Lull

Whitepaper

Lull: sealed market-on-open auctions for xStocks

How a sealed market-on-open auction for xStocks works on Solana: the lull calendar, sealed orders, the reveal, P_{open}, the cross, what is live today, and the $LULL token design.

Version 1 · September 2026 · 27,440 words

Contents

1

Abstract and notice

Abstract

Lull is a sealed market-on-open auction for xStocks, the tokenized equities that trade on Solana. An xStock trades around the clock, but the stock underneath it trades only in the US regular session: 32.5 of the week's 168 hours for Apple. The rest of the week is a lull. During it, xStock pools are thin, their prices drift from the last official price, and anyone who can see a queue of orders for the next open can trade ahead of it.

Brokerages solve this with opening auctions and two order types, market-on-open (MOO) and limit-on-open (LOO). Lull gives a self-custody wallet the same order types. During the lull a user commits to an order by posting a SHA-256 hash of it and a small SOL bond. No tokens move, and nothing about the order is public. In the half hour before the open the user reveals the order, and the program locks its amount. Just after the open the program captures P_{open}, the first fully verified Pyth price of the underlying stock published at or after the open plus a short delay. Every revealed order then crosses once, at that price. Liquidity providers (LPs) absorb the remaining imbalance with sealed spread quotes, cheapest first, each at its own price. Whatever does not fill returns to the owner's balance.

The design rests on four properties:

- **The price is taken, not discovered.** P_{open} is Pyth's aggregate of the primary market's regular session. Trading an xStock pool cannot move it.
- **The queue is sealed until the reveal window.** A commitment binds every order field, a 32-byte random salt, the owner and the lull.
- **Fills do not depend on timing.** The light side fills in full at P_{open}, and the heavy side shares the matched volume pro rata.
- **Accounting is exact.** Amounts are integers in raw token units, the Scaled UI multiplier enters once, and rounding always favors the vault.

The implementation is an Anchor program, a TypeScript SDK that mirrors the program's crossing math line for line, a worker that derives the calendar from Pyth's schedule string and runs the permissionless cranks, a Fastify API over Postgres, and a web app. Five gates pass. They cover the schedule parser against real Pyth schedules, property tests of the crossing invariants, an end-to-end cross on a local validator holding real mainnet accounts, the API, and pricing through the mainnet Pyth receiver and Wormhole programs: an exact P_{open} capture, and the TWAP fallback with real verified updates.

The paper also documents the design of $LULL: a fixed supply of 1,000,000,000 tokens launched on a bonding curve, with no team allocation, presale or vesting. Its planned uses are a protocol fee whose revenue buys $LULL on the open market to be burned, staking for delegated roles, and token-weighted governance. None of these is implemented.

Status at the time of writing

Component	Status
Anchor program	Implemented; all gates pass. **Not deployed on mainnet.** The program id `8xf4NfkWse8YRYghY1femurYsZaBaSEoNgssweT6oGWd` is reserved by its keypair.
SDK, worker, API	Implemented. The worker and API run on Railway in read-only mode against mainnet, serving the calendar, live Pyth prices and lull-drift analytics.
Web app and user docs	Implemented. Write flows run against a local stack.
Exact P_{open} capture and TWAP fallback	Implemented; `check:exact` runs both against the mainnet receiver programs on a local validator and passes.
Encrypted orders (Arcium, Stage 2)	Designed, not built
Protocol fee, $LULL token, staking registry	Designed, not implemented. Nothing in the program references $LULL.
Governance	Designed. A market's authority would today be a single key that the team controls.
Security audit	Not done. It is a prerequisite for real funds.

Every figure in this paper comes from one of four places: the repository's source and documents, the recorded end-to-end run (`docs/e2e-sample-report.json`), the live read-only API (fetched on 2026-09-26, when the server clock read 2026-09-25 23:46 UTC), or a worked example labelled as such. Defaults are the SDK's `DEFAULT_PARAMS` and are marked as defaults.

Reading guide

Chapters 2 to 5 cover the background, the problem and the design goals. Chapters 6 to 12 specify the mechanism. Chapters 13 to 15 cover privacy, security and incentives. Chapters 16 and 17 describe the implementation and the measurements. Chapters 18 and 19 cover governance and the token design, and chapters 20 to 22 the roadmap, the risks and the legal notice. The appendices are reference material. Formulas are in code blocks: `⌈x⌉` rounds up, `⌊x⌋` rounds down, and 1 bps is 0.01%.

Notice

This paper describes software and a token design. It is not an offer to sell, or a solicitation to buy, any security, token or financial instrument, and it is not investment, legal or tax advice. $LULL is designed as a utility and governance token with no claim on revenue, assets or profits. Nothing here promises a price, a return, or that any planned feature will ship. Lull and $LULL are not intended for US persons or for anyone in a jurisdiction where their use is restricted. See chapter 21 for the risks and chapter 22 for the full legal notice.

2

Introduction

A market that never closes, on top of one that does

An xStock is a token on Solana that tracks a listed stock. AAPLx tracks Apple. It can be swapped at any hour. The stock itself trades on US exchanges in the regular session, 09:30 to 16:00 Eastern Time on weekdays, with holidays and early closes: 32.5 hours of a normal week's 168.

For the other 135.5 hours the token keeps trading while its reference market is shut. The price everyone will eventually agree on, the next open, does not exist yet. What exists is a set of pools whose prices move with whatever flow arrives, whose depth is whatever LPs will leave exposed overnight, and whose pending transactions anyone can watch. Lull calls this interval a lull.

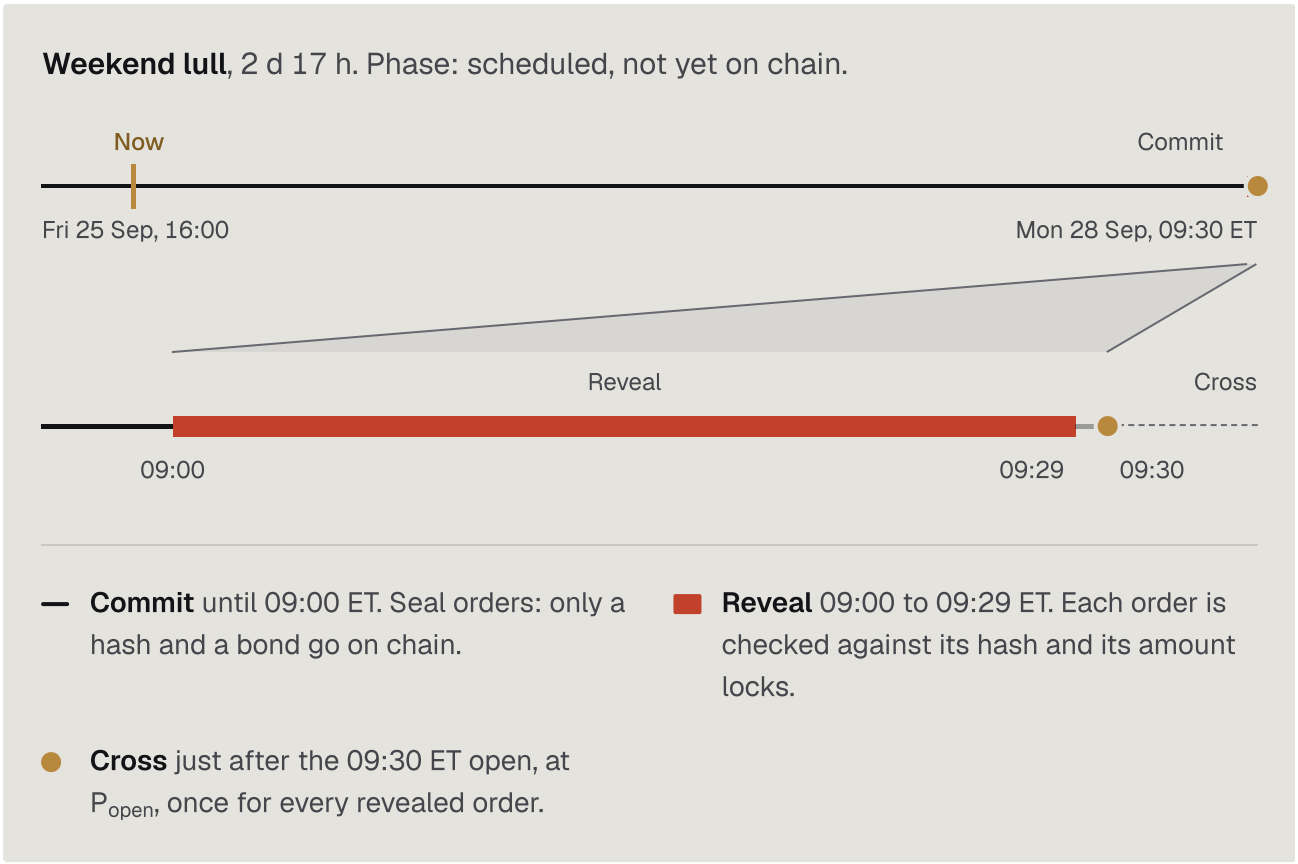
A holder who wants to sell AAPLx on a Saturday can sell into a pool at whatever it offers, or wait until Monday and sell after the open. The first choice pays for the lull in spread and slippage and exposes the order to anyone who sees it coming. The second means watching the clock, and the order is still exposed when it is sent.

Brokerages solved this long ago. A client sends a market-on-open order on Saturday, the broker holds it, and the exchange crosses it at the open with every other opening order at one official price. Nobody outside the broker and the exchange sees the queue. A self-custody wallet has had no equivalent.

What Lull does

Lull is a market-on-open order for a self-custody wallet, with one market per xStock, starting with AAPLx/USDC. Each lull has three steps:

1. **Commit.** During the lull a user seals an order. Only a hash and a small SOL bond go on chain. No tokens move.
2. **Reveal.** From 30 minutes to 1 minute before the open (by default), the user reveals the order. The program checks it against the hash and locks the amount.
3. **Cross.** Just after the open the program captures P_{open}. Buys and sells match at it, and LPs absorb the rest of the heavier side with sealed spread quotes, cheapest first. Unfilled amounts return to the owner's balance.



The current lull of AAPLx/USDC from the Lull API, in New York time. The top bar is the whole lull to scale. The bottom bar enlarges the reveal window and the open.

Every step after the reveal window is permissionless. If no valid price arrives in time, anyone can cancel the lull, and every revealed order is refunded in full.

What is different about it

Exchange opening auctions discover a clearing price from their orders. Lull takes its price from the primary market, through Pyth, and crosses whatever it holds at that price. Three consequences follow:

- **Nothing in Lull can move P_{open}.** A better price would require moving Apple's own opening prints across Pyth's publishers.
- **The imbalance cannot move the price.** At an exchange it moves the clearing price for everyone. At Lull the light side fills at P_{open}, and the heavy side bears the imbalance through pro rata rationing and LP spreads.
- **Reveal timing does not matter.** Revealing first earns nothing, and revealing last only shortens the time an order is public.

The other half of the design is custody and accounting. Users hold funds in a per-market balance account that only the program's rules can change. The API returns unsigned transactions and builds the reveal with every secret field zeroed, so the operator learns an order when everyone else does. Amounts are raw integers, and the xStock's Scaled UI multiplier is read once, at capture.

Contributions

This paper describes a working system, not a proposal. Its technical contributions are:

1. a lull calendar derived from Pyth's schedule string, with time zones, daylight saving time, holidays and early closes, gated against Pyth's own `market_hours`;
2. commitments that bind the owner and the lull, a uniform bond that hides whether a ticket is an order or an LP quote, and a reveal anyone holding the salt can send;
3. a P_{open} capture rule written as a pure state machine, with a publish-time window, a confidence gate, a TWAP fallback, an exactness flag, and a crank path that posts the provably first update itself;
4. a single-price cross with eligibility classes, pro rata allocation and pay-as-quoted LP fills, linear-time and batchable, written in Rust and TypeScript and checked for equality to the raw unit;
5. a unit discipline at the Scaled UI Amount boundary;
6. a measured privacy bound, with the imbalance public for at most about 30.5 minutes by default, and a multi-party-computation design that removes it.

Scope

The program is not deployed on mainnet. The project owner chose to keep it local for now, and deploying it needs the owner's explicit approval. The read-only API and worker run against mainnet and serve real calendar, price and drift data, but no real order has crossed. The $LULL token, its fee, the staking registry and the governance handover are designs (chapters 18 and 19), with their risks and legal position in chapters 21 and 22.

3

Background

Markets that close

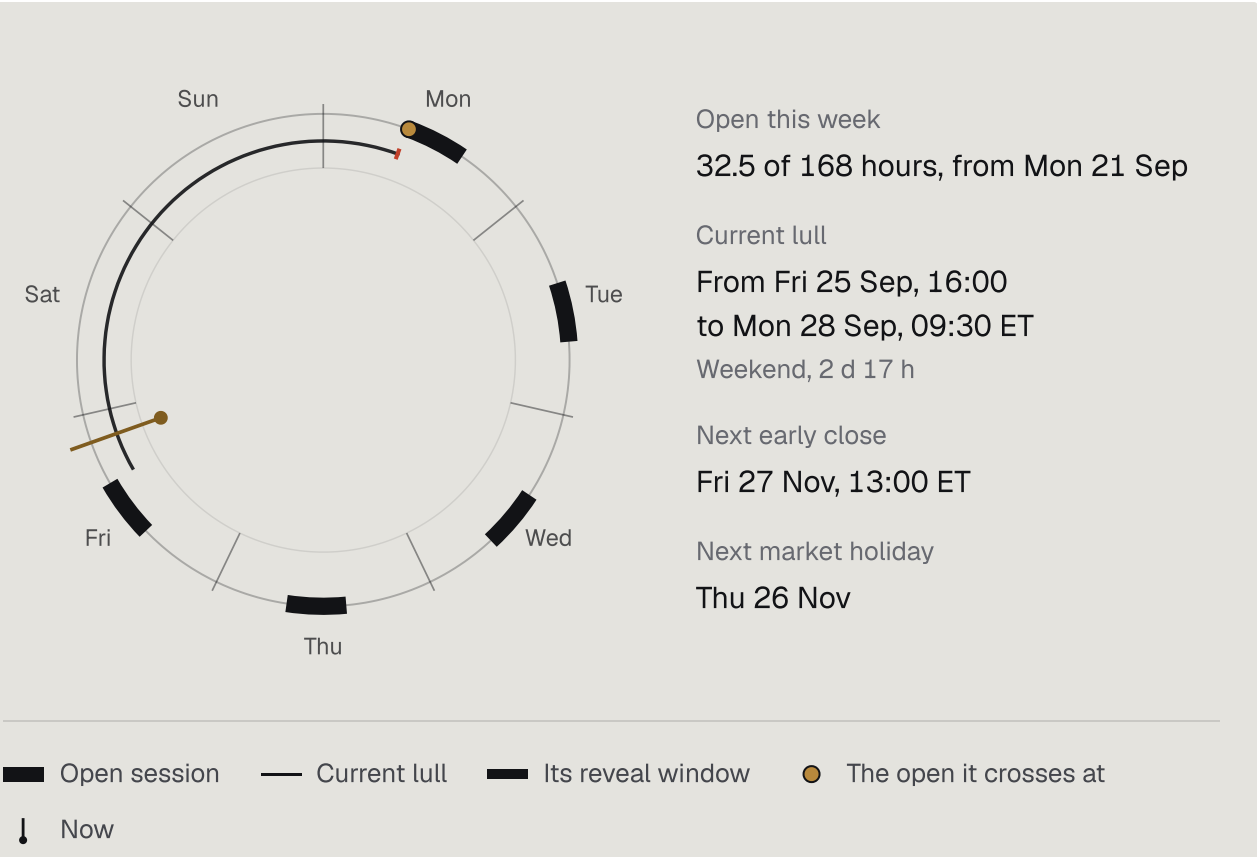
A stock listed on a US exchange has an official trading day: the regular session, 09:30 to 16:00 Eastern Time, Monday to Friday, with market holidays and a few early closes a year. Trading outside the regular session exists, but the day's official reference prices, the open and the close, come from the regular session.

A market that closes accumulates demand while it is shut. Orders that arrive overnight or over a weekend queue until the next open, and the first minutes of the session absorb them at once. That is why exchanges open with a dedicated auction rather than starting the continuous book cold.

The week in numbers

Interval	Length	Share of the week
Regular sessions (5 × 6.5 h)	32.5 h	19.3%
Lulls	135.5 h	80.7%
One overnight lull (16:00 → 09:30 ET)	17.5 h	
A weekend lull (Friday 16:00 → Monday 09:30 ET)	65.5 h	

The calendar gate (`tests/schedule.test.ts`) asserts these figures for AAPL. Holidays and daylight saving time change individual lulls. The weekend on which daylight saving time ends is 66.5 hours long, and the one on which it starts is 64.5 hours. Labor Day 2026 made an 89.5-hour lull, from Friday 4 September 16:00 ET to Tuesday 8 September 09:30 ET.



The lull week of AAPLx/USDC in New York time, built from the Pyth market-hours schedule of Equity.US.AAPL/USD. Monday is at the top and the week runs clockwise.

About four fifths of the week is lull. An asset that trades continuously on top of this schedule spends most of its life priced against a reference that is not being updated.

xStocks

xStocks are tokens on Solana that track listed stocks. The first Lull market uses AAPLx (`XsbEhLAtcf6HdfpFZ5xEMdqW8nfAvcsP5bdudRLJzJp`), a Token-2022 token with 8 decimals. Two of its properties matter here.

