, small-book skew partial fill	30,974	40,000
swap_exact_in, large-book skew partial fill	37,497	40,000

The update stays small because it does only fixed work: verify the signer flag, the Quote owner and length, the header, the cached authority, the generation, the sequence and the Clock address and length, then write the sequence, slot, mid, retail delta and ladder. Of a swap's cost, the design estimates about 10k CU for the two SPL transfers; the rest is the checks and the ladder walk. A skew-limited partial fill costs more because the rules search for the largest admissible quantity below the skew limit.

Solana caps the compute that one block can spend on transactions writing the same account at 12M CU. Each swap writes its Book, so the cap bounds fills per book per block:

$$N_{\text{fills}} = \left\lfloor \frac{C_{\text{acct}}}{c_{\text{swap}}} \right\rfloor$$

where:

- N_{fills} is the number of swaps against one book that fit in one block;
- $C_{\text{acct}} = 12,000,000$ CU is the per-writable-account cap per block;
- c_{swap} is the measured cost of one swap.

At the most expensive measured swap, 37,497 CU, one book takes 320 fills per block; at a five-level sell, 20,911 CU, it takes 573. Every OpenProp transaction is far below the 1.4M CU transaction limit, and a hot-key update transaction is about 300 bytes.

12.7 The shared rules crate

The rules that decide a fill are not written twice. `openprop-rules` (version 0.6.0, `crates/rules`) is a `no_std`, allocation-free integer library with no networking, account access, token transfers or logging. It holds the swap evaluation (book state, quote validity and freshness, reference freshness, confidence and divergence, the reference band, skew and the five-level ladder walk, hourly loss, epoch loss, the pair breaker and the minimum output), inventory marking, excess over hold, loss-window marking, the settlement split, net asset value and the scoring arithmetic. Account authentication, oracle signatures, custody reconciliation and replay protection stay in the program.

The program links the crate directly. `swap_exact_in` builds a `SwapInput` from `Quote`, `RefPrice` and `Book`, a `RiskParameters` from `Config`, and a `BookRiskContext` from the book's epoch and window anchors, then calls `evaluate_runtime_swap_with_state`. A result with status 0 returns the rule's error code unchanged; otherwise the program writes the returned inventory and loss state and transfers exactly `input_consumed` and `output_amount`.

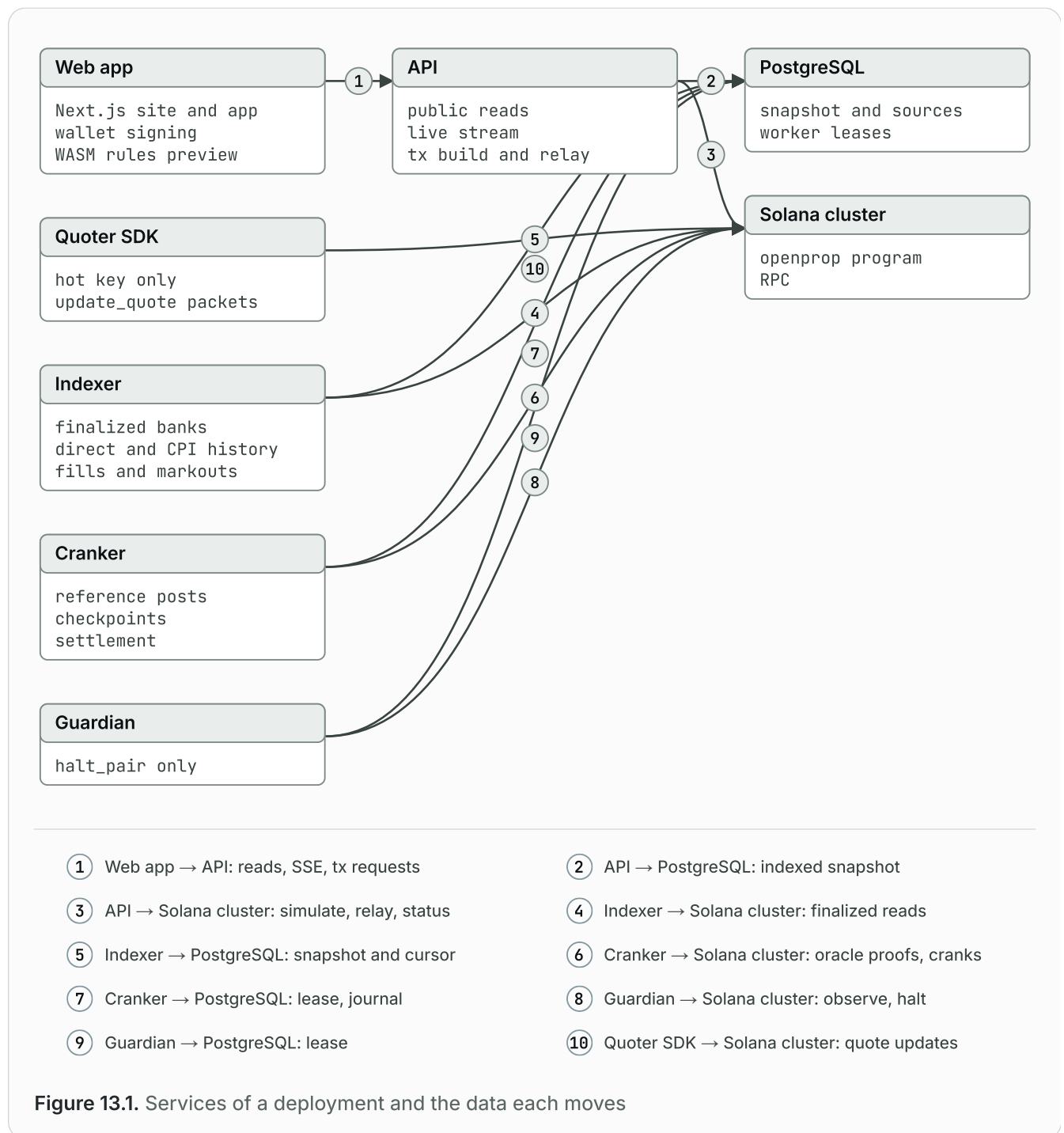
The same source compiles to WebAssembly through `crates/rules-wasm`, a thin `no_std` wrapper that exports `evaluate`, `mark_book_risk`, `settle_book`, `mark_net_asset_value`, `reference_health` and `mark_inventory_value` over fixed 64-word input and output buffers. The release build uses size optimization, link-time optimization and abort on panic. `scripts/build-rules.mjs` embeds the resulting bytes into the contract package together

with the crate version and the SHA-256 of `crates/rules/src/lib.rs`. The browser evaluator, the guardian's reference check, the quoter SDK and the Jupiter adapter all run this one artifact or crate.

Parity is maintained by construction and by test. By construction, both targets compile one source file, and every evaluation result reports `rulesVersion` and `rulesSourceSha256`, which can be compared with the `rulesSourceSha256` recorded next to the measured program in `cu-measurements.json`. By test, the native and WASM builds are run over generated and published inputs and must agree; the program parity check sends generated swap transactions to the actual SBF artifact in LiteSVM and compares token deltas, account balances, error codes and stored loss-window state with the WASM result for the same input; and the Jupiter adapter's parity suite checks its quotes against the same artifact. A change to the rules changes the source hash that every consumer reports, and any divergence between the program and a client fails the parity comparison.

13. Infrastructure and integration

The program is the only authority over custody and risk limits. The off-chain services of a deployment read the chain, build transactions for keys they do not hold, or send permissionless instructions that the program re-checks. No service holds an LP's, a taker's or a quoter's private key, and the database never originates a balance or approves a transfer. A deployment is the openprop program on devnet or mainnet-beta plus six services: the web app, the API, the indexer, the cranker, the guardian and PostgreSQL. Quoters run the quoter SDK on their own machines; aggregators load the Jupiter adapter in their own routers.



13.1 Deployment layout

`npm run program:deploy -- --cluster devnet|mainnet-beta` deploys the program with the deployer, program and fixed buffer keypairs from configuration (the fixed buffer lets an interrupted upload resume) and then checks the SHA-256 of the deployed bytes against the local `openprop.so`. `npm run program:init - - --cluster <cluster> --params config/parameters.<cluster>.json` creates Config with its envelope and identities, the Protocol account at ["protocol"] with its USDC account, the SOL/USDC pair with its vault and share mint, and the address lookup table, which it extends in chunks of 20 addresses and waits to see rooted. It reads the buyback mint from `OPENPROP_BUYBACK_MINT` and stores the mint's owner, classic Token or Token-2022, as its token program; without the variable the mint is zero and buybacks stay disabled. Rerunning it skips what exists.

Init writes every address to `deployments/<cluster>.json`, the manifest that all services read: wire version 6, cluster, programId, baseMint, quoteMint, genesisHash, lookupTable, protocol, protocolUsdc, buybackMint and buybackTokenProgram. On public clusters the manifest accepts only the native SOL mint (`So111111111111111111111111111111112`) as base and Circle-issued USDC as quote (`4zMMC9srt5Ri5X14GAgXhaHii3GnPAEERYPJgZJDncDU` on devnet, `EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v` on mainnet-beta).

Service	Start	Health	Role
web	next start	—	site, app, wallet signing
api	<code>npm start:prisma migrate deploy</code> , then the server	<code>/health</code>	reads, live stream, transaction build, relay, status
indexer	<code>npm run start:indexer (index --watch)</code>	—	finalized projection into PostgreSQL
cranker	<code>npm run start:cranker</code>	<code>/health/live</code>	references, checkpoints, settlement, buybacks
guardian	<code>npm run start:guardian</code>	<code>/health/live</code>	observation and halt_pair
PostgreSQL	managed database	—	shared state of the services

Backend services build with `npm ci --include=dev && npm run build`, which runs `prisma generate`. Only the API runs `prisma migrate deploy`, at start, once the private database network is reachable. Every service listens on the injected `PORT`, and `DATABASE_URL` references the database service (`${Postgres.DATABASE_URL}` on Railway).

13.2 PostgreSQL

The index stores raw source records (AccountObservation, ChainInstruction), typed projections (LpVault, Book, Quote, Fill, Checkpoint, Epoch, BookSettlement, Incident and others) and the IndexedSnapshot that the API serves. Workers store WorkerLease rows. Atomic amounts are `Decimal(20, 0)`, wide enough for any u64; signed cumulative values and fixed-point prices are `Decimal(39, 0)`; squared EWMA sums are `Decimal(78, 0)`. No amount passes through a floating-point number. The schema has no users, sessions or keys: wallets are public addresses.

13.3 Indexer

The indexer is its own long-running service. Every INDEX_POLL_MS (1,000 ms by default) it builds a picture of one finalized bank:

1. **Accounts.** `getProgramAccounts` at finalized lists every program account; a token-program scan filtered by size 165 and the share mint lists share holders, and its slot becomes the bank slot S . Referenced token accounts are fetched with `getMultipleAccounts` in pages of 100 pinned to S ; if any page answers from another slot, the batch is retried, up to five times.
2. **Transactions.** `getSignaturesForAddress` walks finalized signatures in pages of 1,000 back to the newest one stored. Each transaction is fetched as version 0 with its lookup-table addresses resolved, and `getBlockSignatures` fixes its order within the slot.
3. **Invocations.** Each OpenProp invocation is kept under a stable path: the outer instruction index, or `outer.flatInnerIndex` for a CPI such as an aggregator route. Runtime log lines are parsed into a frame tree, and a fill's token transfers are taken only from its own swap invocation's descendants.
4. **Commit.** Sources, projections, snapshot and cursor are written in one database transaction under a per-cluster advisory lock, so a concurrent indexer cannot move the head backwards.

Fills, checkpoints, settlements and incidents come from the decoded instructions and the events of chapter 12. For a fill of b lamports against q USDC micros:

$$\bar{p} = \left\lfloor \frac{q \cdot 10^9}{b} \right\rfloor, \quad \Delta_{\text{ref}} = \frac{(\bar{p} - P_{\text{ref}}) \cdot 10^8}{P_{\text{ref}}}, \quad M_h = \frac{(P_h - \bar{p}) \cdot d \cdot 10^8}{\bar{p}}$$

where:

- $\bar{p}$ is the average fill price in USDC micros per SOL;
- P_{ref} is the reference in force at the fill;
- P_h is the first reference observed after the fill, in canonical order, at an RPC block time at least h later, for h of 2 s and 30 s;
- d is +1 when the taker sold SOL to the book and –1 when the taker bought SOL;
- Δ_{ref} (fillVsReferenceBpsE4) and M_h (markout2sBpsE4, markout30sBpsE4) are basis points times 10^4 , truncated toward zero. A positive markout favours the book; with no qualifying reference it is null.

Worked example. In the sample book (mid and reference 200.00 USDC, bids at 1 and 2 bps of 1 and 2 SOL), a taker sells 3 SOL. Level 1 pays $200,000,000 \times 9,999/10,000 = 199,980,000$ micros for 1 SOL and level 2 pays $2 \times 199,960,000$, so $q = 599,900,000$, $b = 3 \times 10^9$ and $\bar{p} = 199,966,666$. Then $\Delta_{\text{ref}} = -16,667$, or -1.6667 bps. If the reference two seconds later is still 200.00 USDC, $M_{2s} = +16,669$: the book bought 1.6669 bps below the later reference.

Incomplete history. The indexer records only what it can prove. When an RPC truncates logs so that a routed call's survival is unproven, the invocation is stored as UNKNOWN and yields no fill. A fill without attributable transfers or without a preceding authenticated reference, and a ticket without a known request slot, add a typed issue (FILL_TRANSFERS_UNAVAILABLE, REFERENCE_HISTORY_UNAVAILABLE, TIMING_UNAVAILABLE) and set `meta.historyEvidence.complete` to false. Balances, NAV and claims still come from finalized accounts. If the RPC can no longer reach the stored history, the cycle stops without advancing the cursor and requires an archival RPC. The snapshot is UNREACHABLE after a failed cycle, STALE after five poll intervals without success or when it trails the chain by more than INDEX_STALE_SLOTS (50 by default), and HEALTHY otherwise.

13.4 API

The API serves the indexer's snapshot; it never runs a second index writer. Every read returns `{data, meta}`, where `meta.indexedSlot` names the finalized bank, `meta.chainSlot` follows confirmed progress and `meta.indexStatus` reports freshness.

Endpoint	Returns
GET <code>/v1/parameters</code> , GET <code>/v1/network</code>	parameter catalog; program, genesis hash, wire version, lookup table, public RPC URL
GET <code>/v1/vault</code> , GET <code>/v1/pair</code>	NAV, share prices, reserves and liabilities; reference and breaker
GET <code>/v1/books</code> , GET <code>/v1/books/:address</code>	books in registry order; one book's quote, inventory, limits and fills
GET <code>/v1/epochs</code> , <code>/v1/epochs/:epoch</code> , <code>/v1/incidents</code>	settlements and splits; halts, pauses and resumes
GET <code>/v1/buybacks</code>	every buy-and-burn with its mint, USDC spent, tokens burned and signature, and the running totals
GET <code>/v1/wallets/:address/position</code> , <code>.../quoter</code>	an LP position and tickets; a quoter's bond, tier, keys and unbonds
GET <code>/v1/errors</code> , POST <code>/v1/mechanism/evaluate</code>	program error catalog; shared rules on a submitted input
GET <code>/v1/live</code>	server-sent events

In production the API serves no fixture data, accepts only configured CORS origins, returns only a public RPC URL in `/v1/network`, and logs RPC method names but never endpoints. `/health` reports waiting (program-missing or config-missing) until the program and Config exist, then ready.

Live stream. Each update event on `/v1/live` replaces the client's whole window: network, vault, pair, books, each book's quote and recent fills. A snapshot comes first; tick events follow whenever the indexed slot, chain slot or index status changes. A client reconnecting with Last-Event-ID first receives a reset. Clients reconnect after 2 seconds; each may hold eight streams.

Server-built transactions. POST `/v1/tx/<action>` builds a v0 transaction with the named wallet as fee payer, for deposit, request-withdraw, claim, register, add-bond, set-quote-authority, request-unbond, claim-unbond, pause, resume and swap. It puts a `ComputeBudget` instruction first, creates any missing associated token accounts, and compiles over the deployment's lookup table once that table is rooted. On a native-SOL deployment, a temporary token account derived from the wallet with a random seed carries the SOL leg: it is funded with the deposit maximum or the sell amount, stands in for the wallet's SOL account, and is closed afterwards, which returns the output and any unused maximum as native SOL. The transaction must fit in 1,232 bytes. The API simulates it and returns `transactionBase64`, `blockhash`, `lastValidBlockHeight`, `requiredSigners` (including every newly added hot key for set-quote-authority) and `simulation`, with units consumed or the program's named error.