- **It trades at all hours**, in on-chain pools, whether or not Apple is trading.
- **The issuer keeps control of the token.** The mint has a permanent delegate, which lets the issuer move tokens out of any account, a pause over all transfers, a freeze authority, and confidential-transfer extensions. These powers reach every AAPLx account, including the vault of any protocol that holds AAPLx for its users. The issuer does not offer xStocks to US persons.

Scaled UI Amount

AAPLx also uses the Token-2022 Scaled UI Amount extension. The mint stores a multiplier, and wallets display a balance as the raw amount times the multiplier, so one displayed AAPLx is one share. The multiplier is not 1, and it changes, for example when dividends are reinvested. In the market data fetched from the live API on 2026-09-26, the mint held a multiplier of `1.0026642075893797` and a new multiplier of `1.0032690125398187`, effective from 2026-08-08

00:30 UTC. Amounts on chain are raw and prices are per share, so every calculation between the two must cross the multiplier exactly once (chapter 12).

Opening auctions and on-open orders

A US exchange opens the regular session with an auction. It collects orders before the open, publishes information about the buy–sell imbalance in the final minutes, and at the open crosses everything it can at one price. Two order types serve it:

- a **market-on-open (MOO)** order trades at whatever the opening price turns out to be;
- a **limit-on-open (LOO)** order trades only if the opening price is at or better than its limit.

Everyone who takes part gets the same price, liquidity is concentrated in one moment, and the result is an official reference price. A brokerage client relies on the broker and the exchange to keep the queue private until the exchange's own imbalance publication.

Batch auctions

A batch auction executes the orders collected over an interval together, at one price, rather than one by one. Budish, Cramton and Shim argue that frequent batch auctions remove the value of being faster than other traders, because orders within the same interval are treated alike. Lull runs one batch per lull. Unlike a usual batch auction, it does not compute a clearing price from its own book. It uses the primary market's price, so the batch decides only who trades and how much.

Commit–reveal

A commitment scheme fixes a value now and discloses it later. The common construction hashes the value with a random salt. The hash is **binding**, because a different value gives a different hash, and **hiding**, because without the salt nobody can test guesses. Commit–reveal is a standard way to run sealed-bid auctions and votes on a public ledger. Its known weakness is the free option: a participant can watch others reveal and then decline to reveal its own. The usual remedy, which Lull uses, is a deposit that is forfeited when a commitment is not revealed.

Pyth

Pyth is an oracle network. Its publishers submit prices, and Pyth aggregates them into a price and a confidence interval per feed. Equity feeds such as `Equity.US.AAPL/USD` report the regular session and carry a market-hours schedule string.

On Solana, Pyth prices are consumed through the Pyth receiver program (`rec5EKMGg6MxZYaMdyBfgwp4d5rB9T1VQH5pJv5LtFJ`). Updates are produced on Pythnet, and the Wormhole guardians sign the Merkle root of each batch. The receiver verifies those signatures and stores an update in a `PriceUpdateV2` account, marked as fully or partially verified. Pyth also sponsors push feeds, accounts at fixed addresses that it keeps updated. Off chain, Hermes serves updates and feed metadata, and Benchmarks serves history. Asked for a timestamp, both return the last update published at or before it.

Lull uses four of these pieces: the schedule string, for the calendar; the push account, for a price that needs no posting; a Hermes update found by timestamp and posted through the receiver, for the exact first price after the open; and Benchmarks, to fill gaps in the drift analytics.

4

The problem

Two prices for one share

During a lull an xStock has two prices, and neither is the one that will matter next. One is the last official price of the underlying, fixed until the next open. The other is the pool price, which moves with whatever flow reaches the pools overnight. Nothing ties the two together except the expectation that they meet again at the open. Whoever trades in a pool during the lull trades against the gap between the pool price at that moment and the next open, and pays the pool's spread and slippage on top.

Thin pools, wide spreads

An LP in an xStock pool overnight carries a price it cannot hedge in the underlying until the open. It protects itself by quoting wider, providing less depth, or leaving. Each of these makes lull trading more expensive for everyone else, and most of all after news that breaks while the market is closed, when many holders want to act at once.

Gaps

The underlying itself moves between sessions, and over a weekend or holiday there is more time for news to accumulate. A holder who waits for the open accepts the gap but avoids the pool. A holder who trades in the pool accepts the gap and the pool's own error.

Information leakage and leaning

On a public chain an order is visible before it executes. A pending swap can be front-run, and a resting on-chain limit order is visible to everyone. In a thin overnight market this is costly. A trader who sees a large sell imbalance building on a Saturday can sell ahead of it into the thin pools, or position on another venue, and buy back after the imbalance has moved the price. This is leaning: taking the other side of predictable flow with advance knowledge of it. Brokerages prevent it by keeping the opening queue private. A self-custody wallet trading in the open cannot.

What the drift series shows

Lull's worker measures the problem. For every lull it records the underlying's prior close (the last Pyth sample at or before the close), the first Pyth price at or after the open, and the xStock's DEX price through the lull, sampled once a minute from Jupiter's price API. The figures below were fetched from `GET /v1/markets/AAPLx-USDC/drift` on 2026-09-26, when the server clock read 2026-09-25 23:46 UTC. Prices are in USD per share.

Lull (ET)	Kind	Prior close	First Pyth price after the open	Open vs close	DEX samples
Tue 22 Sep 16:00 → Wed 23 Sep 09:30	overnight	339.72018	341.03022	+39 bps	0
Wed 23 Sep 16:00 → Thu 24 Sep 09:30	overnight	336.90501	336.71868	−6 bps	0
Thu 24 Sep 16:00 → Fri 25 Sep 09:30	overnight	335.96119	336.02504	+2 bps	934
Fri 25 Sep 16:00 → Mon 28 Sep 09:30	weekend	340.977	in progress		226

The DEX sampler was not yet running during the first two lulls. The API summarizes the three completed lulls as a mean absolute open-versus-close move of 16 bps. The one completed lull with a DEX series shows the shape of the problem:

Night of 24–25 September	USD per share	vs prior close
Prior close (Pyth)	335.96119	
First DEX sample	336.28261	+10 bps
DEX low	334.23207	−51 bps
DEX high	337.13516	+35 bps
Last DEX sample before the open	335.44368	−15 bps
First Pyth price after the open	336.02504	+2 bps

The underlying ended the night 2 bps from its close. Over the same night the xStock's DEX price ranged from 51 bps below the close to 35 bps above it, about 87 bps in all, and its last sample was 17 bps below the price at the open. A holder who sold into the pool at the low received about 51 bps less than the close and 53 bps less than the open, before any spread or slippage. By the time of the fetch, the weekend lull that began on 25 September was 3.8 hours old, and its DEX price had already ranged from 36 bps below the Friday close to 15 bps above it.

These figures have three limits:

- **The sample is small.** One complete night with DEX data illustrates the problem. It is not evidence of a pattern. The series grows by one lull per trading day, and the drift job covers every lull that closed in the previous 30 days.
- **The DEX series is indicative.** Jupiter's `usdPrice` is an aggregated price per share, not an executable quote, and says nothing about depth. Lull labels it as not an oracle, and it never feeds the cross.

- **The "open" column is not P_{open}.** For lulls that did not cross on chain, the drift job uses the first Pyth sample at or after the open itself, not the capture rule of chapter 9. When the worker recorded no sample, it can fill the gap from Pyth Benchmarks: the prior close is the last update at or before the close, and the price after the open is the first update at or after it, found by the same search the crank uses (chapter 9).

What a solution needs

1. **Execution at the official price:** an order placed in the lull fills at the underlying's open, not at a pool price from some moment in the night.
2. **A sealed queue:** nobody, the operator included, sees an order's side, size or limit until shortly before the open.
3. **Absorption of the imbalance:** someone takes the difference between buys and sells, at a known and bounded price.
4. **Self-custody:** keys and funds stay with the user and the program's rules.
5. **No worse than not trading:** without a trustworthy price at the open, nothing trades and everything is refunded.

5

Design goals and non-goals

Goals

Each goal maps to a mechanism and to a gate that checks it.

Goal	Mechanism	Checked by
Execute at the official open	P_{open} is a fully verified Pyth price of the regular session, published in `[open + delay, open + delay + lag]`	`pricing.rs` tests, `check:e2e`, `check:exact`
Keep the queue sealed	Commit–reveal with a client-side salt; the API never receives order fields before they are on chain	`check:api` asserts the reveal template's secret bytes are zero
Make timing irrelevant	One price for all, and pro rata on the heavy side	`check:cross`
Absorb the imbalance at a bounded price	Sealed LP quotes under a spread cap, cheapest first, pay-as-quoted	`check:cross`, `check:e2e`
Never create a token	Integer math in raw units, with every rounding step in the vault's favor	`check:cross`; vault dust in `check:e2e`
Respect every limit	Exact integer comparison with the Pyth price, and eligibility classes	`check:cross`
Self-custody	Balances changed only by program rules; unsigned transactions; no user keys in the backend	program constraints, `check:api`
Liveness without trust	Every step after the reveal window is permissionless; cancellation refunds everything	`check:e2e` (the worker's crank drives the cross)
A correct calendar	Pyth's schedule string, with time zones, DST, holidays and early closes	`check:schedule` against Hermes' `market_hours`
Verifiable outcomes	Price, source, exactness, multiplier, `px`, class totals and every fill are stored on chain or emitted as events	program state and events

Three of these goals need a word of explanation.

- **Execution at the open.** The capture rule excludes anything published before the open plus a delay, and it can be strict enough that no valid price exists. In that case the design prefers not trading to trading at a worse reference.
- **A sealed queue.** Stage 1 hides the queue from other traders and from the operator until the reveal window. After that, reveals are public, and the imbalance is readable until the cross. Lull bounds and measures that interval and specifies a Stage 2 design that removes it (chapter 13).
- **Exactness.** The same math exists once in Rust and once in TypeScript, and the end-to-end gate requires every on-chain fill to equal the TypeScript prediction to the raw unit.

Non-goals

- **Price discovery.** Lull takes P_{open} from the primary market. If Pyth's price is wrong in a way that passes every check, Lull crosses at it.
- **Trading during the lull.** Lull defers execution. It does not quote prices overnight.
- **Full pre-trade privacy in Stage 1.** The imbalance is public for at most about 30.5 minutes with default parameters.
- **Leverage, shorting and margin.** A buy spends USDC the user holds, and a sell delivers xStock the user holds.
- **Netting across markets.** Each market crosses on its own.
- **Removing issuer powers.** The xStock issuer's controls apply to Lull's vault as to any other account.
- **Verifying the calendar on chain.** The program cannot read Pyth's schedule. The market authority publishes each lull's times, and the program enforces their ordering.
- **Serving US persons.** Lull follows the xStocks issuer's eligibility.

Constraints

- **Transaction size.** A Solana transaction is limited to 1,232 bytes, and every account it touches must be listed. The cross is therefore a batched tally, a finalize step and one claim per ticket, and LP quotes live in eight fixed slots on the lull account.
- **No floating point in the cross.** The mint stores the multiplier as an `f64`. The program converts it once to an integer scaled by 10^{12} and uses integers from then on.
- **Two token programs.** AAPLx is a Token-2022 token and USDC an SPL Token token. Each vault is an associated token account of the market address under its own program, and every transfer uses `transfer_checked`.
- **One feed per market**, stored at initialization and matched by every capture.

6

Lulls and the calendar

Definition

A **session** is one regular trading period of the underlying. A **lull** is the interval from one session's close to the next session's open, and every order placed in a lull crosses at the open that ends it.

Kind	Starts	Ends
Overnight	a weekday close	the next weekday's open
Weekend	Friday's close	Monday's open
Holiday	the close before one or more market holidays	the first open after them

A lull that spans a holiday is a holiday lull even when it also spans a weekend. A lull whose starting session closed early carries an `earlyClose` flag, and the worker records the holiday dates inside each lull.

The schedule string

Pyth attaches a market-hours schedule to each equity feed, and Hermes serves it in the feed's metadata. Lull builds its calendar from the schedule of `Equity.US.AAPL/USD`, the feed that also supplies P_{open}. On 2026-09-26 the live market endpoint returned this string, which the worker had fetched from Hermes at 2026-09-24 21:56 UTC:

```
America/New_York;0930-1600,0930-1600,0930-1600,0930-1600,0930-1600,C,C;0907/C,1126/C,1127/0930-1300,
```

The grammar, as implemented in `sdk/src/schedule.ts`:

```
schedule := timeZone ";" weekly ";" holidays
weekly   := day "," day "," day "," day "," day "," day "," day      (Monday first)
day      := "O" | "C" | range ("&" range)*
range    := HHMM "-" HHMM          (end may be 2400; a range ending at "0000" is empty)
holidays := "" | holiday ("," holiday)*
holiday  := MMDD "/" day
```

`O` means open all day and `C` closed. A holiday entry overrides the weekly entry for its month and day. Above, `1126/C` closes Thanksgiving 2026, and `1127/0930-1300` makes the next day an early close at 13:00.

Evaluating the schedule

The parser follows Pyth's own evaluator:

- **Wall-clock time.** Ranges are local times in the schedule's time zone. The open is 09:30 in New York all year, which is 13:30 UTC in summer and 14:30 UTC in winter. The SDK converts with the platform's time-zone database (`Intl`), with no hard-coded offsets.
- **Half-open ranges.** A range `[start, end)` includes its start and excludes its end.
- **Merging.** Adjacent open ranges merge, across midnight as well.
- **Ranges ending at 0000.** A range such as a Sunday `1700-0000`, which appears in the schedule of VIX futures, is empty. Hermes' `market_hours` read it that way, so the parser does too.
- **Rolling holidays.** A holiday entry matches its month and day in any year. Pyth refreshes the list as a rolling window.

A lull is the gap between two consecutive merged open intervals. The SDK classifies it from the local dates strictly between the close and the open: if any is a holiday closure, it is a holiday lull; otherwise, if any is closed by the weekly schedule, a weekend lull; otherwise an overnight lull.

How the calendar is verified

`npm run check:schedule` holds the parser to Pyth's own output in three ways:

1. **Pinned strings.** 28 distinct schedule strings served by Hermes are stored with the `market_hours` Hermes computed at fetch time (whether the market was open, and the next open and close). The parser must reproduce every one.
2. **Live comparison.** When the network is reachable, the gate compares the parser's `market_hours` with Hermes' own for every feed that has a schedule. The README reports this covering more than 1,800 feeds, and the gate requires more than 1,000.
3. **AAPL cases:**

Case	Expected
A regular week	four overnight lulls of 17.5 h; 32.5 open hours
Labor Day (`0907/C`)	a holiday lull from Friday 4 September 20:00 UTC to Tuesday 8 September 13:30 UTC
DST ends on 2026-11-01	the weekend lull is 66.5 h, and the open moves to 14:30 UTC
Thanksgiving and the day after	Thanksgiving closed; the next day closes at 13:00 ET (18:00 UTC)
Christmas Eve, Christmas, New Year	an early close on 24 December; 25 December and 1 January closed
DST starts on 2027-03-14	the weekend lull is 64.5 h, and the open moves to 13:30 UTC
Good Friday 2027 and two Monday holidays	holiday lulls with the right dates

The gate also rejects malformed strings: a wrong number of weekly entries, an unknown time zone, a time past 24:00, an empty range, an impossible holiday date.

From calendar to chain

The worker runs two loops.

Calendar, every 10 minutes by default, fetches the schedule, stores a snapshot, and writes a row for each lull whose open falls between two days ago and 14 days ahead. Each row gets its reveal window from two worker settings:

```
reveal_start = open − REVEAL_LEAD_SECS     (default 1,800 s)
reveal_end   = open − REVEAL_CLOSE_SECS    (default 60 s)
```

Once a lull is on chain its row is no longer updated. The chain is authoritative.

Publish, every 30 seconds by default and only when the worker holds the market authority's key, sends `create_lull(open, close, reveal_start, reveal_end)` for each lull that closes within `PUBLISH_AHEAD_SECS` (3,600 s by default) and whose reveal window starts more than five seconds from now. The program accepts a lull only if

```
close < reveal_start < reveal_end ≤ open     and     now < reveal_start
```

The lull's address is derived from the market and the open time (`["lull", market, open_ts]`), so a market has at most one lull per open.

The live calendar

The read-only deployment computes the calendar but cannot publish it, since the program is not on mainnet. On 2026-09-26, `GET /v1/markets/AAPLx-USDC/calendar?days=14` returned ten lulls. The first five:

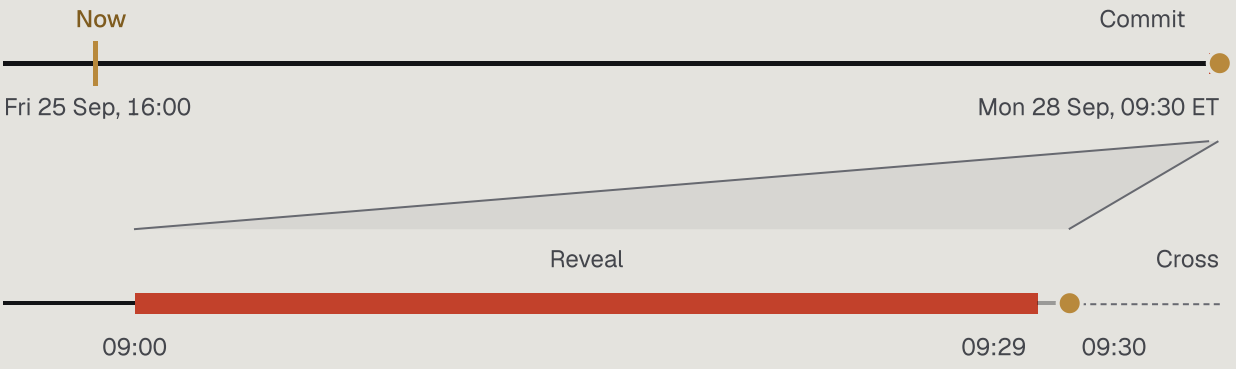
Close (UTC)	Reveal window (UTC)	Open (UTC)	Kind
2026-09-24 20:00	09-25 13:00–13:29	2026-09-25 13:30	overnight
2026-09-25 20:00	09-28 13:00–13:29	2026-09-28 13:30	weekend
2026-09-28 20:00	09-29 13:00–13:29	2026-09-29 13:30	overnight
2026-09-29 20:00	09-30 13:00–13:29	2026-09-30 13:30	overnight
2026-09-30 20:00	10-01 13:00–13:29	2026-10-01 13:30	overnight

The rest followed the same pattern through the open of 8 October, with a weekend lull from 2 to 5 October. None was on chain. The first, already past, had the phase `unpublished`, and the others `scheduled`.

Phases

Phase	When	What participants can do
`scheduled`	on the calendar, not yet on chain	see the countdown
`unpublished`	a past calendar row that never went on chain	nothing
`commit`	from publication until `reveal_start`	seal orders and quotes
`reveal`	`reveal_start` → `reveal_end`	reveal sealed tickets
`sealed`	`reveal_end` → open + delay	wait; unrevealed tickets can be forfeited
`pricing`	open + delay until P_{open} is captured	capture the price
`crossing`	P_{open} captured	tally, then finalize
`settled`	crossed	claim
`cancelled`	no valid price by the deadline, or cancelled by the authority before pricing	claim refunds

Weekend lull, 2 d 17 h. Phase: scheduled, not yet on chain.



— **Commit** until 09:00 ET. Seal orders: only a hash and a bond go on chain.

Reveal 09:00 to 09:29 ET. Each order is checked against its hash and its amount locks.

Cross just after the 09:30 ET open, at P_{open}, once for every revealed order.

The current lull of AAPLx/USDC from the Lull API, in New York time. The top bar is the whole lull to scale. The bottom bar enlarges the reveal window and the open.

Deposits and withdrawals of free balance work in every phase. On chain a lull has five statuses: `Pending` (the commit and reveal windows run on the clock), `Sampling` (collecting TWAP samples), `Priced`, `Finalized` and `Cancelled`. The API splits `Pending` by the clock.

Because the worker publishes a lull up to an hour before its close, the commit phase of the next lull starts in the last hour of the current session. On a normal day an order for the next open can be sealed from about 15:00 ET until 09:00 ET the next morning.

What the program cannot check

The program cannot read Pyth's schedule, so it cannot confirm that a lull's times match the real session. A market authority that published a wrong open time could make P_{open} a price from the wrong moment. Every lull's times are on chain before anyone commits to it, and the API serves the times it derived from Pyth's schedule, so participants can compare the two. Chapter 14 treats the authority as a trusted party with bounded powers, and chapter 18 describes moving that trust to governance.

7

Sealed orders

Balances

Lull separates moving money from placing orders. Each user holds a **balance account** per market, at `["balance", market, owner]`, with four raw amounts:

Field	Unit	Meaning
`base_free` / `base_locked`	raw base	xStock available, and xStock held by revealed sells and LP quotes
`quote_free` / `quote_locked`	raw quote	USDC available, and USDC held by revealed buys and LP quotes

The tokens sit in the market's two vaults, the associated token accounts of the market address. A balance is the program's record of one owner's share.

- **Deposit** moves tokens into the vault with `transfer_checked` and credits `free` with what the vault actually received. The program measures the vault before and after the transfer rather than trusting the requested amount.
- **Withdraw** moves `free` tokens back to the owner, signed by the market address. Locked amounts cannot be withdrawn.

Both work in every phase. As in a dark pool, funding happens outside the auction, so placing an order moves no tokens. A deposit is still public (chapter 13).

Tickets

A sealed order or LP quote is a **ticket** at `["ticket", lull, owner, id]`, with a 32-bit `id` the owner chooses (the API finds a free one on request). A ticket holds the commitment, the bond and the commit time, and after the reveal the revealed fields, the order's class or the quote's slot. Its states are `Committed`, `Revealed` and `Tallied`. It ends by being **claimed**, which settles and closes it, or **forfeited**, which closes it without a trade.

The commitment

```
order: sha256("lull:v1:order" ‖ side u8 ‖ amount u64le ‖ limit u64le ‖ salt[32] ‖ owner ‖ lull)
quote: sha256("lull:v1:quote" ‖ spread u16le ‖ max_base u64le ‖ max_quote u64le ‖ salt[32] ‖ owner ‖
```

Example order

Side	Amount to spend	Limit (USD per share)
Buy / Sell	1000 USDC	$ None: market on open

Tag 13 bytes	The text “lull:v1:order”, which separates orders from LP quotes `6c756c6c3a76313a6f72646572`
Side 1 byte	0, a buy `00`
Amount 8 bytes	1,000,000,000 raw quote: 1,000 USDC × 10^6 `00ca9a3b00000000`
Limit 8 bytes	0: market on open `0000000000000000`
Salt 32 bytes	Random, drawn in this browser. Whoever holds it can reveal the order. Show salt · New salt
Owner 32 bytes	Example owner: `CXvVcXtNcgamNbGPuvmFBTZ4mvBDVy4qQ4xBvdJoK2Ap` `ab5c772e7f3517013108a08aa0b368fd4121b32bfddab9732d08c7b70e7be02d`
Lull 32 bytes	Example lull: `CGMLHPGN3s635wNXoXwtpsTR89d35T4M3t7vWuMCd161` `a75f422077f2044917ede5137b4a031fff09c3768b0a07c854907e105870b172`
	↓ sha256

Commitment
32 bytes
	`9870e0c23156d68c23a5e1e7c1b12553`
	`2318671fb8b7cb89297d38be40469bf5`
	Only this hash, the bond and the owner's address go on chain when the order is sealed.

Example inputs. The hash is computed in your browser with the SDK function the order ticket seals with. The owner and the lull are random example keys, not real accounts, and the salt never leaves this page.

Part	Order	Quote
Domain tag (ASCII)	`lull:v1:order`, 13 bytes	`lull:v1:quote`, 13 bytes
Fields	side 1 + amount 8 + limit 8	spread 2 + max base 8 + max quote 8
Salt, owner, lull	32 + 32 + 32	32 + 32 + 32
Total preimage	126 bytes	127 bytes

Each part has a job:

- **The fields** make the commitment binding. A reveal with any other side, amount or limit fails with `CommitmentMismatch`.
- **The salt**, 256 random bits generated on the client, makes it hiding. Without it, nobody can test candidate orders against the hash, however few plausible orders there are.
- **The owner** binds the order to one wallet's balance.
- **The lull** binds it to one auction, so it cannot be replayed in another.
- **The domain tag** separates orders from quotes and carries a version.

Because the preimage fixes the owner and the lull, a reveal needs no owner signature. Knowing the salt is the authorization (chapter 8). The SDK (`sdk/src/commitment.ts`) and the program (`instructions/tickets.rs`) compute the same hash, and the cross gate checks that it changes with every field, the owner and the kind.

Salts

The salt comes from the platform's cryptographic random source. The rules for handling it are part of the design:

- The client stores the order and salt **before** sending the commit. A commit whose salt is lost can never be revealed.
- The API never receives the salt or the fields before the reveal lands on chain. Its commit endpoint takes only the commitment.
- The web app exports and imports stored secrets as JSON, so a user can reveal from another device.
- Anyone holding an export can read the order early and reveal it, but cannot change or redirect it.

The bond

Every ticket posts `bond_lamports` of SOL at commit, 0.01 SOL (10,000,000 lamports) by default. The amount is copied into the ticket, so later changes to the market's bond do not affect it.

Outcome	Bond	Ticket rent
Revealed, then claimed (settled or cancelled lull)	returned to the owner	returned to the owner
Not revealed by `reveal_end`, then forfeited	to the market's treasury	returned to the owner

Orders and quotes post the same bond, so a commit does not show which kind it is. The bond prices the free option of declining to reveal (chapter 15). The commit also pays the ticket's rent. At the mainnet rent rate implied by `docs/LAUNCH.md`, a 154-byte ticket holds about 0.0014 SOL (a derived figure, appendix B).

Order types

Order	Locks	`amount`	`limit`
Buy, market on open	USDC	USDC to spend (raw quote)	`0`
Buy, limit on open	USDC	USDC to spend (raw quote)	most it will pay per share (raw quote per share)
Sell, market on open	xStock	xStock to sell (raw base)	`0`
Sell, limit on open	xStock	xStock to sell (raw base)	least it will accept per share (raw quote per share)

Buys are sized in USDC. A market-on-open buy cannot name a share count and lock the right USDC before the price exists. Lull therefore locks what the buy will spend, and the shares follow from P_{open}. A buy may spend less than its amount, never more.

Sells are sized in the xStock, in raw base units. The client converts shares with the mint's current multiplier. If the multiplier changes before the cross, the order sells the same raw amount, which is then a slightly different number of shares.

Limits are per share, in raw USDC per one share, and never pass through the Scaled UI multiplier (chapter 12).

Minimums are checked at reveal, when the amount becomes known: `min_buy_quote`, 1 USDC by default, and `min_sell_base`, 100,000 raw base (0.001 of an unscaled token) by default.

What a commit shows

A commit shows the owner, the ticket id, the lull, the bond and the time. It does not show whether the ticket is an order or a quote, or its side, size or limit. Nothing is locked, and the owner does not need the funds yet: the balance check happens at reveal.

`commit` requires the lull to be `Pending` and the clock to be before `reveal_start`, so the commit phase runs from the lull's publication until the reveal window. A sealed ticket is final. To change an order, a user seals a new one and does not reveal the old one, which forfeits its bond.

8

Reveal

The reveal window

Each lull has one reveal window, fixed when the lull is published:

```
reveal_start = open − REVEAL_LEAD_SECS     default 1,800 s   (09:00 ET on a normal day)
reveal_end   = open − REVEAL_CLOSE_SECS    default    60 s   (09:29 ET)
```

These are worker settings, not market parameters, and each lull stores its own window. A reveal must land with a cluster time `t` such that `reveal_start ≤ t < reveal_end`. Solana's clock can differ slightly from a device clock, so reveals sent at the edges can land outside. The minute from `reveal_end` to the open belongs to the sealed phase, in which nothing can change.

What the program checks

`reveal_order(side, amount, limit, salt)` checks, in order:

1. the lull is `Pending` and the clock is inside the window (`BadLullStatus`, `NotInRevealWindow`);
2. the ticket is `Committed` (`BadTicketState`);
3. the side is buy or sell (`BadSide`);
4. the recomputed commitment matches (`CommitmentMismatch`);
5. the amount meets the side's minimum (`OrderTooSmall`);
6. the owner's free balance covers the amount (`InsufficientBalance`), which then moves from free to locked: USDC for a buy, xStock for a sell.

It records the fields on the ticket, marks it `Revealed`, increments the lull's reveal count and emits `OrderRevealed`.

`reveal_quote(spread, max_base, max_quote, salt)` runs the same window, state and hash checks, then requires `spread ≤ max_lp_spread_bps` (`SpreadTooWide`), a non-empty quote (`EmptyQuote`), and fewer than eight revealed quotes (`QuoteSlotsFull`). It locks **both** maximums, takes the next slot, and updates `spread_hi_ask` (the widest spread among quotes with `max_base > 0`) and `spread_hi_bid` (among quotes with `max_quote > 0`), which the cross needs.

Because the balance check happens here, a user can seal before depositing but must deposit before revealing.

Relayed reveals

A reveal has no owner signature. It needs the market, the lull, the ticket and the owner's balance, and any wallet can pay the fee. The preimage fixes the owner and the lull, so whoever reveals can reveal only what the owner sealed, for the owner's balance, in that lull.

This makes delegation possible with no further mechanism. A user who will be offline can hand the order and salt to a **relayer**, who reveals and pays the fee. The relayer cannot alter or redirect the

order or move funds. It does learn the order early, and it can fail to reveal. The end-to-end gate exercises this: the market authority's wallet pays for the two LP reveals. No relayer service is built. `docs/LAUNCH.md` lists an opt-in relayer as a launch consideration, and chapter 19 describes a planned registry of staked relayers.

Keeping the operator blind

`POST /v1/tx/reveal` returns a **template**: an unsigned transaction whose reveal instruction has every secret field zeroed, the salt included. The client's SDK replaces the instruction data with the real fields from local storage (`fillRevealTemplate`), leaving the accounts, fee payer and blockhash as built. `check:api` asserts that the template's secret bytes are zero. The operator learns an order when it lands on chain, like everyone else.

After the window: forfeits

`forfeit` is permissionless once `now ≥ reveal_end`, for a ticket still `Committed`. It sends the bond to the treasury, closes the ticket, returns its rent to the owner and emits `Forfeited`. An unrevealed order never locked anything, so the owner's balance is untouched. The worker forfeits in batches of five. The instruction does not check the lull's status, so unrevealed tickets in a cancelled lull are forfeited too.

Why a reveal fails

Cause	Fixable in the window?
Free balance below the amount, or below either maximum of a quote	yes: deposit and reveal again
The reveal lands after `reveal_end`	no
The salt was lost	no
The market raised its minimum after the order was sealed	no, the sealed amount cannot change
The market lowered its spread cap below the quote's spread	no
Eight quotes were already revealed	no
The lull was cancelled before the reveal	no
The owner chose not to reveal	

Every unfixed case ends in a forfeit. Several depend on parameter changes by the market authority after sealing. Chapter 14 lists that as an authority power, and chapter 18 proposes a timelock so that such changes cannot land inside a lull users have committed to.

Timing inside the window

When an order is revealed does not affect its fill. The heavy side fills pro rata and the light side in full, so there is no queue to be early in. Revealing late shortens the time an order is public, and

revealing early lowers the risk of missing the window. The imbalance becomes readable with the first reveal, whoever sends it.

9

Pricing

What P_{open} is

P_{open} is the price at which every order in a lull crosses: the first fully verified Pyth price of the underlying published at or after the open plus a short delay, or, if that price is too uncertain, the mean of a short series of later verified prices. For AAPLx/USDC the feed is `Equity.US.AAPL/USD` (`49f6b65c…5688`), Pyth's price of Apple in the US regular session, in USD per share.

The rule is a pure state machine in `programs/lull/src/pricing.rs`, so the direct path, the TWAP path and every rejection are unit-tested on the host. `capture_price` adds the account checks around it. `npm run check:exact` also runs both paths through the instruction itself, with real Pyth updates posted and verified on a local validator (chapter 17).

Which accounts qualify

`capture_price` reads one `PriceUpdateV2` account owned by the Pyth receiver. Two kinds exist:

- **The sponsored push account**, which Pyth keeps updated: `D9uk39pqZMcnmtPP9WeC8cREUpKZmyXLga9mSQ79SphW`, derived from shard 1 and the feed id under the push oracle program. The shard-0 account for the same feed has been stale since 2026-08-14. Shard 1 updated about every 10 seconds in the project's checks, including after the 16:00 ET close.
- **A posted update**: a specific Pyth update that anyone posts through the receiver into a fresh account. The worker's exact path does this.

The program parses the account by hand (`oracle.rs`, layout in appendix B) and requires receiver ownership and the `PriceUpdateV2` discriminator (`BadPriceAccount`), `VerificationLevel::Full` (`PriceNotFullyVerified`), the market's feed id (`WrongFeed`) and a positive price (`NonPositivePrice`).

The capture rule

Let `start = open_ts + open_delay_secs`, which is 09:30:30 ET on a normal day with the default 30-second delay. The delay skips the first seconds of the session, while opening prints settle.

```
now ≥ start                                                  else TooEarly
start ≤ publish_time                                         else PriceBeforeOpen
publish_time ≤ start + max_price_lag_secs                    else PriceTooLate
conf × 10,000 ≤ price × max_conf_bps   ⟶  P_open = price     (source = direct)
otherwise                              ⟶  status = Sampling, first TWAP sample
```

With defaults, a direct P_{open} must be published between 09:30:30 and 09:31:00 ET with a confidence interval of at most 25 bps of the price.

The lower bound on the publish time does real work. The push account keeps updating after the close, so during a lull it always holds some price. Without the bound, a capture could use an update

from the lull itself. With it, nothing published before `start` can become P_{open}.

The TWAP fallback

If the first eligible update is too uncertain, the lull switches to `Sampling` and keeps that update as its first sample. Later captures add samples, each fully verified, for the feed, with a positive price and:

- a publish time strictly later than the previous sample's (`StaleSample`);
- a publish time no later than `first_ts + twap_window_secs + max_price_lag_secs` (`PriceTooLate`);
- the first sample's exponent (otherwise rejected as `WrongFeed`).