Relay and status. POST `/v1/tx/relay` takes a fully signed v0 transaction of at most 1,232 bytes. It accepts only instructions of `OpenProp`, `ComputeBudget`, `System`, `SPL Token`, `Associated Token`, `Ed25519` and `Lighthouse` (`L2TExMFKdjpN9kozasaPirfHy9P8sbXoAN1qA3S95`, which Phantom appends), requires at least one `OpenProp` instruction of the deployment's wire version (6), and sends through the server's RPC with

preflight. GET /v1/tx/<signature> returns pending, confirmed, finalized or failed, with the slot and the normalized error. These endpoints allow 60 requests per minute per client.

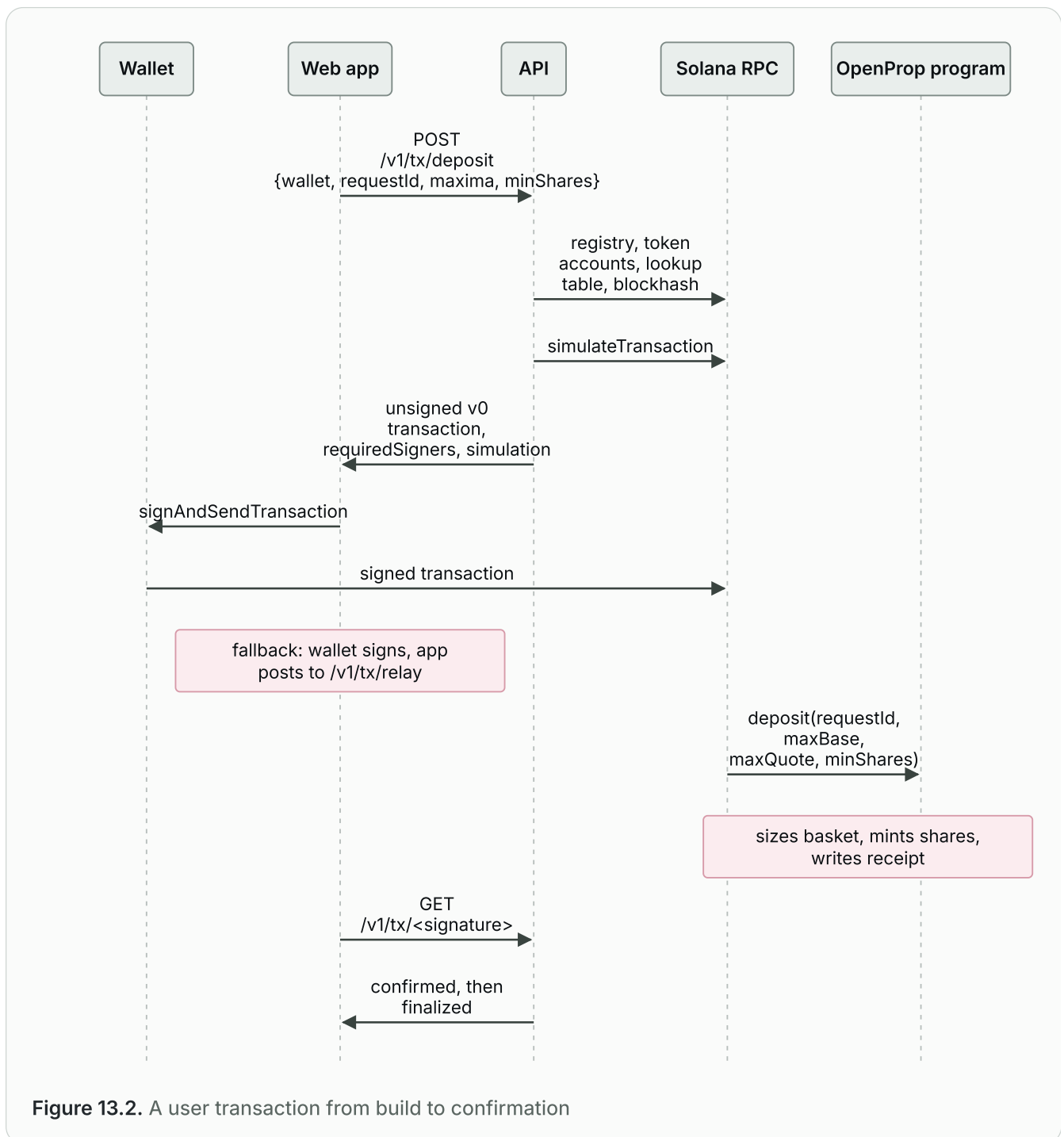


Figure 13.2. A user transaction from build to confirmation

13.5 The web app

The web app reads through the @openprop/contract client and subscribeLive. Previews run in the browser: evaluateMechanism executes the shared rules as embedded WASM, so a swap preview shows every check and level without a request, and previewDepositV5 estimates the consumed basket and shares from one snapshot. Signing goes through Wallet Standard wallets such as Phantom, Solflare and Backpack. The app requests the transaction from the API, shows the simulation, and calls the wallet's signAndSendTransaction; when a wallet only signs, the app posts the bytes to /v1/tx/relay. It then follows /v1/tx/<signature>.

normalizeTransactionError turns wallet, simulation and confirmation failures into OpenProp's named errors without attributing another program's codes to OpenProp. Keys never leave the wallet.

13.6 Cranker

Each cycle the cranker observes the chain and plans a **reference** job when the slot is past `RefPrice.postedSlot`, a **checkpoint** job for each allocated book once `checkpointSlots` have passed since its `lastCheckpointSlot`, and a **settlement** job when the epoch's end slot is reached. A reference post is one transaction: compute-budget instructions, three Ed25519 verifications (the Pyth Pro update, the Switchboard Surge update and the slot attestation), then `post_ref_price` naming their indices. A checkpoint appends `checkpoint_book`, so the book is marked against the reference just verified. Settlement is a separate transaction over the lookup table.

Buyback. After a settlement, once the protocol USDC account holds at least `BUYBACK_MIN_USDC_MICROS`, the cranker asks Jupiter (`lite-api.jup.ag/swap/v1/quote`) for an exact-in route from USDC to the buyback mint for the whole balance, with `maxAccounts` 48 and `BUYBACK_SLIPPAGE_BPS`, then fetches the route's instructions from `/swap-instructions`. It composes one v0 transaction over the deployment's and Jupiter's lookup tables: compute budget, `buyback_begin(balance, otherAmountThreshold)`, Jupiter's setup, swap and cleanup, `buyback_burn`, `buyback_finish` (9.5). The transaction is simulated and sent through the same journal as every other job. A failed quote waits for the next cycle; on localnet the cranker skips buybacks. The Pyth Pro update is fetched over HTTPS with the deployment's token, the Surge update comes from Switchboard's signed stream, and the slot attestor signs `OPENPROP_PYTH_SLOT_V1`, binding the Pyth payload to the program, `Config`, a source slot and its slot hash. The cranker refuses an admin, oracle or quoter cold key as fee payer.

Coordination runs through PostgreSQL. All cranker jobs share one lease row per cluster and program. Acquisition writes a fresh owner UUID with an expiry in database time, the holder renews every $\max(100 \text{ ms}, T_{\text{lease}}/3)$, where T_{lease} is `WORKER_LEASE_TTL_MS` (30 s by default), and it re-checks the lease immediately before broadcast. The signed packet, its blockhash and last valid block height are journaled before broadcast; after a timeout any cranker retransmits the same bytes until the finalized block height passes `lastValidBlockHeight`, then reconciles from account state. Each job retries on its own schedule:

$$t_{\text{retry}} = \min \left(B_{\text{max}}, T_{\text{poll}} \cdot 2^{\min(f, 10)} \right)$$

where f is the job's consecutive failures, T_{poll} is `WORKER_POLL_MS` (500 ms by default) and B_{max} is `WORKER_MAX_BACKOFF_MS` (30 s by default). A failing checkpoint therefore never delays reference posts.

13.7 Guardian

The guardian bot signs with `guardianHaltSigner`, which the program accepts for `halt_pair` only, and its adapter builds no other instruction. Without a policy it observes and reports health. With `GUARDIAN_DIVERGENCE_BPS` (above the on-chain `divergenceBps`, at most 10,000) and `GUARDIAN_DIVERGENCE_SUSTAINED_SLOTS` set, it halts the pair when the two signed sources disagree beyond that threshold across advancing source and posted slots for that many slots, using the shared rules' reference check. A failed observation, an invalid source, a sampling gap or a restart resets the window. The program's breaker handles staleness, confidence, transient divergence and moves; the admin resumes with `resume_pair`.

13.8 Quoter SDK

openprop-quoter-sdk is the Rust library a quoter runs beside its strategy. HotQuoter holds only the hot key, which is the transaction's sole signer and fee payer. It finds the key's position in the authority cache on Quote and signs the indexed update_quote. A packet holds a compute-unit limit, a compute-unit price and one update; it writes only Quote. The SDK runs with exactly one hot authority on the book and reads Config, RefPrice, Book and Quote with getMultipleAccounts.