When a sample arrives with `publish_time ≥ first_ts + twap_window_secs` and the count has reached `twap_min_samples`, P_{open} is the integer mean of all samples, the first included, and the source is TWAP. Later samples are not confidence-gated: the mean over the window is the answer to an uncertain instant. With the defaults (a 60-second window and 3 samples), the unit tests start sampling at `start + 2` with a 50 bps confidence, reject a repeated publish time, and finish with samples at `start + 30` and `start + 62`. `check:exact` runs the same path end to end: on a test market with a confidence limit of zero, so that no single price qualifies, three real Hermes updates are posted, verified and captured, and the lull is priced at their integer mean.

The exactness flag

The push account holds whichever update Pyth last posted, so a capture from it is a valid price from the window but not necessarily the first one after `start`. Every `PriceUpdateV2` carries `prev_publish_time`, the publish time of the feed's previous Pythnet update. On a direct capture the program records

```
price_exact = (prev_publish_time < start ≤ publish_time)
```

When it is true, the captured update is provably the first at or after `start`. A TWAP price is never exact. The flag is stored on the lull and emitted in `PriceCaptured`.

The exact path

With a Pyth Pro key, the worker's crank makes the capture exact. Hermes serves updates by timestamp only with a key (it answers 401 without one), and asked for a time `t` it returns the **last** update published at or before `t`, not the first after it. The crank therefore finds the first update at or after `start` in two stages (`firstUpdateFrom` in `server/src/worker/pythHistory.ts`):

- **One question about the end of the window.** It asks for `start + max_price_lag_secs`, or for one second before the present if that is earlier. If the answer was published before `start`, no update exists in the window, or none yet.
- **A binary search over the window.** Otherwise some update lies in `[start, start + lag]`, and the search narrows the window to the first one. Over a 30-second window that takes about five requests. They are paced, and a rate-limit answer (HTTP 429, observed on bursts) is retried after a pause.

The crank then posts the update through the receiver itself (`sdk/src/receiver.ts`), following the receiver SDK's own steps:

1. **Parse** Hermes' accumulator update (`PNAU`): a Wormhole VAA signing a Merkle root, followed by each price message and its proof.
2. **Verify** the VAA on the Wormhole receiver that the Pyth receiver's configuration points to (`HDwcJBJXjL9FpJ7UBsYBtaDjsBUhuLCUYoz3zr8SWWaQ`): create an encoded-VAA account, write the VAA in chunks, and call `verify_encoded_vaa_v1`, which checks every guardian signature against the current guardian set.
3. **Post and capture in one transaction.** `post_update` checks the AAPL message's Merkle proof against the verified VAA and writes a new `PriceUpdateV2` with `VerificationLevel::Full`. `capture_price` reads it in the same transaction, so the flag is set.
4. **Reclaim** both posted accounts' rent in a last transaction.

The crank holds back its own push-account capture for 15 seconds after `start` (`EXACT_GRACE_SECS`), so that an update that is merely inside the window does not win the race. If Hermes cannot help in that time, it falls back to the push account. `npm run check:exact` runs this path against the mainnet receiver and Wormhole programs, with their live configuration and guardian set, on a local validator (chapter 17). The same posting routine, `postAndCapture`, serves any update, so the gate uses it for TWAP samples too.

Bounded discretion

Capture is permissionless, and the first valid capture wins. The program accepts any fully verified update published in `[start, start + max_price_lag_secs]`, and anyone can post such an update or capture from the push account while it holds one. A party that wanted a particular price could choose among the updates of that 30-second window (by default) and try to land first. Three things limit this: the window is short, which is why the default lag is 30 seconds; the honest crank captures within seconds of the first eligible update; and a capture that is not the first update shows as `price_exact = false`. The residual discretion is the underlying's movement within the window. The flag exists so that it is measured rather than assumed.

No price: cancellation

```
deadline = open + open_delay_secs + 2 × max_price_lag_secs + twap_window_secs
         = open + 150 s                                          (defaults)
```

After the deadline, anyone can cancel a lull that is still `Pending` or `Sampling`. The market authority can cancel at any time before pricing, for example during a trading halt. A cancelled lull trades nothing, and claims return every revealed lock with its bond and rent. Unrevealed tickets are still forfeited.

The program bounds when a captured update was published, not when the capture is sent. Until someone cancels, a late capture of an in-window update is valid, and Hermes keeps serving such updates by timestamp, where the same search finds them. Lull's worker simply cancels once the deadline has passed without a price.

The multiplier is read with the price

In the same instruction, the program reads the mint's Scaled UI multiplier (applying a scheduled new one once its time has passed) and computes the conversion rate for the cross (chapter 12):

```
px = price × M × 10^(expo + quote_decimals − base_decimals)          M = multiplier × 10¹²
```

The price, confidence, exponent, publish time, source, exactness, multiplier and `px` are stored on the lull, and its status becomes `Priced`.

A default timeline

Time (ET)	Event
09:00:00	reveal window opens
09:29:00	reveal window closes; forfeits may start
09:30:00	regular open
09:30:30	`start`: the earliest publish time for P_{open}, and the earliest capture
09:30:45	the worker may fall back to the push account
09:31:00	the latest publish time for a direct P_{open}
09:32:30	pricing deadline; anyone may cancel an unpriced lull

10

The cross

Overview

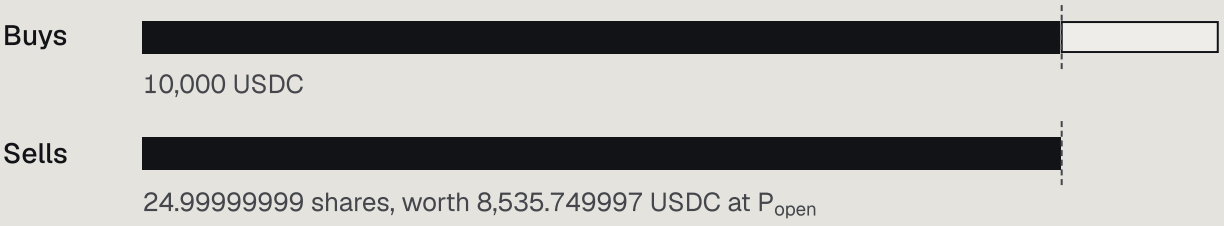
Once P_{open} is captured, the lull's revealed book crosses once, at `px`. The cross is split into three permissionless instructions, because a transaction can carry only a limited number of accounts:

1. **`tally`** classifies revealed orders and adds them to running totals. It takes tickets as extra accounts, runs in any number of batches, and skips tickets already tallied. The worker sends batches of eight.
2. **`finalize`** requires every revealed order to be tallied (`tallied == reveals`, otherwise `TallyIncomplete`). It decides the heavy side, allocates the matched volume and the LP leg, and stores the result.
3. **`claim`** settles one ticket into its owner's balance, releases its lock, closes it and returns its bond and rent. The worker sends batches of four.

Once every ticket is gone, claimed or forfeited (`claimed + forfeits == commits`, otherwise `TicketsOutstanding`), anyone can call **`close_lull`** on a finalized or cancelled lull. It closes the lull account, returns its rent to the market authority and emits `LullClosed`. The worker does this on a later pass, after saving the LP quotes and their fills to its database (`LullWindow.lpQuotes`), because those exist only in the lull account. The API serves them from there once the account is gone.

Example inputs

Buys, total	Sells, total	P_{open} (USD per share)
10000 USDC	25 shares	$ 341.43



Heavy side	Buys. The sells fill in full.
Matched at P_{open}	8,535.749997 USDC for 24.99999999 shares Shared pro rata across the heavy side's orders, by amount.
Imbalance	1,464.250003 USDC of buys, about 4.28858038 shares at P_{open} LP quotes absorb what they can, cheapest first. The rest is refunded at claim.
● px at P_{open}	3,425,461,389,515 Raw USDC per raw AAPLx × 10^{12}, with the multiplier already applied.

Example inputs. P_{open} starts at the latest Pyth price and can be changed. Every order is market on open and no LP quote is revealed. Shares convert to raw units with the AAPLx mint's live Scaled UI multiplier (1.00326901254), and the matching runs the SDK's mirror of the program's cross.

The math is in `programs/lull/src/math.rs`, with no Anchor types, and `sdk/src/matching.ts` mirrors it line for line. Below, `base` is raw xStock, `quote` raw USDC, `px` raw quote per raw base × 10^{12}, a limit is raw quote per share, `SCALE = 10¹²` and `BPS = 10,000`.

Step 1: classify

Class	Condition	Takes part in
None	the limit is not met at P_{open}	nothing; fully refunded
A	met at P_{open}, but not at the worst revealed LP price	the matched volume only
B	met at P_{open} and at the worst revealed LP price	the matched volume and the LP leg

The worst LP price is `P_open × (1 + s_hi_ask)` for a buy and `P_open × (1 − s_hi_bid)` for a sell, where `s_hi_ask` is the widest revealed spread among quotes that sell base (`max_base > 0`) and `s_hi_bid` among quotes that spend quote (`max_quote > 0`). A market-on-open order is always class B. With no quote on the absorbing side, `s_hi` is zero and every eligible order is class B.

The comparison is exact. P_{open} per share in raw quote is `price × 10^e`, with `e = expo + quote_decimals`. `cmp_limit` compares `limit × BPS` with `price × factor × 10^e` by integer cross-multiplication:

```
buy:   limit < P_open                          ⟶ None
       limit < P_open × (BPS + s_hi_ask)/BPS   ⟶ A
       otherwise                               ⟶ B
sell:  limit > P_open                          ⟶ None
       limit > P_open × (BPS − s_hi_bid)/BPS   ⟶ A
       otherwise                               ⟶ B
```

Limits and the Pyth price are both per share, so the multiplier plays no part.

Step 2: tally

Total	Sum over eligible orders (classes A and B)
`buy_quote_all`	quote amounts of buys
`buy_quote_lp`	quote amounts of class-B buys
`buy_base_at_px`	`⌊quote × SCALE / px⌋` per buy
`sell_base_all`	base amounts of sells
`sell_base_lp`	base amounts of class-B sells
`sell_quote_at_px`	`⌊base × px / SCALE⌋` per sell

Each conversion rounds down per order, exactly as each light-side order will be paid, so totals and settlements agree to the raw unit.

Step 3: the heavy side

```
buy-heavy   if buy_quote_all > 0 and buy_quote_all ≥ sell_quote_at_px
sell-heavy  else if sell_base_all > 0 and sell_quote_at_px > buy_quote_all
none        otherwise
```

The light side fills completely at `px`. The heavy side shares the matched volume pro rata between classes A and B, and class B also shares what the LP quotes absorb.

Step 4: allocate, buy-heavy

The sellers' value at `px`, `qm = sell_quote_at_px`, is the matched quote, and they deliver `s = sell_base_all`.

```
q_all = buy_quote_all        q_B = buy_quote_lp        q_A = q_all − q_B
pay_A          = ⌈qm × q_A / q_all⌉        class A's share of the matched quote
base_A         = ⌊s  × q_A / q_all⌋        class A's share of the sellers' base
pay_B_matched  = qm − pay_A
left           = q_B − pay_B_matched       class B's unspent quote, offered to LPs
```

The LP leg walks the quotes with `max_base > 0`, cheapest spread first, ties in reveal order, each at its own price:

```
px_k = ⌈px × (BPS + spread_k) / BPS⌉
full = ⌈max_base_k × px_k / SCALE⌉
if full ≤ left:  fill_k = (max_base_k, full)
else:            b = ⌊left × SCALE / px_k⌋;  fill_k = (b, ⌈b × px_k / SCALE⌉)
left = left − fill_k.quote
```

```
class A:  debit = pay_A,                    credit = base_A,                weight = q_A
class B:  debit = pay_B_matched + lp_quote, credit = s − base_A + lp_base,  weight = q_B
```

Debits are in the locked token (USDC here), credits in the other token, and weights are the classes' total locked amounts.

Step 4: allocate, sell-heavy

The buyers' base at `px`, `bm = buy_base_at_px`, is the matched base, and they pay `q = buy_quote_all`.

```
s_all = sell_base_all        s_B = sell_base_lp        s_A = s_all − s_B
give_A          = ⌈bm × s_A / s_all⌉
recv_A          = ⌊q  × s_A / s_all⌋
give_B_matched  = bm − give_A
left            = s_B − give_B_matched     class B's unsold base, offered to LPs
```

The LP leg walks the quotes with `max_quote > 0`, cheapest first:

```
px_k   = ⌊px × (BPS − spread_k) / BPS⌋     (a quote with px_k = 0 is skipped)
b      = min(⌊max_quote_k × SCALE / px_k⌋, left)
fill_k = (b, ⌊b × px_k / SCALE⌋)
left   = left − b
```

```
class A:  debit = give_A,                    credit = recv_A,                   weight = s_A
class B:  debit = give_B_matched + lp_base,  credit = q − recv_A + lp_quote,    weight = s_B
```

Step 5: settle

For a heavy-side order in class `c`:

```
debit  = min(⌈amount × debit_c / weight_c⌉, amount)
credit = ⌊amount × credit_c / weight_c⌋
```

For a light-side order: a sell settles as `(amount, ⌊amount × px / SCALE⌋)`, and a buy as `(amount, ⌊amount × SCALE / px⌋)`. A class-None order, or any order in a lull with no heavy side, settles as `(0, 0)`. The claim removes the whole lock from `locked`, returns `amount − debit` to `free`, and adds `credit` to `free` in the other token.

An LP quote settles from its slot: when buys were heavy it pays `fill_base` and receives `fill_quote`, and when sells were heavy the reverse. Unused parts of both locks return to `free`. In a cancelled lull every revealed ticket settles as `(0, 0)`.

Rounding favors the vault

Debits round up and credits round down, so in each token the credits paid out never exceed the debits taken in, and no token is created. The remainder, at most one raw unit per order, stays in the vault. On the LP leg the rounding favors the LP: an ask rounds up and a bid rounds down, so each LP gets at least its quoted price. Appendix C gives the derivations.

Invariants

`npm run check:cross` runs four fast-check property tests, each on 4,000 random books with up to 16 orders and 8 quotes, prices from $1 to $2,000, multipliers from 1.0 to 1.3, spreads from 0 to 300 bps and limits within ±400 bps of the price. A randomized Rust test runs 3,000 more books through `math.rs` itself. Together they check that:

1. matched buy equals matched sell: base and quote delivered reach the other side, within one raw unit per order;
2. no token is created in either token;
3. limits hold, on the LP leg too, up to two raw units of each token (about $0.00001 on AAPLx);
4. no LP fill exceeds `max_base` or `max_quote`, and every LP gets at least its quoted price;
5. the light side fills in full, and a wider quote never fills while a cheaper one has room.

Two fixtures pin exact values. At $335.62834 with the multiplier 1.0032690125398187, one unscaled AAPLx (10^8 raw) is worth 336.725513 USDC in both Rust and TypeScript, and a small buy-heavy book produces hand-checked fills.

A worked example: sell-heavy

The numbers are hypothetical. The multiplier is 1, so one share is 10^8 raw base, and amounts are shown in shares and USDC. P_{open} is $100.00 (`price = 10,000,000`, `expo = −5`), so `px = 10¹²`.

Ticket	Revealed
B1	buy, market on open, 3,000 USDC
S1	sell, market on open, 60 shares
S2	sell, limit on open, at least $99.60, 40 shares
L1	quote, 20 bps, max quote 1,996 USDC, max base 0
L2	quote, 50 bps, max quote 5,000 USDC, max base 0

Classify. The widest bid spread is 50 bps, so the worst LP price for sellers is $99.50. S1 is class B. S2's $99.60 limit is met at $100.00 but not at $99.50, so S2 is class A. B1 is class B.

Heavy side. Sells are worth 10,000 USDC at P_{open} against 3,000 USDC of buys, so sells are heavy. B1 buys 30 shares for 3,000 USDC.

Pro rata. Class A holds 40 of the 100 shares, class B 60:

```
give_A = ⌈30 × 40/100⌉ = 12 shares        recv_A = ⌊3,000 × 40/100⌋ = 1,200 USDC
give_B_matched = 18 shares                  left = 60 − 18 = 42 shares
```

LP leg. L1 bids $99.80 with 1,996 USDC, so it buys 20 shares for 1,996 USDC, leaving 22. L2 bids $99.50, and its 5,000 USDC could buy 50.25 shares, so it takes the remaining 22 for 2,189 USDC.

Ticket	Locked	Pays	Receives	Refunded
B1	3,000 USDC	3,000 USDC	30 shares	none
S1 (class B)	60 shares	60 shares	5,985 USDC	none
S2 (class A)	40 shares	12 shares	1,200 USDC	28 shares
L1	1,996 USDC	1,996 USDC	20 shares	none
L2	5,000 USDC	2,189 USDC	22 shares	2,811 USDC

Sellers deliver 72 shares, and B1 and the LPs receive 72. B1 and the LPs pay 7,185 USDC, and the sellers receive 7,185. S1 averages $99.75 per share, between P_{open} and the worst bid. S2 gets exactly $100.00 on its matched part and keeps 28 shares, although L1's $99.80 bid would have met its limit. That is the cost of the class rule (chapter 15). `simulateCross` in the SDK reproduces

these figures to the raw unit. Chapter 17 walks through a buy-heavy cross run against real mainnet accounts.

Why the cross does not compute a price

P_{open} is an input. The cross decides only eligibility, allocation and the LP leg, which keeps it linear in the number of orders, with the eight quotes sorted by a fixed insertion sort. Every step is an integer function that can be tested exhaustively and, in Stage 2, evaluated under multi-party computation (chapter 13).