Sequences. Before reading anything, the SDK takes an exclusive OS lock on a per-Quote file. SequenceTracker starts above both the chain's sequence and the persisted high-water mark of the same generation, and HotQuoter::sign durably writes {generation, last_issued} before returning signed bytes. A newer on-chain generation opens a fresh sequence space.

Reference engine. build_quote maps the reference P , the book's inventory and EngineParameters to a ladder:

$$\beta = \text{trunc}\left(\frac{(V - Q)k}{V + Q}\right), \quad m = P - \text{sgn}(\beta) \left\lfloor \frac{P|\beta|}{10^4} \right\rfloor, \quad o_i = h + i s \quad (i = 0, \dots, 4)$$

where V is the book's SOL marked at P in USDC micros, Q its USDC, k is inventorySkewBps, m the quoted mid, h is halfSpreadBps, s is levelStepBps, and o_i the offset of level i on both sides, sized by levelSizesBaseAtoms. A book long SOL lowers its mid. The engine requotes when the quote's age reaches refreshAfterSlots or the reference moves by referenceMoveBps, refuses a refreshAfterSlots above quoteStalenessSlots or a skew above bandBps, and validates the ladder with the shared rules. A ParameterHook supplies the parameters on every call, so a model or agent tunes spread, skew and sizes while the deterministic engine quotes at slot cadence.

Landing. land sends one signed packet down an ordered list of paths, through network clients the quoter supplies (the reference quoter sends through a standard RPC endpoint). On a BAM leader slot, if enrolled, it tries the Maker Priority Plugin, whose adapter admits only the enrolled signer, one update_quote writing the enrolled Quote, generation at byte 2, sequence at byte 10 and a compute price at the configured floor or above. It then tries the Block Engine (sendTransaction, base64), then the TPU. An acknowledgement only enqueues; only an observed landing stops the fallback. Every path carries the same bytes, and the sequence check makes duplicates harmless.

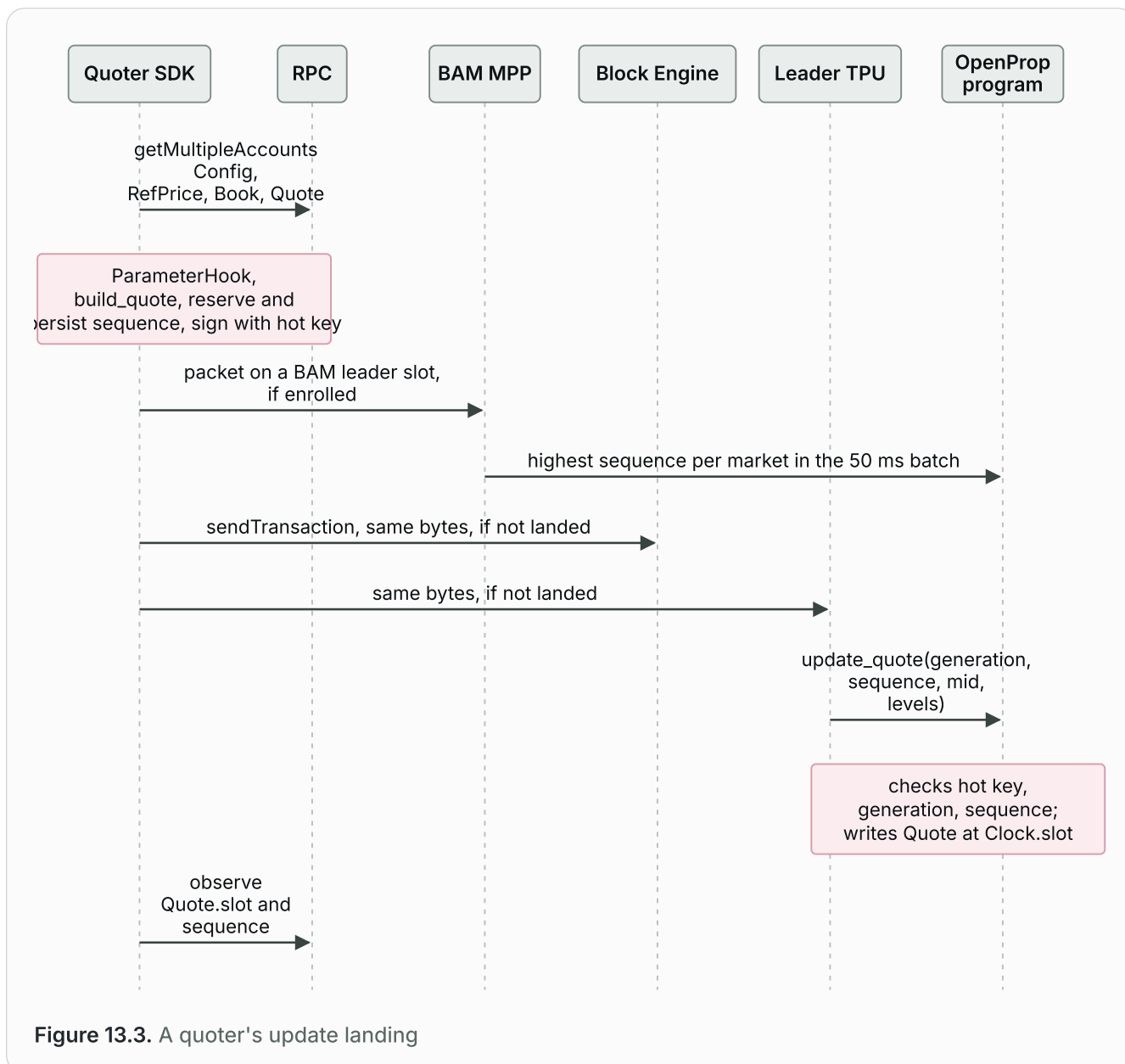


Figure 13.3. A quoter's update landing

13.9 Jupiter Amm adapter

openprop-jupiter-adapter implements the official jupiter-amm-interface Amm trait, one market per Book, labelled OpenProp. Jupiter's account provider supplies Config, RefPrice, Book, Quote and the book's two token accounts. update checks ownership, exact version, kind and length, PDA links, mints, token authority and that custody covers tracked inventory; a failed update clears the old snapshot. quote and every other trait method make no network call.

Quotes come from the same openprop-rules crate the program executes, at the slot of Jupiter's shared clock. The adapter returns no quote for a stale quote or reference, an ended epoch, a halted or paused book or pair, an invalid ladder, a confidence or divergence failure or a breached loss limit, and is_active turns false in the same states. It supports ExactIn only and declines a quote that would leave input unspent at the skew or depth boundary. A fee-mode hint never unlocks retail offsets; only the configured retail signer does. The swap carries ten native account metas, and encode_swap_with_min_output writes the user's minimum into the instruction, so the program rejects a short fill with SLIPPAGE_EXCEEDED. jupiter-amm-test-kit's assert_pool_parity checks the adapter against the real SBF artifact in both directions.

Appendix A. Parameters

The protocol's parameter catalog. A value given here is the one the protocol states; a parameter marked "set at initialization" takes its value from the deployment's parameter file when the protocol accounts are created, and is readable on chain from the `Config` account and through the API's `/v1/parameters` and `/v1/network` reads.

Parameter	Meaning	Value	Unit
<code>epochSlots</code>	Epoch length	432000	slots
<code>checkpointCadenceSlots</code>	Checkpoint cadence	1500	slots (approximate)
<code>quoteLevelsPerSide</code>	Quote levels per side	5	levels
<code>quoteStalenessSlots</code>	Maximum quote age	2	slots
<code>referenceStalenessSlots</code>	Maximum reference age	10	slots
<code>hourlyLossLimitBps</code>	Hourly book loss limit	50	bps of allocation
<code>epochLossLimitBps</code>	Epoch book loss limit	200	bps of allocation
<code>breakerMoveBps</code>	Pair breaker move threshold	300	bps
<code>breakerMoveWindowSlots</code>	Pair breaker move window	150	slots
<code>breakerResumeDelaySlots</code>	Resume after condition clears	50	slots
<code>lpSettlementShareBps</code>	LP settlement share	5000	bps of positive excess above HWM
<code>quoterSettlementShareBps</code>	Quoter settlement share	4000	bps of positive excess above HWM
<code>protocolSettlementShareBps</code>	Protocol settlement share	1000	bps of positive excess above HWM
<code>tier0Leverage</code>	Tier 0 allocation leverage	1	times USDC bond
<code>tier1Leverage</code>	Tier 1 allocation leverage	5	times USDC bond
<code>tier2Leverage</code>	Tier 2 allocation leverage	15	times USDC bond
<code>tier1MinimumEpochs</code>	Tier 1 minimum history	14	epochs
<code>tier1PositiveScoreRequired</code>	Tier 1 positive score required		boolean
<code>tier1NoTripsWindowEpochs</code>	Tier 1 no-limit-trip window	7	epochs
<code>tier2MinimumEpochs</code>	Tier 2 minimum history	45	epochs
<code>tier2GuardianReviewRequired</code>	Tier 2 guardian review in v0		boolean
<code>tierDowngradeEpochLossTrips</code>	Epoch loss trips before tier downgrade	2	trips
<code>tierDowngradeLookbackEpochs</code>	Tier downgrade lookback	7	epochs

Parameter	Meaning	Value	Unit
explorationAllocationBps	Tier 0 exploration pool	1000	bps of deployable allocation
maxVaultPerQuoterBps	Maximum vault allocation per quoter	3500	bps of vault
maxAllocationChangeBpsPerEpoch	Maximum allocation change per epoch	2500	bps change per epoch
scoreEwmaHalfLifeDays	Score EWMA half-life	7	days
withdrawServiceEpochs	Withdrawal first service	1	epochs after request
withdrawMaximumEpochs	Maximum withdrawal wait	2	epochs
unbondMinimumEpochs	Minimum unbond delay	2	epochs
excessBondWithdrawDelayEpochs	Excess bond withdrawal delay	2	epochs
loosenTimeLockEpochs	Limit loosening timelock	3	epochs
tightenTimeLockEpochs	Limit tightening timelock	0	epochs (immediate)
maxQuoteAuthorities	Maximum hot quote authorities	3	keys
vaultCapMicros	V0 vault cap	2000000000000	USD micros
updateQuoteComputeTarget	Update quote compute target	200	CU target
swapComputeTarget	Swap compute target	40000	CU estimate
launchProtocolFeeWaiverEpochs	Proposed launch protocol fee waiver	90	epochs (launch offer)
launchFeeWaiverEnabled	Whether the proposed launch fee waiver is active	set at initialization	boolean
bandWidthBps	Reference band width	set at initialization	bps
confidenceGateBps	Reference confidence gate	set at initialization	bps
oracleDivergenceLimitBps	Maximum source divergence	set at initialization	bps
skewLimitBps	Inventory skew limit	set at initialization	bps
allocationChangeBasis	Basis of the per-epoch allocation change cap	set at initialization	relative allocation or vault weight
scoreShrinkageN0	Score shrinkage n0	set at initialization	checkpoint samples
scoreStddevFloorE9	Score standard-deviation floor	set at initialization	return times 1000000000

Parameter	Meaning	Value	Unit
scoreMinMeanReturnE9	Minimum scoring mean return	set at initialization	return times 1000000000
minScoredNotionalBps	Minimum traded notional for scoring	set at initialization	bps of allocated value
minimumScoreStddevFloorE9	Immutable minimum score standard-deviation floor	set at initialization	return times 1000000000
minimumScoreMeanReturnE9	Immutable minimum scoring mean return	set at initialization	return times 1000000000
minimumScoredNotionalBps	Immutable minimum eligible notional threshold	set at initialization	bps of allocated value
minimumConfidenceBps	Immutable minimum allowed confidence gate	set at initialization	bps
minimumDivergenceBps	Immutable minimum allowed oracle divergence gate	set at initialization	bps
minimumRefStalenessSlots	Immutable minimum allowed reference staleness limit	set at initialization	slots
maximumResumeDelaySlots	Immutable maximum breaker resume delay	set at initialization	slots
minimumBreakerMoveBps	Immutable minimum allowed price-move breaker threshold	set at initialization	bps
tier2ScoreThresholdE6	Tier 2 score threshold	set at initialization	score times 1000000
retailOffsetLimitBps	Retail offset limits	set at initialization	bps
maxLevelSizeBaseLamports	Maximum ladder level size	set at initialization	lamports
minimumBondMicros	Minimum registration bond	set at initialization	USDC micros
hourlyLossWindowSlots	Hourly loss slot window	set at initialization	slots
scoreEwmaHalfLifeSlots	Score EWMA half-life in slots	set at initialization	slots
withinSlotSpreadWideningLimitBps	Within-slot spread widening cap	set at initialization	bps
addressScreeningProvider	Address screening provider	set at initialization	provider identifier
referencePriceRounding	Two-source midpoint and price rounding	set at initialization	rounding policy
settlementRounding	Settlement and share rounding	set at initialization	rounding policy

Parameter	Meaning	Value	Unit
emptyVaultInitialSharePriceMicros	Initial vault share price	set at initialization	USD micros
profitLockEpochs	Bond profit lock delay	set at initialization	epochs
virtualSharesAtomic	Virtual shares preventing first-deposit inflation	set at initialization	share atoms
virtualQuoteMicros	Virtual quote assets paired with virtual shares	set at initialization	USDC micros
pythSlotAttestor	Signer binding signed Pyth payloads to source slots	set at initialization	Solana address
guardianHaltSigner	Optional automated halt-only delegate	set at initialization	Solana address or disabled
settlementFallbackPolicy	Oracle-unavailable settlement policy	set at initialization	disabled or separately specified policy
evictionPolicy	Inactive quoter eviction policy	set at initialization	disabled or separately specified policy
jupiterRetailSigner	Recognized Jupiter retail signer	set at initialization	Solana address
oracleFeedIds	Pyth Pro and Switchboard feed identifiers	set at initialization	feed identifiers

Appendix B. Instructions

Every instruction of the program at wire version 6. The first byte of instruction data is the wire version and the second the tag below; the arguments follow as little-endian integers (Chapter 12).

Instruction	Tag
update_quote	0
swap_exact_in	1
post_ref_price	2
checkpoint_book	3
settle_epoch	4
deposit	5
request_withdraw	6
claim	7
register_quoter	8
add_bond	9
request_unbond	10
set_quote_authority	11
pause_book	12
halt_pair	13
set_params	14
claim_unbond	15
initialize_config	16
initialize_pair	17
review_tier	18
resume_book	19
resume_pair	20
rotate_identities	21
deregister_quoter	22
buyback_begin	23
buyback_burn	24
buyback_finish	25

Appendix C. Account layouts

The byte layout of every program account at account version 5. Every account begins with its version byte and its kind byte, and has an exact length; integers are little-endian, public keys are 32 bytes.

Config · kind 0 · 1728 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
admin	8	pk	32
guardian	40	pk	32
treasuryToken	72	pk	32
baseMint	104	pk	32
quoteMint	136	pk	32
pythSigner	168	pk	32
switchboardSigner	200	pk	32
switchboardFeed	232	pk	32
pythFeedId	264	u64	8
retailSigner	272	pk	32
epochSlots	304	u64	8
checkpointSlots	312	u64	8
hourlyWindowSlots	320	u64	8
scoreHalfLifeSlots	328	u64	8
scoreShrinkageN0	336	u64	8
tier2ScoreE6	344	u64	8
bandBps	352	u64	8
confidenceBps	360	u64	8
divergenceBps	368	u64	8

Field	Offset	Type	Bytes
skewBps	376	u64	8
retailDeltaLimitBps	384	u64	8
maxLevelBaseAtoms	392	u64	8
minBondQuoteAtoms	400	u64	8
vaultCapQuoteAtoms	408	u64	8
quoteStalenessSlots	416	u64	8
refStalenessSlots	424	u64	8
breakerMoveBps	432	u64	8
breakerWindowSlots	440	u64	8
resumeDelaySlots	448	u64	8
unbondEpochs	456	u64	8
withdrawEpochs	464	u64	8
timelockEpochs	472	u64	8
tier1Epochs	480	u64	8
tier2Epochs	488	u64	8
maxBooks	496	u64	8
oracleTimestampToleranceUs	504	u64	8
initialSharePrice	512	u64	8
allocationChangeBasis	520	u64	8
hourlyLossBps	528	u64	8
epochLossBps	536	u64	8
lpShareBps	544	u64	8
quoterShareBps	552	u64	8
tier0Leverage	560	u64	8
tier1Leverage	568	u64	8
tier2Leverage	576	u64	8
explorationBps	584	u64	8
allocationChangeCapBps	592	u64	8
quoterAllocationCapBps	600	u64	8
downgradeTrips	608	u64	8