11

Liquidity providers

Why LPs are needed

Buys and sells in a lull rarely balance. After matching at P_{open}, part of the heavy side is left over, and without a counterparty it is refunded. LPs turn part of that remainder into fills. Before seeing the book, they commit to sell the xStock above P_{open} if buyers are heavy, or to buy it below P_{open} if sellers are heavy, at a spread they choose.

The quote

Field	Unit	Meaning
`spread_bps`	basis points	distance from P_{open} at which the quote fills
`max_base`	raw base	most xStock sold if buyers are heavy
`max_quote`	raw quote	most USDC spent if sellers are heavy

A quote may offer both sides, but only the side opposite the heavy one can fill. The program requires `spread_bps ≤ max_lp_spread_bps` (200 bps by default), a non-empty quote, and at most eight quotes per lull, taken in reveal order. At reveal both maximums are locked in full, and each is exactly what the LP could have to deliver, so a fill can never overdraw.

A quote follows an order's lifecycle: sealed with a commitment over the spread, both maximums, a salt, the LP's address and the lull; posting the same bond as an order, so the commit does not reveal it is a quote; revealed in the window; crossed; claimed.

Pricing: pay-as-quoted, cheapest first

Heavy side	The quote	At
buys	sells xStock	`P_open × (1 + spread)`
sells	buys xStock	`P_open × (1 − spread)`

Quotes fill cheapest spread first, ties in reveal order. A wider quote never fills while a cheaper one has room, and the last to fill may fill in part. Rounding favors the LP, so each gets at least its quoted price.

Only the heavy side's class-B orders reach the LP leg: market-on-open orders, and limit orders whose limit clears the worst revealed LP price on their side. Because the widest revealed spread sets that price, a wide quote narrows the set of limit orders that can reach any LP on its side, including the cheap ones (chapter 15).

Where the LP stands

- **It commits before the book is visible.** During the reveal window it can watch the imbalance form, but it can only decline to reveal, which forfeits the bond.
- **It prices against an official, external price.** Its fills are at P_{open} plus or minus its spread, where P_{open} is Pyth's aggregate of the regular session just after the open. No pool price enters the fill.
- **Both sides are locked until claim**, though only one can fill.
- **A cancelled lull fills nothing**, and both locks return.
- **It carries the xStock's issuer risk** on the base it holds, like every holder.

How an LP manages the exposure of selling at `P_open × (1 + s)`, for example by trading around the regular session, is its own business. The design's contribution is that the reference price is the official open, which the LP can observe, rather than a pool price from the middle of the night.

When no LP quotes

With no quotes on the absorbing side, every eligible heavy-side order is class B and the LP leg is empty. The heavy side shares the matched volume pro rata, and the rest is refunded. This is safe, but large imbalances go largely unfilled. `docs/LAUNCH.md` makes LP onboarding a launch prerequisite: at least one market maker per market, with documented quoting obligations.

Economics

An LP earns its spread, relative to P_{open}, on the volume it absorbs. Pay-as-quoted pricing pays each LP its own spread rather than a common clearing spread, so an LP has no reason to quote below what it needs, and LPs compete on priority. The protocol pays LPs nothing else. No LP reward program exists or is planned here, and the planned protocol fee (chapter 19) is charged to orders, not LP fills, so that it does not widen quotes.

Limits of the current design

- **Eight slots**, first come first served. A quote revealed after the eighth is rejected and forfeits its bond. This reintroduces a race among LPs and a way to crowd them out (chapter 14).
- **One spread per quote.** A price that worsens with size takes several quotes, and several slots.
- **Both sides locked.** An LP willing to take either side funds both for the whole lull.

12

Units and the Scaled UI multiplier

Why units get their own chapter

Tokenized equities put two scales on one asset. The token program counts raw units, and the wallet shows shares. A price per share must be applied to an amount in raw units, and the ratio between the two changes over time. Every amount that crosses that boundary is a place where a multiplier can be applied twice or not at all. Lull's rule is to name the unit of every amount and to apply the ratio in exactly one place.

The units

Unit	Meaning	AAPLx/USDC	Used for
raw base	Token-2022 amount before the Scaled UI multiplier	8 decimals	sell sizes, `max_base`, balances, fills
raw quote	USDC base units	6 decimals	buy sizes, `max_quote`, balances, fills
raw quote per share	USDC base units per one share (one UI xStock)		limits
`px`	raw quote per raw base × 10^{12}		every conversion in the cross
multiplier	shares per unscaled token × 10^{12}		read from the mint at capture

In the API every amount is a decimal integer string whose field name ends in `Raw`, with a `unit` or `amountUnit` field wherever the unit can vary. `GET /v1/units` returns the definitions.

Scaled UI Amount

The AAPLx mint stores a multiplier, a new multiplier and the time from which the new one applies. A wallet displays:

```
shares = raw base × multiplier / 10^8
```

AAPLx shares and raw base units, 8 decimals

Shares, as a wallet shows them		Raw base, as the program stores it
1 shares	⇄	99674163

`raw base = ⌊shares × 10^20 / M⌋` with M = `1,003,269,012,540`, a multiplier of 1.00326901254. Back in shares that is 0.99999999, because the raw amount is rounded down.

USDC and raw quote units, 6 decimals

USDC		Raw quote
1000 USDC	⇄	1000000000

`raw quote = USDC × 10^6`. Digits past the sixth decimal are dropped.

Example inputs, converted with the SDK's unit helpers. AAPLx uses the live Scaled UI multiplier of its mint, as the app does. Edit either side of a pair.

In the market data fetched on 2026-09-26 the extension held a multiplier of 1.0026642075893797 and a new multiplier of 1.0032690125398187, effective from 2026-08-08 00:30 UTC. Lull applies the new value once its time has passed, the rule Token-2022 uses for display, so the effective multiplier × 10^{12} is 1,003,269,012,540.

`oracle::scaled_ui_multiplier` walks the mint's extension list to the Scaled UI Amount entry (type 25), reads the two `f64` values and the timestamp, requires the chosen value to be finite, positive and below 10^6, and stores `round(m × 10¹²)`. From then on everything is an integer. A mint without the extension has a multiplier of exactly 10^{12}. The multiplier is read once per lull, in `capture_price`, and stored on the lull, and the whole cross uses that value.

From P_{open} to px

Pyth publishes the price as an integer `price` with an exponent `expo`, in USD per share. With `m` the multiplier as a real number and `M = m × 10¹²`:

```
raw quote per share     = price × 10^(expo + quote_dec)
shares per raw base     = m / 10^base_dec
raw quote per raw base  = price × 10^(expo + quote_dec) × m / 10^base_dec
px = that × 10¹²        = price × M × 10^(expo + quote_dec − base_dec)
```

The price scale and the multiplier scale are both 10^{12}, so they cancel. For AAPLx/USDC, `expo = -5`, `quote_dec = 6` and `base_dec = 8`, giving `px = ⌊price × M / 10^7⌋`. `px` already contains the multiplier, and no later step applies it again.

The program's unit test and the SDK's fixture check this against a real price. At AAPL $335.62834 (`price = 33,562,834`) and M = 1,003,269,012,540, one whole unscaled AAPLx (10^8 raw) is worth `⌊10^8 × px / 10^12⌋ = 336,725,513` raw quote, which is 336.725513 USDC, or 1.00327 shares at $335.62834.

Conversions and their rounding

Conversion	Formula	Rounding
base → quote, what a seller receives	`⌊base × px / 10¹²⌋`	down
base → quote, what an LP ask costs	`⌈base × px_k / 10¹²⌉`	up
quote → base, what a buyer receives	`⌊quote × 10¹² / px⌋`	down
shares → raw base, sizing a sell	`⌊shares × 10^(base_dec + 12) / M⌋`	down
raw base → shares, for display	`raw × M / 10^(base_dec + 12)`	truncated
Pyth price → raw quote per share	`price × 10^(expo + quote_dec)`	down

The SDK exposes these as named functions (`sharesToBaseRaw`, `baseRawToShares`, `quoteUiToRaw`, `priceToQuotePerShare`, `pythPriceToUi`, `pxToUsdPerShare`), and the frontend contract asks clients to convert only through them. With the multiplier above, `sharesToBaseRaw("3")` gives 299,022,491 raw base, the sell size in the end-to-end cross of chapter 17.

A limit is raw quote per share and the Pyth price is per share, so classification never touches the multiplier, and a limit means the same thing before and after a multiplier change.

One source for the ratio

Pyth also publishes a redemption-rate feed for AAPLx, `Crypto.AAPLX/AAPL.RR`, which describes the same quantity as the mint's multiplier. When the project checked it, the RR feed's shard-0 push account had last updated on 2026-07-21 and reported 1.0026642, exactly the mint's old multiplier. Multiplying by both would count the ratio twice. The mint is authoritative, because it defines what every wallet displays, while a feed can lag behind a change. Lull reads only the mint and keeps the RR feed id for monitoring.

What this means for an order

- A **buy** is sized in raw quote. Its shares are computed with the multiplier in effect at capture.
- A **sell** is sealed in raw base, converted from shares with the multiplier at sealing time. If the multiplier changes before the cross, it sells the same raw amount, which is then slightly different in shares.
- **Balances** are stored raw and displayed in shares with the current multiplier.
- The **average price per share** of a buy is `quote paid / (base received × M / 10^(8 + 12))`, and a sell's is the equivalent. Clients compute it with integer math or the SDK.

13

Privacy

The promise and its limit

Lull's privacy promise is that nobody can see the overnight order queue and trade against it before the open. Stage 1, which is built, keeps that promise until the reveal window. From the first reveal until the cross the book is public, but nobody who reads it can move the price it will cross at. Stage 2, designed but not built, removes the public window with multi-party computation. This chapter follows `docs/PRIVACY.md`.

Stage 1: commit–reveal

What is hidden, and until when

Information	Visible to anyone from
That a wallet placed a ticket in this lull, its ticket id and bond	commit
Whether the ticket is an order or an LP quote	reveal (both post the same bond)
Side, size and limit, or spread and maximum sizes	reveal
The aggregate imbalance	builds up as reveals land
P_{open} and every fill	the cross
Deposits and withdrawals	always

The API never receives the salt, and the reveal transaction is a zeroed template that the client fills in, so the operator learns an order when the reveal lands on chain.

How long the imbalance is public

```
exposure = capture_time − first_reveal_time
         ≤ (open − reveal_start) + open_delay + capture latency
         = 30 min + 30 s + a few seconds                    (default parameters)
```

With defaults that is at most about 30.5 minutes. It is also measured: the worker records `firstRevealAt`, `lastRevealAt` and `capturedAt` for every lull, and the API reports `exposure.imbalancePublicSeconds`.

Run	Reveal window	First reveal → P_{open} capture
`check:e2e`, compressed timeline	18 s	34 s in the recorded run; 34–40 s across three runs
Production defaults	29 min	at most about 30.5 min, recorded per lull

No real lull has crossed on mainnet, so no production measurement exists yet. The window can be shortened, since it is two worker settings fixed per lull. At one minute the exposure falls to about 1.5 minutes, at the cost of more missed reveals and forfeited bonds. Half an hour is the compromise.

Why the leak is less damaging than it looks

- **P_{open} does not come from a pool.** It is Pyth's aggregate of the underlying's regular session. Seeing a Lull buy imbalance at 09:05 ET does not let anyone move that price against Lull. They would have to move Apple's own opening prints.
- **Fills do not depend on reveal order**, so knowing the book does not let anyone jump a queue.
- **Exchanges publish similar information.** A watcher can trade pools, perpetuals or other venues on a Lull imbalance, much as traders use the imbalance information exchanges publish before their opening auctions. Lull's is earlier and, in Stage 1, per order.

Remaining leaks

Leak	Mitigation in place	Further mitigation
The free option: a committed user watches reveals and declines to reveal	the bond is forfeited	size the bond to the option's value; Stage 2 removes the option
A deposit during a lull signals intent	balances persist across lulls; commits move no tokens	standing balances; batched deposits
Commit count and timing per wallet	uniform bond; user-chosen ticket ids	relayed commits with a different fee payer (possible, not built)
LPs see the imbalance before revealing	LPs commit before the window, so they can only decline and forfeit	Stage 2
A relayer sees a delegated order early	opt-in, per ticket	Stage 2

Stage 2: encrypted orders with Arcium (designed, not built)

Arcium runs multi-party execution environments (MXEs) that evaluate confidential instructions over encrypted inputs. Each Arx node holds a secret share, and no single node sees plaintext. The project's design notes describe Arcium as running on Solana mainnet as an alpha. Stage 2 uses it so that nothing about individual orders, or the aggregate imbalance, is public before the cross.

Flow

```
lull:     submit_encrypted(order ciphertext)   ── the Lull program stores the ciphertext and locks
          (x25519 to the MXE key, nonce)          a maximum balance chosen by the user
open+δ:   capture_price (unchanged, public)
          queue_computation(cross_mxe, [all ciphertexts, P_open, px, quotes])
Arcium:   decrypt shares ⟶ classify, tally, heavy side, LP allocation (the math.rs logic)
          ⟶ encrypted per-order fills + public aggregate totals
callback: cross_callback(totals, per-order encrypted fills, proof of correct execution)
claim:    the owner decrypts its own fill client-side; the program settles against the callback outp
```

Program changes

1. **Commit becomes submit.** The ticket stores a ciphertext of the order or quote, and the reveal phase disappears.
2. **Locking without revealing.** The program cannot lock an amount it does not know, so the user locks a public cap from rounded buckets, such as "up to 5,000 USDC". The computation enforces that the encrypted amount does not exceed the cap and treats a violation as zero.
3. **The cross runs in MPC.** `classify`, `Tally::add`, `cross` and `settle_order` are already integer functions without floating point or data-dependent loops, and the eight-quote sort becomes a fixed sorting network.
4. **Callback.** Arcium returns public totals and per-ticket results encrypted to each owner, and claim reads them from the callback account.
5. **Bonds remain as spam protection.** With no reveal phase there is no option to price.

What stays public

That a wallet participates and its cap bucket; the crossed totals and LP fills after the open, comparable to a post-auction report; and deposits and withdrawals, unless balances also become encrypted in a later step.

Trust and cost

Confidentiality holds unless Arx nodes collude beyond the MXE's threshold, and correctness rests on Arcium's verification of the computation. The Pyth price stays public. Each lull costs one computation of linear size plus an eight-element sort, one fee per market per lull. Markets that do not opt in keep Stage 1. Stage 2 depends on Arcium's production readiness, the port of the math to its circuit language and an audit. It is an open item in `docs/LAUNCH.md` and is not scheduled.

14

Security and threat model

Scope

This chapter covers the parties Lull depends on, what each can do, and the attacks the design considers, for the program as it stands. The program has not been audited. `docs/LAUNCH.md` makes an audit a prerequisite for real funds, with emphasis on the rounding in `math.rs`, `claim`, `tally` and the account constraints.

The assets at stake are the funds in the market's vaults, the bonds held in tickets, the order secrets held by clients, the integrity of P_{open} and the multiplier, and liveness: the ability of every lull to cross, or to cancel and refund.

Parties and their powers

Party	Can	Cannot
User	deposit and withdraw its own free balance; commit; reveal; claim	move another user's balance; change a sealed order
Anyone	reveal a ticket whose salt it holds; forfeit, capture, tally, finalize, claim; cancel after the deadline; close a finished lull, whose rent goes to the market authority	alter a revealed order; redirect a claim or the rent; capture outside the window
Market authority	publish lulls; change parameters within bounds; cancel before pricing	move user funds; change a sealed order; set the price or the cross
Upgrade authority	deploy new program code; open markets (`init_market` requires it)	nothing is out of reach of new code
Pyth publishers, Wormhole guardians	determine and sign the price	bypass the window, feed and verification checks
xStock issuer	move, pause or freeze AAPLx in any account, the vault included	affect USDC
API and worker operator	build unsigned transactions; crank; publish lulls with the authority key	sign for users; see salts; move funds

The market authority

The market authority is the strongest party inside the current program.

Publishing lulls. The program enforces `close < reveal_start < reveal_end ≤ open` and `now < reveal_start`, but it cannot check the times against Pyth's schedule. A wrong open time would make P_{open} a price from the wrong moment, and because the push account keeps updating after the close, an open time set inside a closed market could be priced from an off-hours update. The

defence is transparency: the times are on chain before anyone commits, and the API serves the times derived from Pyth.

Changing parameters. `update_params` accepts any parameters that satisfy `max_lp_spread_bps < 10,000`, `max_conf_bps ≤ 10,000`, `twap_min_samples ≥ 1`, `max_price_lag_secs ≥ 1`, `open_delay_secs ≤ 3,600`, and positive minimums. These bounds are wide, and changes apply at once: new minimums and spread caps to later reveals, new pricing rules to any lull not yet priced. Only the bond is fixed per ticket. An authority that raised a minimum or lowered the spread cap during a reveal window could make sealed tickets unrevealable, and their bonds would go to the treasury it also chose.

Cancelling. The authority can cancel a lull whenever it is `Pending` or `Sampling`. That includes the seconds between the open plus delay and the capture, and the sampling period of a TWAP. An authority watching the Pyth price could void a cross it disliked. It could not choose the price, and every revealed lock would be refunded.