Field	Offset	Type	Bytes
downgradeWindowEpochs	616	u64	8
profitLockEpochs	624	u64	8
virtualShares	632	u64	8
virtualQuoteAtoms	640	u64	8
oracleSlotAttestationPolicy	648	u64	8
settlementFallbackPolicy	656	u64	8
evictionPolicy	664	u64	8
pendingActivationSlot	672	u64	8
pendingParams	680	bytes368	368
pythSlotAttestor	1048	pk	32
guardianHaltSigner	1080	pk	32
identityActivationSlot	1112	u64	8
pendingIdentities	1120	bytes296	296
scoreStddevFloorE9	1416	u64	8
scoreMinMeanReturnE9	1424	u64	8
minScoredNotionalBps	1432	u64	8
pendingScoreParams	1440	bytes24	24
minimumScoreStddevFloorE9	1464	u64	8
minimumScoreMeanReturnE9	1472	u64	8
minimumScoredNotionalBps	1480	u64	8
minimumConfidenceBps	1488	u64	8
minimumDivergenceBps	1496	u64	8
minimumRefStalenessSlots	1504	u64	8
maximumResumeDelaySlots	1512	u64	8
minimumBreakerMoveBps	1520	u64	8
pendingActivationEpoch	1528	u64	8
cranker	1536	pk	32
buybackMint	1568	pk	32
buybackTokenProgram	1600	pk	32
pendingBuybackIdentities	1632	bytes96	96

Pair · kind 1 · 168 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
config	8	pk	32
baseMint	40	pk	32
quoteMint	72	pk	32
reference	104	pk	32
vault	136	pk	32

Reference · kind 2 · 3776 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
pair	8	pk	32
mid	40	u64	8
confidence	48	u64	8
sourceSlot	56	u64	8
postedSlot	64	u64	8
pythPrice	72	u64	8
pythConfidence	80	u64	8
pythSlot	88	u64	8
pythTimestampUs	96	u64	8

Field	Offset	Type	Bytes
switchboardPrice	104	u64	8
switchboardConfidence	112	u64	8
switchboardSlot	120	u64	8
switchboardTimestampUs	128	u64	8
trippedSlot	136	u64	8
clearedSlot	144	u64	8
resumeSlot	152	u64	8
lastPythTimestampUs	160	u64	8
lastSwitchboardSlot	168	u64	8
history	176	bytes3600	3600

Vault · kind 3 · 408 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
pair	8	pk	32
baseToken	40	pk	32
quoteToken	72	pk	32
shareMint	104	pk	32
epoch	136	u64	8
epochStartSlot	144	u64	8
lastSettledEpoch	152	u64	8
shareSupply	160	u64	8
reservedBase	168	u64	8
reservedQuote	176	u64	8
pendingShares	184	u64	8

Field	Offset	Type	Bytes
allocatedBase	192	u64	8
allocatedQuote	200	u64	8
bookCount	208	u64	8
lastPrice	216	u64	8
lastNav	224	u64	8
books	232	bytes160	160
deferredQuoterFees	392	u64	8
deferredProtocolFees	400	u64	8

Quoter · kind 4 · 1304 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
owner	8	pk	32
authorities	40	bytes96	96
bondToken	136	pk	32
authorityCount	168	u8	1
tier	169	u8	1
reviewed	170	u8	1
padding	171	bytes5	5
bond	176	u64	8
pendingUnbond	184	u64	8
lockedProfit	192	u64	8
profitUnlockEpoch	200	u64	8
completedEpochs	208	u64	8
scoreE6	216	i64	8

Field	Offset	Type	Bytes
limitTripBitmap	224	u64	8
epochLossTripBitmap	232	u64	8
lastTripEpoch	240	u64	8
book	248	pk	32
profitTranches	280	bytes1024	1024

Book · kind 5 · 576 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
pair	8	pk	32
quoter	40	pk	32
baseToken	72	pk	32
quoteToken	104	pk	32
startBase	136	u64	8
startQuote	144	u64	8
allocation	152	u64	8
epoch	160	u64	8
epochExcess	168	i128	16
cumulativeExcess	184	i128	16
highWaterMark	200	i128	16
hourlyStartSlot	216	u64	8
hourlyLoss	224	u64	8
epochLoss	232	u64	8
lastCheckpointSlot	240	u64	8
checkpointExcess	248	i128	16

Field	Offset	Type	Bytes
ewmaMean	264	i128	16
ewmaSquare	280	u128	16
checkpointCount	296	u64	8
scoreE6	304	i64	8
lastSettledEpoch	312	u64	8
stateChangedSlot	320	u64	8
lastFillSlot	328	u64	8
lastFillSequence	336	u64	8
lastFillInput	344	u64	8
lastFillOutput	352	u64	8
lastFillSide	360	u64	8
epochEndSlot	368	u64	8
hourlyStartExcess	376	i128	16
trackedBase	392	u64	8
trackedQuote	400	u64	8
epochFillCount	408	u64	8
checkpointFillCount	416	u64	8
deferredQuoterFees	424	u64	8
deferredProtocolFees	432	u64	8
epochFillNotional	440	u64	8
checkpointFillNotional	448	u64	8
lastScoreSlot	456	u64	8
owner	464	pk	32
scoreBaseDelta	496	i128	16
scoreQuoteDelta	512	i128	16
scoreBaselineExcess	528	i128	16
scoreResidual	544	i128	16
epochScoredFillCount	560	u64	8
checkpointScoredFillCount	568	u64	8

Quote · kind 6 · 304 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
book	8	pk	32
sequence	40	u64	8
slot	48	u64	8
mid	56	u64	8
retailDeltaBps	64	u32	4
padding	68	bytes4	4
levels	72	bytes120	120
cachedAuthorityCount	192	u8	1
cachePadding	193	bytes7	7
cachedAuthorities	200	bytes96	96
generation	296	u64	8

Withdraw · kind 7 · 144 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
owner	8	pk	32
pair	40	pk	32
epoch	72	u64	8

Field	Offset	Type	Bytes
shares	80	u64	8
minBase	88	u64	8
minQuote	96	u64	8
baseClaim	104	u64	8
quoteClaim	112	u64	8
readyEpoch	120	u64	8
claimed	128	u64	8
fulfilled	136	u64	8

Unbond · kind 8 · 136 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
owner	8	pk	32
quoter	40	pk	32
requestId	72	u64	8
amount	80	u64	8
epoch	88	u64	8
readyEpoch	96	u64	8
claimed	104	u64	8
remaining	112	u64	8
paid	120	u64	8
haircut	128	u64	8

WithdrawBatch · kind 9 · 104 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
vault	8	pk	32
epoch	40	u64	8
shares	48	u64	8
baseReserved	56	u64	8
quoteReserved	64	u64	8
readyEpoch	72	u64	8
claimedShares	80	u64	8
claimedBase	88	u64	8
claimedQuote	96	u64	8

Receipt · kind 10 · 320 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
owner	8	pk	32
requestId	40	u64	8
tag	48	u64	8
payloadLength	56	u64	8
payload	64	bytes256	256

Protocol · kind 11 · 272 bytes

Field	Offset	Type	Bytes
version	0	u8	1
kind	1	u8	1
bump	2	u8	1
flags	3	u8	1
reason	4	u8	1
reserved	5	bytes3	3
config	8	pk	32
cranker	40	pk	32
crankerUsdcToken	72	pk	32
crankerBuybackToken	104	pk	32
buybackMint	136	pk	32
buybackTokenProgram	168	pk	32
beginIndex	200	u64	8
finishIndex	208	u64	8
usdcAmount	216	u64	8
minBurn	224	u64	8
supplyBefore	232	u64	8
usdcBefore	240	u64	8
tokenBefore	248	u64	8
protocolUsdcBefore	256	u64	8
slot	264	u64	8

Appendix D. Events

The program emits fixed-layout events: a header (version, kind, subject, epoch, slot, values) followed by values of type `i128`, 210 bytes for up to 10 values and longer for a kind that carries more. The indexer reads them to build checkpoint, settlement, incident and buyback histories.

Event	Kind	Bytes	Values, in order
checkpoint	1	210	excess, value, hold, scoreE6, referencePrice, hourlyLoss, epochLoss
settlement	2	242	excess, lpShare, quoterShare, protocolShare, bondCharge, lpLoss, newAllocation, oldAllocation, highWaterMark, referencePrice, targetWeightBps, cancelledDeferredQuoter
incident	3	210	reason, action, referencePrice
unbondClaim	4	210	requested, paid, haircut, remaining
buyback	5	210	usdcSpent, tokensBurned

Appendix E. Program errors

The program's custom errors. A failed transaction carries the code in its first Program ... failed log line; the API and the contract client return the name and message below.

Code	Name	Message
6000	INVALID_INSTRUCTION	The instruction data is invalid.
6001	INVALID_ACCOUNT	An account does not match the expected address or owner.
6002	UNAUTHORIZED	The required authority did not sign.
6003	UNDECIDED_PARAMETER	A required protocol parameter is unset.
6004	QUOTE_STALE	The quote is older than the allowed slot age.
6005	REF_STALE	The reference price is older than the allowed slot age.
6006	MID_OUTSIDE_BAND	The quote mid is outside the reference price band.
6007	CONFIDENCE_TOO_WIDE	Reference price confidence exceeds its configured limit.
6008	PAIR_HALTED	The pair circuit breaker is active.
6009	BOOK_HALTED	The book is halted.
6010	BOOK_PAUSED	The quoter has paused this book.
6011	SKEW_LIMIT	The fill would exceed the book inventory skew limit.
6012	HOURLY_LOSS_LIMIT	The book hourly loss limit has been reached.
6013	EPOCH_LOSS_LIMIT	The book epoch loss limit has been reached.
6014	STALE_SEQUENCE	The quote sequence number must increase.
6015	SLIPPAGE_EXCEEDED	The output is below the signed minimum.
6016	INSUFFICIENT_LIQUIDITY	The book has insufficient executable liquidity.
6017	VAULT_CAP_EXCEEDED	The deposit would exceed the vault cap.
6018	NON_PROPORTIONAL_DEPOSIT	The deposit does not satisfy the vault token ratio.
6019	WITHDRAW_NOT_READY	The withdrawal ticket is not claimable yet.
6020	UNBOND_NOT_READY	The unbond request is not claimable yet.
6021	BOND_IN_USE	The requested bond is required by the current allocation.
6022	AUTHORITY_LIMIT	A quoter can have at most three hot quote authorities.
6023	ALREADY_PROCESSED	This request has already been processed.
6024	REPLAY_PAYLOAD_MISMATCH	This request identifier was used with different instruction data.
6025	ARITHMETIC_OVERFLOW	The integer calculation exceeds its supported range.

Code	Name	Message
6026	EPOCH_NOT_READY	The epoch is not ready for this operation.
6027	INVALID_ORACLE	The reference oracle proof or source is invalid.
6028	TIMELOCK_PENDING	The parameter change timelock has not elapsed.
6029	INVALID_TOKEN_ACCOUNT	A token account has an unexpected mint, authority, or token program.
6030	MINIMUM_SHARES	The deposit would mint fewer than the signed minimum shares.
6031	UNBOND_EXCEEDS_AVAILABLE	The unbond request exceeds the available bond.
6032	INVALID_AUTHORITY_CHANGE	The hot authority replacement is invalid.
6033	INVALID_AMOUNT	The token amount must be a valid nonzero integer.
6034	QUOTE_LADDER_INVALID	The quote ladder levels or offsets are invalid.
6035	REFERENCE_DIVERGENCE	The reference sources diverge beyond the configured limit.
6036	TIER_REVIEW_REQUIRED	Tier two requires the configured guardian review.
6037	FEE_LIABILITY_SHORTFALL	Deferred fee liabilities exceed available assets; deposits are blocked until the shortfall is resolved.
6038	REGISTRY_FULL	The vault has no free quoter seats. A quoter must deregister before another can register.
6039	BUYBACK_INVALID	The buyback instruction, configuration, or account state is invalid.
6040	BUYBACK_INCOMPLETE	The buyback did not spend and burn within its signed limits.

Glossary

Terms are listed alphabetically. Program names of accounts, instructions, fields and parameters are given in code. Values marked "v0" are the starting parameter set of the first deployment.

Term	Definition
\$OPENPROP	The protocol's token: 1,000,000,000 fixed supply, sold in one public launch on equal terms. The protocol's 10% share of settlement buys it on the market and burns it.
Admin	The multisig key stored in <code>Config.admin</code> . It initializes the deployment, proposes and applies parameter changes with <code>set_params</code> , resumes a manually halted pair with <code>resume_pair</code> and rotates identities with <code>rotate_identities</code> ; no admin instruction moves vault, book or bond assets.
Allocation (<code>Book.allocation</code>)	The value in USDC micros, at the settlement reference price, of the SOL and USDC basket a book receives from the vault at <code>settle_epoch</code> . It is capped at <code>tier leverage × bond</code> , at <code>quoterAllocationCapBps</code> of vault NAV (35% in v0) and by a per-epoch change of at most <code>allocationChangeCapBps</code> (25% in v0).
BAM Maker Priority Plugin (MPP)	Jito's maker plugin, which collects quote updates on BAM leader slots in batches of about 50 ms, deduplicates them by sequence number and places the latest update first. The quoter SDK tries it first when the quoter is enrolled, then the Block Engine and the TPU, each through a client the quoter supplies.
Band (<code>bandBps</code>)	The maximum distance, in bps of the reference price, between a quote's mid and the reference price at which the quote can fill. A swap against a quote outside the band fails with <code>MID_OUTSIDE_BAND</code> .
Basis point (bps)	One hundredth of a percent, 10^{-4} . Ladder offsets, bands, limits and splits are expressed in whole basis points.
Bond	The quoter's USDC, held in the program-owned token account at <code>["bond", quoter]</code> and recorded in <code>Quoter.bond</code> . Negative epoch excess is charged to it before any loss reaches LPs, and the quoter's share of positive excess is credited to it.
Book (Book)	One quoter's segregated market for the pair: a 576-byte account at <code>["book", quoter, pair]</code> with its own SOL and USDC token accounts. It records allocation, starting and tracked inventory, excess, high-water mark, loss windows, scoring sums and state.
Book state	Byte 3 of Book: 0 live, 1 halted by a loss limit at a checkpoint, 2 paused by the cold owner or deregistered. Only a live book fills; settlement clears a loss-limit halt at the epoch boundary.
BpsE4	A basis-point value in fixed point scaled by 10^4 , so 1 bps = 10,000 BpsE4. The API reports fill-versus-reference and 2 s and 30 s markouts in BpsE4.
Buyback	The transaction that spends the protocol USDC: <code>buyback_begin</code> claims it, a Jupiter route swaps it for \$OPENPROP, <code>buyback_burn</code> burns every token bought, and <code>buyback_finish</code> requires the supply to have fallen by at least <code>min_burn</code> . It fails whole otherwise.
Checkpoint (<code>checkpoint_book</code>)	A permissionless instruction that marks one book to the reference price at most once per <code>checkpointSlots</code> (about 1,500 slots in v0). It updates the book's loss windows and scoring moments and halts the book if a loss limit is exceeded.
Circuit breaker	The pair-level halt flag in <code>RefPrice</code> . A reference older than <code>refStalenessSlots</code> , a confidence or divergence failure, or a move above <code>breakerMoveBps</code> within <code>breakerWindowSlots</code> stops every book, which resumes <code>resumeDelaySlots</code> after the condition clears (v0: 10 slots, 300 bps in 150 slots, 50 slots).