What it cannot do. Only `withdraw`, signed by the owner, sends tokens out of a vault. The authority cannot move funds, change a sealed order, set P_{open} or alter a cross. The program also has no instruction to change a market's authority or treasury, both fixed at `init_market`, so moving either to a multisig or to governance needs a program upgrade (chapter 18).

The upgrade authority

Whoever holds a Solana program's upgrade authority can replace its code, and new code can do anything the program's accounts allow, including moving vault funds. This power sits outside the program's rules. `docs/LAUNCH.md` recommends moving it to a multisig after deployment, and chapter 18 describes the path to governance with a timelock, so that users can see an upgrade coming and withdraw first.

Threats and mitigations

Threat	Mitigation	Residual
Changing a sealed order	SHA-256 commitment over every field	none known
Replaying a commitment in another lull or wallet	the owner and the lull are in the preimage	none known
Guessing an order from its hash	256-bit random salt	depends on the client's random source
The operator reading orders early	the salt stays on the client; the reveal template is zeroed and filled locally	a user who shares an export or uses a relayer trusts that party
Front-running a reveal	fills ignore reveal order; P_{open} cannot be moved from a pool	the imbalance is public for up to about 30.5 min
An early, stale or off-hours price	publish time must lie in `[open + delay, open + delay + lag]`	relies on correct open times
A forged or unverified price	receiver ownership, discriminator, full verification, feed id	trust in Pyth's publishers and the guardian set
A cranker choosing its price	30 s window by default; the honest crank captures first; the exactness flag	the price movement within the window
An uncertain price	25 bps confidence gate by default, with a TWAP fallback	a confident, verified, wrong price would be used
No price, or no crank	every step is permissionless; cancellation after the deadline refunds all	no trade at that open
Creating tokens by rounding	debits round up, credits down; property tests	at most one raw unit of dust per order
Overflow	checked arithmetic with 128-bit intermediates (`MathOverflow`)	totals are 64-bit raw amounts, far above any realistic book
Wrong token or vault in a deposit	mint and vault must be the market's (`WrongMint`, `WrongVault`); `transfer_checked`; credit what arrived	none known
Double claim or double tally	claim closes the ticket; tally skips tallied tickets and checks the ticket's lull	none known
A wrong mint for the multiplier	`capture_price` requires the market's base mint	none known
Taking a market first after deployment	`init_market` requires the program's upgrade authority (`NotUpgradeAuthority`)	trust in the upgrade authority's key

Threat	Mitigation	Residual
Closing a lull with tickets outstanding	`close_lull` requires a finalized or cancelled lull with `claimed + forfeits == commits` (`TicketsOutstanding`)	none known

Known weaknesses

These follow from the current design and are not yet mitigated.

Quote-slot crowding. A lull takes eight quotes in reveal order, and the program sets no minimum quote size. An adversary could reveal eight tiny quotes at the start of the window, so that every later LP reveal fails with `QuoteSlotsFull` and forfeits its bond. The adversary's own bonds return at claim, so its cost is fees and a few raw units locked. Tiny quotes at the maximum spread would also raise `s_hi` and push limit orders near P_{open} into class A. A minimum quote size, more slots or LP admission rules would help. None is implemented.

Authority discretion. The authority's cancellation right around the open, and its ability to change parameters inside a lull, are described above. A timelock longer than any lull would remove the second (chapter 18).

No escape after pricing. `cancel_lull` accepts only `Pending` or `Sampling` lulls. If `tally`, `finalize` or `claim` failed permanently on a priced lull for an unforeseen reason, its revealed locks would stay locked until an upgrade unwound them. The property tests and the audit are the defence.

Weaknesses already addressed

Market creation. A market's address depends only on its two mints. If `init_market` accepted any signer, whoever first initialized AAPLx/USDC after a deployment would become its authority and set its treasury and parameters. `init_market` now requires the program's upgrade authority. The program reads that authority by hand from its own program data account, which kept the binary at 497 KB, and rejects any other signer with `NotUpgradeAuthority`. `check:e2e` checks that another key is rejected. Operationally, `scripts/mainnet-init.ts`, signed by the deployer, must run before the upgrade authority moves to a multisig. After that move, only the multisig can open a market.

Rent of settled lulls. Lull accounts used to stay open after settlement, each holding about 0.0049 SOL of the authority's rent at mainnet rent. `close_lull` now returns that rent once every ticket is claimed or forfeited, and the worker calls it on a later pass. `check:e2e` checks that one more crank pass closes the settled lull and returns its rent, 6,702,480 lamports at the local validator's default rent.

Oracle trust

The cross depends on one feed. Lull checks verification, feed, window and confidence, but does not compare the price with any other source. The DEX series in the drift analytics is labelled as not an oracle and never feeds the cross. A second opinion from an xStock pool is exactly the kind of price Lull exists to avoid, so the design accepts the single-oracle dependency deliberately.

Issuer controls

Through its permanent delegate, pause, freeze authority and confidential-transfer extensions, the AAPLx issuer can move AAPLx out of any account including the vault, pause transfers so that deposits and withdrawals fail, and freeze accounts including the vault. The cross and claims only change records inside the program, so they keep working during a pause, but tokens cannot leave. Lull discloses these powers and cannot mitigate them.

Clients and operations

A lost salt costs the bond and nothing else, because an unrevealed order never locked funds. A leaked export reveals the order early and lets the holder reveal it, but not change it or move funds. Every transaction is signed by the user's wallet. The web app supports Wallet Standard wallets and does not offer hardware wallets that connect over USB.

Only the worker holds keys: the market authority key, if it publishes lulls, and a separate, low-balance crank key. The API holds none. RPC URLs, which can contain keys, and the Pyth Pro key are never logged. With no keys at all, the worker only syncs the calendar, indexes and samples prices. That is how the read-only mainnet deployment runs.

15

Mechanism properties and incentives

A price-taking auction

Most auction analysis asks how bidders shade their bids to influence the price. In Lull no order or quote can change P_{open}, so that channel is gone. What remains are four decisions: whether to commit and how many tickets; what each ticket contains; whether to reveal each ticket; and, for LPs, which spread to quote.

Who bears the imbalance

At an exchange's opening auction the imbalance moves the clearing price, and everyone pays the moved price. At Lull the light side fills in full at exactly P_{open}, and the heavy side bears the whole imbalance: through **rationing**, since the matched volume is shared pro rata, and through **spread**, since class-B orders fill part of the rest against LP quotes. The side that creates the imbalance pays to have it absorbed. Nobody on the light side is made worse off by the size of the other side.

Pro rata instead of time priority

With one price, the only question is how to ration the heavy side. Time priority would reward early reveals, which are exactly the information Lull withholds as long as it can. Under pro rata, a fill is independent of when the order was revealed.

Pro rata has a known weakness: a participant who expects rationing can overstate its size to get the fill it wants. Three things limit this in Lull. Every order locks its full amount, so overstating needs real funds. If the side turns out light, the whole order fills. And the book is sealed at commit time. A participant who watches the imbalance during the window cannot change its size. It can only decline, and pay the bond.

Why classes A and B

The natural rule, letting each limit order take LP liquidity only at prices within its limit, is circular. The LP prices that clear depend on how much class-B volume reaches the LP leg, which depends on which limit orders are admitted, which depends on the clearing LP prices. Lull breaks the circle with `s_hi`, the widest revealed spread on each side, which is known before the tally. The rule has four properties:

- **Limits are always safe.** A class-B order is within its limit whichever quotes fill, up to two raw units of rounding.
- **The tally is linear and batchable.** A class depends only on the order's own limit, P_{open} and `s_hi`.
- **It is monotone.** A market-on-open order is always class B, and a tighter limit can only move an order from B to A, or from A to None.
- **It has a cost.** Some limit orders just beyond P_{open} take no LP liquidity even when a cheap quote would have satisfied them, like the seller in chapter 10 who kept 28 shares while an LP bid $99.80 against its $99.60 limit.

A participant who wants LP liquidity for certain can send a market-on-open order, whose worst price is still bounded by the spread cap: `P_open × 1.02` for a buy and `P_open × 0.98` for a sell with the default 200 bps.

The free option and the bond

In Stage 1 a committed participant can watch the imbalance form and then decide whether to reveal. It can straddle (commit a buy and a sell and reveal one), size after the fact (commit several sizes and reveal one), or, as an LP, withdraw liquidity when the imbalance runs against it. Each unrevealed ticket forfeits its bond, which is the option's price.

The bond is uniform, 0.01 SOL by default, and that has two sides:

- **It is necessary for hiding.** The bond is visible at commit. A bond proportional to size would reveal size, and one that differed between orders and quotes would reveal the kind. Only a uniform bond, or at most a coarse bucket, keeps the commit uninformative.
- **It is weak against large orders.** For a large order, seeing the book before deciding can be worth far more than a flat bond.

The bond is therefore a floor on the cost of optionality, not a full price. `docs/PRIVACY.md` suggests sizing it to the option's expected value, per market and within the need to keep it uniform. The structural fix is Stage 2, which has no reveal phase and so no option.

The LP version matters most. An LP that declines to reveal when the imbalance is large withdraws liquidity exactly when it is needed. The bond makes that costly without preventing it, and `docs/LAUNCH.md` expects onboarded market makers to have documented quoting obligations.

LP competition

The LP leg is discriminatory, or pay-as-bid: quotes fill cheapest first, each at its own spread. An LP has no reason to quote below the spread it needs, because a lower quote lowers its own price, and LPs compete on priority. A uniform-price LP leg would pay every filled quote the marginal spread. It would reward inframarginal LPs, but it needs the margin computed before settlement and invites shading towards it. Pay-as-quoted keeps the LP leg a single pass over at most eight sorted quotes, with every fill checkable against its quote.

Relayers and cranks

Relayers reveal for offline users and learn their orders early. Cranks capture the price and drive the cross, and pay the fees. Today the only crank is Lull's worker, paying from its crank key. Neither role earns anything from the protocol or answers for failure beyond reputation. The staking registry of chapter 19 would let both stake $LULL, charge a service fee and be slashed for failing an accepted duty. It is a design.

Cancellation as the safe default

When the pricing rules cannot produce a trustworthy price, the lull cancels and everyone is refunded. A participant never trades at a stale price, a pool price or a price from the wrong moment. For an order whose purpose is "trade at the open", a missed open is worse than a fill but better than a fill at the wrong price.

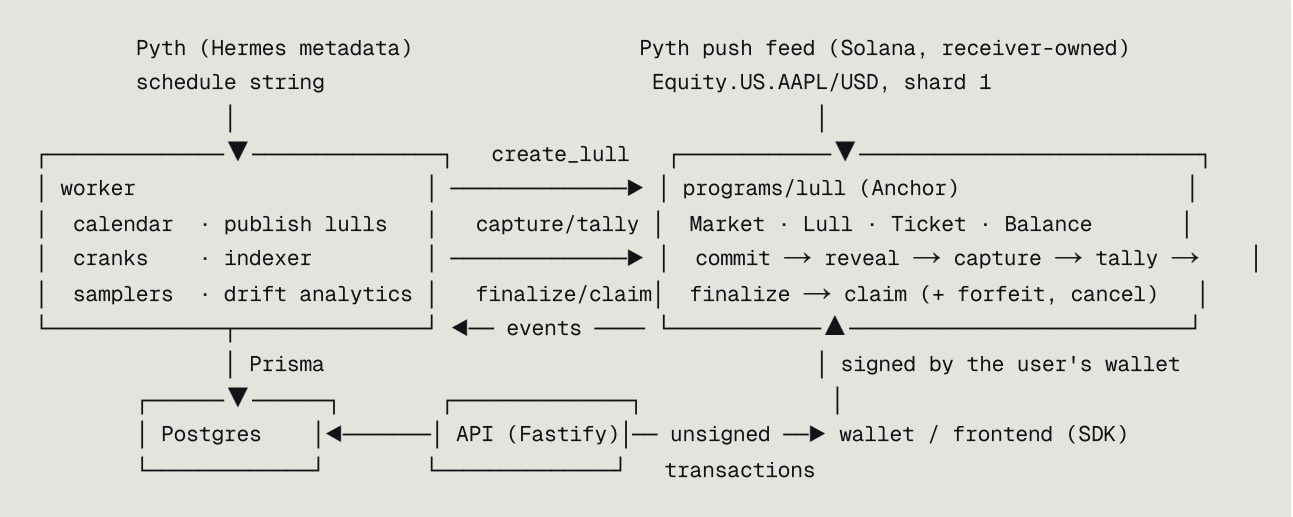
Summary

Property	Holds	Mechanism
One price for all matched volume	yes	P_{open} for the light side and the matched part of the heavy side
Bounded worst price	yes	limits for LOO; the spread cap for MOO
Fill independent of reveal time	yes	pro rata
No token created	yes	vault-favoring rounding, property-tested
Price cannot be moved from inside Lull	yes	external oracle
No free option	no, in Stage 1	priced by a uniform bond; removed in Stage 2
No pre-cross leak	no, in Stage 1	bounded to about 30.5 min and measured; removed in Stage 2
Guaranteed LP liquidity	no	depends on LPs revealing

16

Implementation

Architecture



The repository holds the program (`programs/lull`), the SDK (`sdk/src`), the worker and API (`server/src`), the Prisma schema and migrations, the gates (`tests`, `scripts`) and the web app (`web`).

The program

The program is written in Rust with Anchor 0.31.1, and its compiled binary is 496,816 bytes (497 KB). Its IDL (`sdk/src/idl/lull.json`) lists 15 instructions (appendix A), 4 account types, `Market` (348 bytes), `Lull` (835), `Ticket` (154) and `Balance` (105) (appendix B), 14 events and 32 error codes, numbered 6000 to 6031. Three modules carry the logic without Anchor types, so they run in host unit tests (`cargo test`, part of `check:cross`):

- `math.rs`: `px_from_price`, the conversions, `cmp_limit`, `classify`, `Tally`, `cross` and `settle_order`;
- `pricing.rs`: the P_{open} state machine, `step(sampling, observation, rules, now)`;
- `oracle.rs`: hand-written readers for `PriceUpdateV2` and the Scaled UI Amount extension, which keep the Pyth SDK and a newer Token-2022 crate out of the build.

The same approach reads the program's upgrade authority, which `init_market` requires: a few bytes of the program data account are parsed by hand rather than through a bincode decoder, which kept the binary at 497 KB.

The program id is `8xf4NfkWse8YRYghY1femurYsZaBaSEoNgssweT6oGWd`. Its keypair, kept in `target/deploy/`, is the only way to deploy to that address.

The SDK

The SDK is shared by the worker, the API, the gates and the web app, and it runs in the browser. `constants.ts` holds program ids, mints, feed ids and market specs; `pda.ts` the addresses;

`commitment.ts` commitments and salts; `client.ts` instruction builders, readers, `DEFAULT_PARAMS` and the reveal template; `matching.ts` the mirror of `math.rs` and `simulateCross`; `units.ts` the Scaled UI parsing and conversions; `pyth.ts` the `PriceUpdateV2` decoder; `receiver.ts` Hermes accumulator parsing and hand-built receiver instructions (the Pyth receiver SDK's dependency tree did not load in the project); and `schedule.ts` the schedule parser.

The worker

Loop	Default interval	Does
calendar	10 min	Hermes schedule → snapshot and lull rows for 14 days
publish	30 s	authority only: `create_lull` for lulls closing within the hour
crank	3 s	forfeit; capture (exact via Hermes with a Pyth Pro key, else the push account); tally; finalize; claim; cancel after the deadline; on a later pass, `close_lull`, after saving the LP quotes and fills to the database
indexer	4 s	program signatures → Anchor events → `ChainEvent` rows and projections, each applied once
pyth	15 s	push account → `PriceSample`
dex	60 s	Jupiter Price API v3 `usdPrice` per share → `DexSample`, labelled as not an oracle
drift	5 min	per lull: prior close, first price after the open, DEX first, last, min and max, drifts in bps; with a Pyth Pro key, missing Pyth prices come from Benchmarks (the last update at or before the close, and the first at or after the open, found by search)

Every crank step re-reads the chain before acting, so restarts are safe. Without an authority or crank key the worker only syncs, indexes and samples, which is safe to point at mainnet.

The API

A Fastify server over Postgres. Reads come from the indexer's projections plus live account reads. Transaction endpoints return unsigned base64 transactions with a fresh blockhash.

Endpoint	Returns
`GET /health`	database, RPC and indexer status
`GET /v1/markets[/:slug]`	markets with parameters, Scaled UI configuration, latest Pyth price and schedule
`GET /v1/markets/:slug/calendar`, `/lulls/current`, `/drift`	the calendar, the current lull, the drift series
`GET /v1/lulls/:address`	a lull with its LP quotes and fills, served from the database once the lull account is closed
`GET /v1/lulls/:address/tickets`	every ticket of a lull, in commit order
`GET /v1/lulls/:address/series`	the Pyth and DEX price series through a lull, each thinned to at most 600 points
`GET /v1/owners/:owner/balances`, `/tickets`, `/fills`	a user's balances, tickets and fills
`GET /v1/units`	unit definitions
`POST /v1/tx/deposit`, `/withdraw`, `/commit`, `/reveal`, `/claim`	unsigned transactions; the reveal is a zeroed template

`docs/FRONTEND-INTEGRATION.md` is the contract. The database has nine models (`Market`, `ScheduleSnapshot`, `LullWindow`, `Ticket`, `ChainEvent`, `IndexerCursor`, `PriceSample`, `DexSample`, `LullDrift`), with raw amounts stored as decimal strings.