Term	Definition
Cold owner	The wallet that registered a quoter, stored in <code>Quoter.owner</code> . It manages the bond, the hot authority list and the book's pause state; it cannot write quotes or withdraw book or vault assets.
Confidence gate (<code>confidenceBps</code>)	The maximum reference confidence, in bps of the reference price, at which books may fill. The stored confidence is the larger of the Pyth Pro confidence and half the disagreement between the two sources.
Config (<code>Config</code>)	The deployment's singleton 1,728-byte account at <code>["config"]</code> . It holds the identities, the two mints, the active and pending parameter sets and the immutable safety envelope.
Cranker	Any fee payer that runs <code>post_ref_price</code> , <code>checkpoint_book</code> and <code>settle_epoch</code> . These instructions compute their results from accounts, the clock and verified proofs and take no decision from their caller.
Divergence gate (<code>divergenceBps</code>)	The maximum disagreement between the Pyth Pro and Switchboard Surge prices, in bps of the reference price, at which books may fill. A wider disagreement fails swaps with <code>REFERENCE_DIVERGENCE</code> and trips the pair breaker.
E6	Fixed-point scaling by 10^6 . Scores (<code>scoreE6</code>) and the tier-2 score threshold (<code>tier2ScoreE6</code>) are stored in E6.
E9	Fixed-point scaling by 10^9 . Scoring returns, normalized to the allocation that earned them, and the score policy values <code>scoreStddevFloorE9</code> and <code>scoreMinMeanReturnE9</code> are stored in E9.
Epoch	The OpenProp settlement period of <code>epochSlots</code> slots (432,000 in v0, about 24 hours), counted by <code>LpVault.epoch</code> . It is distinct from a Solana validator epoch.
Epoch loss limit (<code>epochLossBps</code>)	The maximum loss of a book against its starting inventory within one epoch, as a fraction of allocation: 200 bps (2%) in v0. Two epoch-loss trips within 7 epochs lower the quoter's tier by one.
Excess over hold	The value of a book's tracked inventory minus the value of its starting inventory, both marked at the current reference price. It measures a quoter's spread capture and inventory management independently of the direction of SOL.
Exploration pool (<code>explorationBps</code>)	The share of allocation weight, 1,000 bps (10%) in v0, divided equally among live tier-0 books. The remaining weight is divided in proportion to positive scores.
Freshness (<code>quoteStalenessSlots</code>)	The maximum quote age at which a quote can fill, where age is <code>Clock.slot</code> minus <code>Quote.slot</code> : 2 slots in v0. A swap against an older quote fails with <code>QUOTE_STALE</code> .
Generation (<code>Quote.generation</code>)	A counter incremented by every change of a book's hot authority list. Every <code>update_quote</code> must carry the current generation, so a packet signed before a change cannot execute after it.
Guardian	The multisig in <code>Config.guardian</code> that can halt the pair with <code>halt_pair</code> and approve tier-2 reviews with <code>review_tier</code> . An optional <code>guardianHaltSigner</code> key can call <code>halt_pair</code> only; neither can resume the pair, change parameters or move funds.
High-water mark (<code>Book.highWaterMark</code>)	The highest cumulative excess a book has reached. Only positive excess above it is split between LPs, quoter and protocol at settlement.
Hot authority	One of up to three keys cached in <code>Quote</code> that may sign <code>update_quote</code> for that book. It can write only the <code>Quote</code> account and is installed only with its own co-signature and the cold owner's.

Term	Definition
Hourly loss limit (hourlyLossBps)	The maximum fall in a book's excess within one window of hourlyWindowSlots slots, as a fraction of allocation: 50 bps (0.5%) in v0. A swap that would exceed it fails with HOURLY_LOSS_LIMIT.
Lamport	The smallest unit of SOL, 10^{-9} SOL. All SOL amounts and ladder level sizes are in lamports.
Level	One rung of a quote ladder: an offset from the mid in whole bps (u32) and a size in lamports (u64). A quote has five bid levels and five ask levels.
Leverage	The multiple of bond, net of pending unbond, up to which a quoter's allocation may grow: 1× at tier 0, 5× at tier 1 and 15× at tier 2 in v0 (tier0Leverage, tier1Leverage, tier2Leverage).
LP	A liquidity provider: a wallet that deposits SOL and USDC into the vault with deposit, holds OPLP shares, and exits with request_withdraw and claim.
LpVault (LpVault)	The pair's 408-byte vault account at ["vault", pair]. It owns the free SOL and USDC token accounts and the share mint and records the epoch, share supply, reserved withdrawals, deferred fees and the registry of books.
Micro	One millionth of a USDC, 10^{-6} USDC. USDC amounts and values are in micros, and prices are in micros per SOL.
Mid (Quote.mid)	The centre of a quoter's ladder, in micros per SOL. Level prices are offsets from it, and it must lie within the band around the reference price for the quote to fill.
NAV	The vault's net asset value: free SOL and USDC across the vault and all books, marked at the reference price, plus pending first-loss coverage, minus accrued and deferred fees. Deposits and redemptions convert shares at NAV.
OperationReceipt	An account at ["receipt", owner, requestId] that stores a monetary request's original payload. A retry with the same request ID and payload is a no-op, and a changed payload fails with REPLAY_PAYLOAD_MISMATCH.
OPLP	The vault's LP share token: a classic SPL mint with 6 decimals at ["shares", pair], whose mint authority is the LpVault.
PairConfig (PairConfig)	The 168-byte account at ["pair", baseMint, quoteMint] that links the deployment's single pair to its Config, RefPrice and LpVault.
Protocol (Protocol)	The 272-byte account at ["protocol"]. Its PDA owns the protocol USDC account that receives the protocol share, and it holds the guard state of an open buyback.
Quote (Quote)	The 304-byte account at ["quote", book] that holds a book's current ladder, sequence, landing slot, retail delta, generation and cached hot authorities. Only update_quote and set_quote_authority write it.
Quoter (Quoter)	The 1,304-byte account at ["quoter", owner] that records a quoter's cold owner, hot authorities, bond, tier, review flag, limit-trip history and profit-lock tranches. The term also names the role: a bonded party that prices one book.
Reference price (RefPrice)	The pair's oracle account at ["ref", pair], written by post_ref_price. Its mid is the integer midpoint of verified Pyth Pro and Switchboard Surge prices, and it carries confidence, source slots and breaker state.
Registry	The array of up to five book addresses in LpVault.books, in canonical order, with bookCount and the configured limit maxBooks. deposit and settle_epoch must receive every registered book in that order, and registration at capacity fails with REGISTRY_FULL.

Term	Definition
Retail delta (retailDeltaBps)	The number of bps by which a quote's offsets are reduced, saturating at zero, on a swap co-signed by <code>Config.retailSigner</code> . It is chosen per update and bounded by <code>retailDeltaLimitBps</code> ; a zero <code>retailSigner</code> disables the retail class.
Score (scoreE6)	A book's risk-adjusted performance: the EWMA mean of its checkpoint excess returns above a hurdle, divided by the larger of their standard deviation and a floor, multiplied by the shrinkage factor $n/(n + n_0)$. Positive scores set the non-exploration share of allocation weight.
Sequence (Quote.sequence)	The update counter carried by every <code>update_quote</code> . An update must carry a sequence strictly greater than the stored one, or it fails with <code>STALE_SEQUENCE</code> ; authority changes reset it to zero.
set_quote_authority	The cold-owner instruction that replaces a book's list of up to three hot authorities. Every newly added key co-signs; every actual change increments the generation and clears the live quote.
Settlement (settle_epoch)	The permissionless end-of-epoch instruction that recalls every book, charges negative excess to bonds, splits positive excess above the high-water mark (50% LPs, 40% quoter, 10% protocol in v0), reserves withdrawals and reallocates the vault.
Skew (skewBps)	The inventory imbalance $ V_B - Q /(V_B + Q)$, where V_B is the SOL inventory's value at the reference price and Q the USDC inventory. A fill may not take a book's skew above <code>skewBps</code> , and the swap fills partially at that boundary.
Slot	Solana's unit of block time, 200 ms on mainnet-beta since epoch 1053. Every <code>OpenProp</code> age, window, cadence and <code>timelock</code> is counted in <code>Clock.slot</code> .
swap_exact_in	The taker instruction that sells SOL (side 0) or buys SOL (side 1) for an exact input against one book's ladder, after every check passes, and transfers the consumed input and the output. It fails atomically when the output is below the signed minimum.
Taker	The wallet, or the aggregator router acting for it, that calls <code>swap_exact_in</code> with a side, an exact input and a signed minimum output.
Tier	A quoter's level, 0, 1 or 2, which sets its allocation leverage. Tier 1 requires 14 completed epochs, a positive score and no limit trips in the last 7; tier 2 requires 45 epochs, a score above <code>tier2ScoreE6</code> and guardian review (v0).
Timelock (timeLockEpochs)	The delay before a loosening parameter change can apply: <code>timeLockEpochs</code> × <code>epochSlots</code> slots (3 epochs in v0) and a vault epoch at least the withdrawal queue plus one beyond the proposal. Tightening applies immediately.
Tracked inventory (trackedBase, trackedQuote)	The SOL and USDC a book is accounted to hold, changed only by swaps and settlement. Its token accounts must cover it, and tokens sent to them by anyone else are excluded from it.
UnbondTicket	The account at <code>["unbond", quoter, requestId]</code> that records one bond withdrawal request. It becomes claimable after at least <code>unbondEpochs</code> epochs (2 in v0) and records the amounts paid, haircut and remaining.
update_quote	The hot-authority instruction (tag 0) that replaces a book's mid, ten levels and retail delta in its <code>Quote</code> account. It checks the signer, generation and sequence, records <code>Clock.slot</code> , and measured 178 CU on the hinted path.
Virtual shares (virtualShares, virtualQuoteAtoms)	Fixed offsets added to share supply and to NAV in every share conversion. They protect initial issuance against donations to the vault.

Term	Definition
Wire version	The version byte that opens every instruction ([6, tag]) and every account. This paper describes wire and account version 6.
WithdrawTicket	The account at ["wd", owner, epoch] that records an LP's withdrawal request. The next settlement reserves its basket, and claim pays it once its ready epoch is reached.