The web app

`web/` is a Next.js site with the app and the user documentation: the lull clock drawn from the real schedule, the current lull and its phase, an order ticket that seals orders client-side, sealed orders with reveal, claim, export and import, balances, fills, and a drift panel that charts each lull's series. A cross replay steps through a crossed lull. While no lull has crossed on mainnet, it plays the cross that `check:e2e` recorded on a local validator and labels it as such. The app connects to Wallet Standard wallets and does not offer hardware wallets that connect over USB. It asks each visitor once whether they are a US person, and shows US persons a notice instead of the app.

For testing the app end to end, `npm run dev:stack` runs a persistent local Lull: Postgres, a validator with the program, the real mints behind a local faucet and, with a Pyth Pro key, the mainnet Pyth receiver, plus the worker and the API. It opens a short lull with a house sell. `npm run dev:fund` gives any wallet local SOL, USDC and AAPLx.

Deployment

One Dockerfile serves both backend roles. It runs `prisma generate` at build time and, at start, `prisma migrate deploy`, then the API on `$PORT` or the worker when `LULL_ROLE=worker`. Migrations run at start because the build cannot reach Railway's private network.

Live today. The Railway project `lull` runs Postgres, `lull-api` (https://lull-api-production.up.railway.app) and `lull-worker`, read-only against mainnet, with no keys on the worker. The API serves the market, the Pyth-derived calendar, live push prices and drift. Because the program is not on mainnet, `params` is `null`, lulls stay `scheduled`, and transaction endpoints answer 404 ("lull is not on chain"). On 2026-09-26, `/health` reported the database and RPC as healthy and no indexed slot, as expected with no program on chain.

Going live requires, per `docs/LAUNCH.md`, the owner's approval, a program deployment (about 2.53 SOL of program rent at current mainnet rent, plus an equal buffer returned after upload, so about 5.1 SOL to fund), market initialization (about 0.009 SOL) signed by the upgrade authority, a keyed RPC endpoint, the authority, crank and Pyth Pro keys on the worker, and a shorter indexer interval than the current 120 seconds.

Gates

Command	Proves	Result
`check:schedule`	the parser against 28 pinned schedules and Hermes' live `market_hours`; AAPL holidays, early closes, both DST switches; empty ranges ending at 0000	11/11 pass
`check:cross`	matching invariants (TypeScript properties and a Rust randomized test); the P_{open} state machine	pass
`check:e2e`	commit → reveal → forfeit → worker-cranked capture, tally, finalize, claim → withdraw → `close_lull`, against real mainnet accounts, with exact equality to the SDK; a market opened by a key other than the upgrade authority is rejected	pass, about 85 s
`check:api`	worker, API and a fresh database against the local validator: publishing, indexing, every endpoint, validation, and the drift backfill with a Pyth Pro key	pass, about 40 s
`check:exact`	against the mainnet Pyth receiver and Wormhole receiver with the live guardian set: the exact P_{open}, and the TWAP path through three real verified updates	pass, about 75–80 s; needs a Pyth Pro key

The end-to-end gate runs in two validator phases. A cloned account is frozen at its clone time, but P_{open} must be published after the open, which comes after the reveals. Phase A runs deposits, commits, reveals and a forfeit. Phase B restarts from phase A's exact state plus a fresh clone of the Pyth account, published after the open, and the worker's crank drives the cross. The gate patches the cloned mints' mint authority to create test balances. Otherwise the mints are the real ones.

Known limitations

- The end-to-end gate reaches only the direct pricing path, because a static clone gives one sample. `check:exact` covers the TWAP path through the instruction instead.
- A lull holds at most eight LP quotes.
- The indexer polls rather than subscribing.

17

Empirical results

What has been measured

No real order has crossed through Lull, because the program is not on mainnet. The evidence is of three kinds: gate runs on a local validator holding real mainnet accounts, property and unit tests, and live read-only data from the Railway deployment. The e2e figures come from `docs/e2e-sample-report.json`. The live figures were fetched on 2026-09-26, when the server clock read 2026-09-25 23:46 UTC.

The end-to-end sample cross

Setup

`npm run check:e2e` ran on 2026-09-24 against a validator loaded with the real AAPLx and USDC mints and the Pyth push account for `Equity.US.AAPL/USD`, cloned from mainnet. To fund the test wallets, the gate patched the cloned mints' mint authority. The mints are otherwise unchanged.

The run compresses a lull into about a minute of validator time: an 18-second reveal window, and an open 4 seconds after it. The timing parameters were overridden (an open delay of 2 s, a maximum lag of 180 s, a 30-second TWAP window with 2 samples), and the rest were defaults. The "open" is therefore a local instant, not 09:30 ET. The P_{open} it captured is a real, fully verified Pyth update, published at 21:12:55 UTC on 2026-09-24, after that day's close.

The book

Actor	Ticket	Amount	Limit
alice	buy, market on open	2,000 USDC	none
carol	buy, limit on open	500 USDC	95% of the pre-run Pyth price
erin	buy, limit on open	300 USDC	110% of the pre-run Pyth price
bob	sell, market on open	3 shares = 299,022,491 raw base	none
lp1	quote, 30 bps	max base 12 shares, max quote 1,000 USDC	
lp2	quote, 15 bps	max base 1.5 shares, max quote 0	
dave	buy, never revealed	100 USDC	none

All seven tickets were committed without moving tokens. The LP reveals were relayed: the market authority's wallet paid their fees.

The price

P_{open} was `price = 33,575,101` at `expo = -5`, or $335.75101 per share, with a confidence of $0.04899, about 1.46 bps of the price and well within 25 bps. The source was direct, and `price_exact` was false because the capture came from the push account. The multiplier read from the mint was 1,003,269,012,540 (× 10^{12}), and `px` was 3,368,485,842,620.

The cross

Classify. Both quotes offer base, so `s_hi_ask = 30 bps`. carol's limit was below P_{open}, so carol is class None. alice (market on open) and erin (limit far above P_{open} × 1.003) are class B.

Heavy side. Eligible buys total 2,300,000,000 raw quote. bob's 299,022,491 raw base is worth `⌊299,022,491 × px / 10¹²⌋ = 1,007,253,027` raw quote. Buys are heavy, the imbalance is 1,292.746973 USDC, and bob sells in full.

LP leg. lp2 is cheaper and fills first:

```
lp2:  px_k = ⌈px × 10,015 / 10,000⌉ = 3,373,538,571,384
      cost of 149,511,245 raw base = 504,381,952 raw quote ⟶ fills in full; left = 788,365,021
lp1:  b = ⌊788,365,021 × 10¹² / px_k⌋ = 233,341,339 raw base, cost 788,365,018 raw quote
      left = 3 raw quote, refunded to class B
```

Class B's totals are a debit of 2,299,999,997 raw quote, a credit of 681,875,075 raw base, and a weight of 2,300,000,000.

Settlement

Actor	Paid (raw)	Received (raw)	Paid	Received	Per share
alice	1,999,999,998 quote	592,934,847 base	1,999.999998 USDC	5.94873158 shares	$336.2061
erin	300,000,000 quote	88,940,227 base	300.000000 USDC	0.89230973 shares	$336.2061
carol	0	0	none	none	500 USDC refunded
bob	299,022,491 base	1,007,253,027 quote	2.99999999 shares	1,007.253027 USDC	$335.7510
lp2	149,511,245 base	504,381,952 quote	1.49999999 shares	504.381952 USDC	$336.2546
lp1	233,341,339 base	788,365,018 quote	2.34104134 shares	788.365018 USDC	$336.7583
dave	0.01 SOL bond forfeited				never revealed

Shares are raw base × multiplier, truncated to eight decimals. The results check against the design:

- **The light side got P_{open}.** bob's $335.7510 per share is P_{open} up to rounding.
- **Each LP got its quote.** lp2's $336.2546 is P_{open} × 1.0015, and lp1's $336.7583 is P_{open} × 1.003.
- **Class B paid one blended price.** alice and erin both paid $336.2061, between P_{open} and the LP prices.
- **The limit order below P_{open} did not trade**, and carol's 500 USDC was refunded.
- **The lull balanced.** Sellers and LPs delivered 681,875,075 raw base, and buyers received 681,875,074. Buyers paid 2,299,999,998 raw quote, and sellers and LPs received 2,299,999,997. The vault kept one raw unit of each token.
- **The unrevealed ticket forfeited** its 10,000,000 lamports to the treasury.

The gate also asserted that four invalid actions fail: a reveal before the window (`NotInRevealWindow`), a reveal with a changed amount (`CommitmentMismatch`), a withdrawal of locked funds (`InsufficientBalance`), and a capture before the open plus delay (`TooEarly`). It asserted that the captured price and publish time equal the Pyth account's, and that `px`, the heavy side, the LP totals and every fill equal the SDK's `simulateCross` to the raw unit. Finally, it checked that the worker settled and closed all six revealed tickets, that nothing stayed locked, and that alice and bob withdrew their proceeds.

The gate has since gained three checks. It rejects opening the market from a key that is not the program's upgrade authority (`NotUpgradeAuthority`). After the cross, one more crank pass must close the settled lull and return its rent to the authority: 6,702,480 lamports at the local validator's default rent, which corresponds to about 0.0049 SOL at mainnet rent. And the gate writes the cross in the API's lull shape (`.e2e/replay.json`), which the site's cross replay plays, labelled as a local-validator recording, while no lull has crossed on mainnet.

The imbalance was public for 34 seconds, from the first reveal to the capture. `docs/PRIVACY.md` reports 34 to 40 seconds across three runs. Under production defaults the bound is about 30.5 minutes.

The pricing check: exact P_{open} and TWAP

`npm run check:exact` loads the mainnet Pyth receiver and Wormhole receiver programs into a local validator, with the receiver's configuration, its fee treasury, the current guardian set and the two mints. It opens two markets with default parameters: AAPLx/USDC, priced by `Equity.US.AAPL/USD`, and a test market with two fresh mints, priced by `Crypto.SOL/USD`, which publishes around the clock. It needs a Pyth Pro key and takes about 75–80 seconds.

Scenario 1: the exact P_{open}. An empty lull opens seconds after creation, on AAPLx/USDC while Pyth publishes AAPL and on the test market otherwise. After the open plus delay, the worker's crank finds the first Hermes update at or after that instant by the search of chapter 9, verifies the VAA's guardian signatures, posts the update and captures it in one transaction, and closes the posted accounts. The scenario passes if the lull is priced directly with `price_exact = true`, at exactly that update's price and publish time, with no posted account left.

Scenario 2: the TWAP path. On the test market the parameters change to an open delay of 2 s, a confidence limit of 0 bps, a 10-second TWAP window and 3 minimum samples. With a zero limit no single price qualifies, so the lull must sample. Three real Hermes updates, at the start and 5 and 10

seconds after the first, are each posted with full verification and captured. The scenario passes if the lull is priced by TWAP at the integer mean of the three prices, dated by the last, with `price_exact = false`, if the worker's crank then finalizes it, and if every posted account is closed.

In one run, on 2026-09-26 at 00:18 UTC, AAPL was not publishing, so both scenarios ran on the SOL/USD test market:

Scenario	Update	Price (expo −8)	Published (UTC)
Exact	the first update after open + delay	12,197,872,307 ($121.97872307)	00:18:13, previous update 00:18:12
TWAP	sample 1	12,198,698,694	00:18:31
TWAP	sample 2	12,198,785,018	00:18:36
TWAP	sample 3	12,198,377,387	00:18:41
TWAP	P_{open}, the integer mean	12,198,620,366 ($121.98620366)	dated 00:18:41

The gate confirmed `price_exact` for the exact capture: its previous update, at 00:18:12, was published before the open plus delay, and it was published at or after it. The mean is `⌊(12,198,698,694 + 12,198,785,018 + 12,198,377,387) / 3⌋`. The repository keeps no report file for this gate, so the figures are from the run's output.

Calendar and math

`check:schedule` passes 11 of 11 tests: 28 pinned schedule strings reproduce Hermes' `market_hours`, the live comparison covers more than 1,800 feeds, and every AAPL case in chapter 6 holds. `check:cross` passes its four property tests on 4,000 books each, the Rust test on 3,000 books, two fixtures with exact expected values, the commitment test, and the state-machine tests for the direct path, exactness, the TWAP fallback and every rejection.

Live drift figures

Chapter 4 gives the drift series as fetched. The columns it omits:

Lull close (UTC)	DEX last vs close	DEX last vs open	Largest DEX deviation from close	Samples
2026-09-24 20:00	−15 bps	−17 bps	51 bps	934
2026-09-25 20:00 (weekend, in progress)	−13 bps		36 bps	226

The API's summary covers three completed lulls: a mean absolute open-versus-close move of 16 bps, and, from the single lull with DEX data, 17 bps between the last DEX price and the open and a largest deviation of 51 bps. The 934 samples over the 17.5-hour night correspond to one a minute

for about 89% of it. On that one night, the pool price ranged about 87 bps while the underlying moved 2 bps from close to open. More lulls are needed before anything can be said about typical drift.

At the same fetch the market endpoint reported a latest push price of $341.37536 (`34137536` at expo −5), with a confidence of $0.02536, published at 23:46:41 UTC, three hours and 46 minutes after the close. The push account's updates after the close are why the capture rule requires a publish time at or after the open plus delay. The endpoint also reported `params: null`, since the market does not exist on mainnet.

What remains unmeasured

- Real lulls on mainnet: exposure times, reveal and forfeit rates, LP participation.
- How often the exact update wins in production against push-account captures by others.
- Drift over a meaningful number of lulls.

18

Governance

What there is to govern

The program enforces some rules for everyone, and a market's authority chooses the rest. Governance concerns the second part, and the power to change the program itself.

Power	Holder in the current design	Covers
Market authority	one key per market, set at `init_market`	publishing lulls; `update_params`; cancelling before pricing
Treasury	one account per market, set at `init_market`	receiving forfeited bonds
Upgrade authority	the deployer key, unless moved	replacing the program's code; opening markets
Market creation	the program's upgrade authority, once per mint pair	which markets exist

A market is a PDA of its two mints, so there is one market per pair, and `init_market` accepts only the program's upgrade authority as signer (`NotUpgradeAuthority` otherwise). The signer becomes the new market's authority. In practice, "listing" a market means the upgrade authority creating it with the intended feed and parameters, and adding it to the worker's and the app's configuration (`LULL_MARKETS`, `sdk/src/constants.ts`). Because the right to open markets follows the upgrade authority, it moves with that authority to a multisig and then to governance.

Where control sits today

The program is not deployed. As prepared for deployment, the **market authority** and the **treasury** are one key that the team controls: `scripts/mainnet-init.ts` passes the authority key as the treasury, and `docs/LAUNCH.md` runs it with the deployer key. The **upgrade authority** is the deployer key. `init_market` must be signed by it, so `docs/LAUNCH.md` runs the market initialization first and moves the upgrade authority to a multisig afterwards.

The market authority cannot move user funds, change a sealed order, set the price or alter a cross. It can publish lull times, change parameters within wide bounds, and cancel a lull before pricing, including in the seconds after the open plus delay (chapter 14). The upgrade authority is not bounded by the program at all.

What governance should control

Area	Items
Market parameters	open delay, maximum lag, confidence limit, TWAP window and samples, LP spread cap, bond, minimum sizes, and the fee rate up to its compiled cap once implemented
Markets	creating markets with their feed and push shard; retiring them from the worker and app
Fee flows and treasury	the split of buyback proceeds between burning and the treasury; treasury spending
Staking registry	minimum stakes, slashing amounts, the compensation cap, admission rules, once it exists
Upgrades	approving new program code, behind the timelock

Governance should not control what the program fixes for everyone: the commitment scheme, the rounding direction, the ordering of lull windows, the structure of the capture rule, and the absence of any authority over user funds. Those should change, if ever, only through an audited upgrade that governance approves.

Mechanism

The design uses SPL Governance (Realms) with $LULL as the governing token. Proposals carry executable instructions, such as `update_params` or `set_upgrade_authority`. Votes are token-weighted, with holders depositing $LULL into the governance program. A timelock separates a passed vote from its execution.

The timelock has a clear lower bound. A parameter change can make sealed tickets unrevealable or change what users committed to. To keep any change from landing inside a lull that users have already committed to, the timelock should be longer than the longest lull on the calendar. In the current AAPL schedule that is the 2026 Christmas lull, from the early close on Thursday 24 December at 13:00 ET to Monday 28 December at 09:30 ET: 92.5 hours. Weekends with a Monday or Friday holiday run to 89.5 hours. The same timelock gives users time to withdraw before an upgrade they object to. Durations, quorum and thresholds are not decided and will be proposed with the handover.

Handover in phases

1. **Team key.** At deployment one key holds the market authority, the treasury and the upgrade authority. It opens the first market while it still holds the upgrade authority, because `init_market` requires it.
2. **Team multisig.** The upgrade authority moves with the standard `set-upgrade-authority` command, and with it the right to open markets. The market authority and treasury cannot move yet: the current program has no instruction to change them, so this step needs a program upgrade that adds one, planned together with the fee (chapter 19).
3. **Governance with a timelock.** The market authority, the treasury and the upgrade authority move from the multisig to the governance program, one at a time, each in a transaction anyone can verify.

Whether any emergency power stays with a multisig is open. The program's only emergency lever is the authority's right to cancel a lull before pricing, for example during a trading halt, and it must act faster than a timelocked vote. A narrowly scoped cancel-only role is one option. It would need a program change and is not designed in detail.

Limits of token governance

- **Distribution determines control.** A bonding-curve launch gives tokens to whoever buys, and voting power can be concentrated.
- **Borrowed votes.** Tokens held briefly can swing a vote. Deposit-based voting and the timelock reduce this but do not remove it.
- **Low participation** leaves decisions to a few.
- **An upgrade is still code.** Governance can approve a flawed or malicious upgrade. The timelock gives users time to leave. It does not make the code safe.

For these reasons the handover is phased, and within the current code no authority, whether key, multisig or governance, can move user funds.

19

The $LULL token

Status

This chapter documents a design. At the time of writing nothing in the Lull program references $LULL, the protocol charges no fee, and no staking registry exists. The fee and the registry need program changes and audits before activation. Every part of the design may change, and some parts may never ship. Nothing here is an offer of the token or a promise about its price. Chapters 21 and 22 set out the risks and the legal position.

Summary

Item	Design
Total supply	1,000,000,000 $LULL, fixed
Launch	fair launch on a bonding curve
Team allocation, presale, private round	none
Vesting schedule	none, since there are no allocations to vest
Team holdings	only what the team's disclosed launch wallet buys on the curve, like any other buyer, plus whatever governance later allocates from protocol revenue
Uses	governance; staking for delegated roles
Value flow	a planned protocol fee whose revenue buys $LULL on the open market, burned by default
Yield	none: staking earns no protocol yield, and holding earns nothing

Launch

A bonding curve

$LULL is designed to launch on a bonding curve of the kind pump.fun popularized. Buyers purchase directly from a curve whose price is a fixed function of how many tokens have been sold: each purchase raises the price for the next, and each sale back lowers it. When the curve completes, the launch platform moves the accumulated liquidity into an automated market maker pool under its own rules, and the token trades like any other. The curve's parameters, including the share of supply sold on it, the completion threshold, the platform's fees and where liquidity goes, belong to the platform, not to Lull, and will be stated at launch.

No allocations

There is no team, presale, private, advisor or investor allocation, and no vesting schedule. Nobody receives $LULL except by buying it. The team takes part like any other buyer. Its launch wallet will be disclosed, and every purchase it makes is visible on chain. This paper does not state how much the team will buy. Beyond that wallet, the team will hold only what governance decides to allocate to it from protocol revenue, if anything, through a public proposal.

What holders can verify

Once the mint exists, anyone can check on chain the total supply of 1,000,000,000; that the mint has no mint authority, which a fixed supply requires; whether a freeze authority exists (the design calls for none); the disclosed team wallet's purchases and balance; and every buyback burn. The mint address is not part of this paper.

Use 1: the protocol fee and the buyback

The fee

Aspect	Design
Rate	`fee_bps`, a market parameter, set to 5 bps at activation
Cap	a maximum compiled into the program, so that no authority or vote can exceed it; its value will be published with the upgrade
Base	the crossed notional of each filled order: the USDC a buy pays, or the USDC a sell receives
Payers	filled orders; LP fills are not charged, so the fee does not widen quotes
Currency	the token the order receives: the xStock for a buy, USDC for a sell
Method	deducted from the order's credit at claim, `fee = ⌈credit × fee_bps / 10,000⌉`, which at the fill's own price is `fee_bps` of its notional; the rounding stays in the vault's favor
Limits	evaluated before the fee, as brokerage commissions sit outside a limit price
Not charged	unfilled remainders, refunds, cancelled lulls, forfeits

Accrued fees would stay in the vaults as a separately tracked amount per token, never part of any balance account. A governed instruction would move them to the account that executes buybacks, and every accrual and collection would emit an event. None of this exists in the current program. It needs an upgrade, and `docs/LAUNCH.md` already requires an audit before real funds.

A worked example

Applied to the end-to-end cross of chapter 17, which charged no fee, the planned fee would have been:

Order	Fill	Fee at 5 bps
alice (buy)	1,999.999998 USDC → 592,934,847 raw AAPLx	296,468 raw AAPLx (0.00297437 shares, about $1.00)
erin (buy)	300.000000 USDC → 88,940,227 raw AAPLx	44,471 raw AAPLx (0.00044616 shares, about $0.15)
bob (sell)	299,022,491 raw AAPLx → 1,007.253027 USDC	503,627 raw USDC (0.503627 USDC)
lp1, lp2	LP fills	none

The orders crossed about 3,307.25 USDC of notional, and the fee would have been about 1.65 USDC, part in AAPLx and part in USDC. This is a hypothetical calculation.

Buyback and burn

1. Accrued fees are collected from the vaults on a published cadence.
2. xStock proceeds are sold for USDC, or swapped directly, through public on-chain routes.
3. The proceeds buy $LULL on the open market.
4. By default every token bought is burned with the SPL burn instruction, which permanently reduces the supply.
5. Governance may redirect a share of the bought tokens, or of the fee revenue, to the governance treasury.

Until the handover, the team's multisig executes buybacks. After it, governance does, through a delegated executor or a program. Every collection, swap and burn is a public transaction, and a running report links each collection to its swaps and burns so that anyone can reconcile fees charged with tokens burned.

What the buyback is not

- **Not a distribution.** Holders receive nothing. The fee pays for open-market purchases, and the tokens bought are burned or held by governance.
- **Not a promise.** Its size depends entirely on volume through Lull, which is zero today because the program is not deployed. Governance can pause, resize, redirect or end it, and it may never start.
- **Not a claim.** No holder has any right to fees, the treasury or the buyback.

Scheduled buybacks are predictable and can be traded against. Splitting orders and using public routes helps, but slippage and front-running can reduce what the fee buys. Selling xStock proceeds depends on xStock liquidity and on the issuer's transfer controls, and xStock held in a fee account carries the issuer risks of chapter 14. Regulators in some jurisdictions may view a revenue-funded buyback as creating an expectation of profit. That is a material risk to this part of the design, and it may be changed or dropped for that reason.

Use 2: staking for delegated roles

Roles

- **Reveal relayers** reveal sealed tickets for users who will be offline in the reveal window. The program already allows this, because the salt authorizes the reveal.
- **Crank operators** capture P_{open}, preferably by the exact path, and drive tally, finalize and claim. Every step is already permissionless.

Neither role is accountable today. The registry would give them a way to offer their service with something at stake.

The registry

A separate registry program, or an extension of Lull, would list delegates:

- A delegate stakes at least a governance-set minimum of $LULL per role, and publishes its terms, including its service fee.
- A user who hands a ticket to a relayer records the delegation on chain, and the relayer signs an acceptance before the reveal window. The relayer first checks, off chain, that the preimage it received hashes to the ticket's commitment. By accepting, it vouches that it can reveal.
- Unstaking is delayed, so that a delegate cannot withdraw just before a duty falls due.

Duties, evidence and slashing

- **A missed delegated reveal.** If a ticket with an accepted delegation is still unrevealed at `reveal_end`, the relayer failed. A reveal can also fail for reasons outside the relayer's control, such as the owner's free balance being too low or the lull being cancelled. The planned upgrade therefore records a hash-verified reveal even when the lock fails. The relayer's duty becomes "publish a matching preimage inside the window", which the chain can check whether or not the owner funded the order.
- **Crank failures** are harder to prove. The chain cannot tell a crank that failed to capture from a Pyth outage with no eligible update. Crank operators are therefore not slashed automatically. A slashing proposal goes through governance with off-chain evidence, such as Hermes' record of an eligible update in the window.

A slashed stake goes **first to the affected user**, up to a compensation amount that governance sets to cover the forfeited bond and the user's costs. Any remainder is burned. The user's bond still goes to the treasury, as for any forfeit. Compensation never exceeds the slashed amount, so a user and a delegate who collude to fake a missed reveal lose at least as much as they recover.

Service fees, no yield

Delegates may charge users a service fee on terms they publish. The protocol pays delegates nothing, and staking earns no protocol yield: no emission, no share of fees, no reward. A delegate's income is its fee, and its stake is a bond against failure.

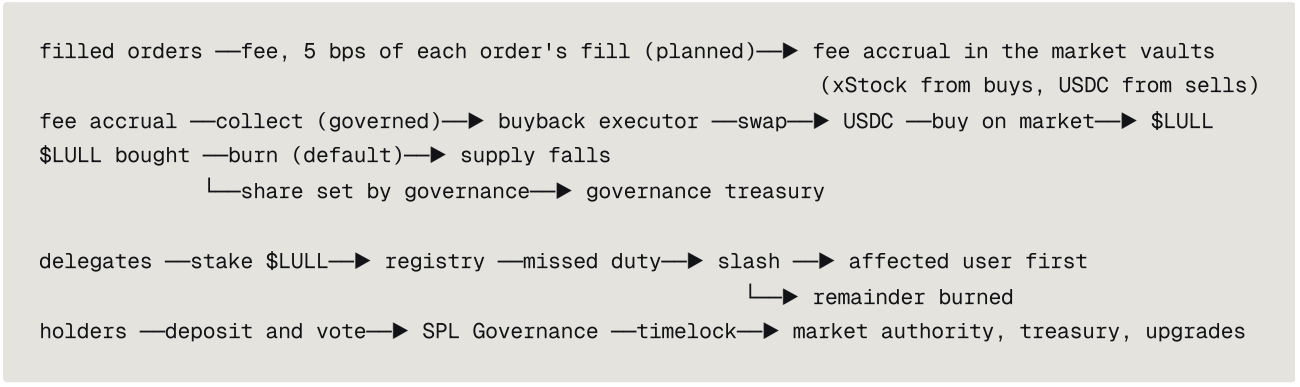
Open problems

A relayer still learns each order early, and staking makes it accountable for liveness, not discretion. Only Stage 2 privacy removes that. The encrypted handover of the preimage needs a specified format and client support. Stakes, slashing amounts and compensation caps must be large enough to matter and small enough to admit new delegates. The registry, the hash-verified reveal record and the delegation accounts all need design review and an audit.

Use 3: governance

$LULL is the voting token of chapter 18. Behind a timelock, holders vote on market parameters (including the fee up to its cap), market listings, the burn and treasury split, treasury spending, registry parameters and program upgrades. The market authority moves from the team's key to a multisig and then to governance.

Token flow



Parameters: fixed and governable

Parameter	Fixed or governable	Value or bound
Total supply	fixed	1,000,000,000, with no mint authority
Team, presale, private and vesting allocations	fixed	none
Fee cap	fixed in the program by the upgrade	published with the upgrade
Fee rate	governable, up to the cap	5 bps at activation
Fee payers	fixed by the upgrade	filled orders; LP fills exempt
Burn versus treasury split	governable	burn by default
Buyback cadence and execution	governable	published schedule
Bond, open delay, maximum lag, confidence limit, TWAP window and samples, LP spread cap, minimum sizes	governable within the program's bounds	defaults in appendix D
Market listings	governable, through whoever holds the upgrade authority	one market per mint pair; `init_market` requires the upgrade authority
Registry stakes, slashing, compensation cap, unstaking delay	governable	not set
Governance quorum, thresholds, timelock	set at handover, then governable	timelock longer than the longest lull
Commitment scheme, rounding, window ordering, no authority over user funds	fixed in the program	change only by an audited, timelocked upgrade

Phases

The phases are ordered by dependency. No date is given for any of them.

Phase	What happens	Depends on
1. Launch	$LULL launches on a bonding curve; the team's launch wallet is disclosed; the token has no protocol function yet	the owner's decision
2. Fee activation	an upgrade adds fee accounting, the compiled cap, and instructions to transfer a market's authority and treasury; after an audit the fee is set to 5 bps and buybacks begin with public reporting	the program on mainnet (itself subject to the owner's approval), the upgrade, an audit, volume
3. Staking registry	the registry, the hash-verified reveal record and delegation accounts are deployed after an audit; relayers and crank operators can stake	phase 2's changes, design review, an audit
4. Governance handover	SPL Governance goes live with a timelock; the market authority, treasury and upgrade authority move from the team's multisig to governance, one at a time	a working distribution; the authority-transfer instruction

Any phase can be delayed, changed or abandoned. Phases 2 and 3 depend on the Lull program being deployed and used, which has not happened.

20

Roadmap

How to read this roadmap

Every item comes from `docs/LAUNCH.md` or from the token phases of chapter 19. Items are grouped by dependency, and no dates are given, because none has been decided. Deploying the program, funding keys and deploying to Railway each need the project owner's explicit approval. As of 2026-09-24 the owner had approved only the read-only Railway deployment.

Before real funds

Item	What it involves
Audit	an external review, with emphasis on the rounding in `math.rs`, `claim`, `tally` and the account constraints
Exact P_{open} in production	the Pyth Pro key kept on the worker; without it the crank captures from the push account
Issuer risk disclosure	users understand that the vault inherits the AAPLx issuer's delegate, pause and freeze
Jurisdiction gating	the frontend applies the xStocks issuer's eligibility, which excludes US persons. The app already asks each visitor once whether they are a US person and shows US persons a notice instead of the app.
LP onboarding	at least one market maker per market, with documented quoting obligations
Reveal UX	reminders for the reveal window, and an opt-in relayer, which the program already permits

Mainnet deployment

Item	Amount, per `docs/LAUNCH.md`
Program data rent for the 497 KB binary, at current mainnet rent	about 2.53 SOL, reclaimable if the program is closed
Deploy buffer, returned after upload	about 2.53 SOL
Deploy transactions (about 480 writes, with a priority fee)	about 0.01–0.05 SOL
Market initialization (market and two vaults)	about 0.009 SOL
Each published lull	about 0.0049 SOL of rent, returned to the authority by `close_lull` once the lull settles
Crank fees	about 0.00001–0.0001 SOL per transaction

The deployer key needs about 5.1 SOL. The market is initialized with the deployer key, because `init_market` requires the program's upgrade authority, and only after that does the upgrade authority move to a multisig. The Railway services switch from read-only to live with a keyed RPC endpoint on both, the authority, crank and Pyth Pro keys on the worker, and a shorter indexer interval.

Open engineering items

- **More markets:** SPYx, NVDAx and SPCXx (with `Equity.US.SPCX/USD` for the regular session). Each needs its mint, feed id and push shard in `sdk/src/constants.ts`, and one `init_market` signed by the upgrade authority.
- **Arcium Stage 2**, the encrypted-order design of chapter 13.
- **Indexer:** a websocket subscription instead of polling, and backfill tooling.

Token phases

1. **Launch** of $LULL on a bonding curve, with the team's launch wallet disclosed.
2. **Fee activation** after a program upgrade (fee accounting, the compiled cap, and authority and treasury transfer) and an audit. The fee starts at 5 bps, and buybacks follow with public reporting.
3. **Staking registry** for relayers and crank operators, after its own review and audit.
4. **Governance handover** of the market authority, the treasury and the upgrade authority to SPL Governance with a timelock.

Phases 2 to 4 depend on the program being deployed and used. Each can change or be abandoned.

Dependencies

```
owner approval ──▶ mainnet deploy + market init ──▶ upgrade authority to a multisig
                └──▶ live worker (keys, Pyth Pro, keyed RPC)
audit, LP onboarding, jurisdiction gating ──▶ real funds
program upgrade (fee, cap, authority/treasury transfer) + audit ──▶ fee activation ──▶ buybacks
registry design + audit ──▶ staking registry
authority-transfer instruction + token distribution ──▶ governance handover
Arcium readiness + circuit port + audit ──▶ Stage 2 privacy
```

21

Risks

Scope

This chapter lists the main risks of using Lull and of holding $LULL. It is not exhaustive, and chapters 13, 14 and 18 treat several of these risks in more depth. Anyone considering either should read the whole paper and seek independent advice.

Protocol risks

Smart-contract risk. The program has not been audited. Funds in the vaults are under the program's control. A flaw in the program, in the token programs, in Anchor, or in the Pyth and Wormhole receivers could lose funds or make them impossible to withdraw. Tests reduce this risk but do not remove it, and future upgrades, including the planned fee and staking changes, add new code.

Upgrade and key risk. Until the upgrade authority moves to a multisig and then to governance with a timelock, whoever holds it can replace the program, and new code could move funds. The same key is the only one that can open a market. At deployment a market's authority and treasury would be a single key. It can publish lull times, change parameters in ways that forfeit sealed tickets' bonds, and cancel lulls before pricing. It cannot move user funds within the current code.

Oracle risk. Every cross depends on one Pyth feed. A verified, confident, in-window price that is wrong would be used. A Pyth or Wormhole outage around the open cancels the lull. A push-account capture may not be the first update after the open, and the exactness flag records when it is not.

Issuer risk. The AAPLx issuer can move AAPLx out of any account, including Lull's vault, pause all transfers, and freeze accounts. xStocks are not offered to US persons. An xStock's value depends on its issuer and the arrangements behind it, which Lull neither assesses nor guarantees.

Stablecoin risk. The quote asset is USDC, whose value and transferability depend on its issuer.

Liquidity risk. Imbalances fill only as far as LP quotes allow. Without quotes, or when LPs decline to reveal, the heavy side's remainder is refunded, so an order may fill in part or not at all.

Liveness risk. If nobody captures a price in the window, the lull cancels and refunds, and the user misses the open. If the operator's worker is down, users or third parties must send the steps.

Privacy risk. In Stage 1, revealed orders and the imbalance are public until the cross, for up to about 30.5 minutes by default. Deposits and withdrawals are always public. Relayers and holders of an exported secret see orders early.

User-error and network risk. A lost salt or a late reveal forfeits the bond. An order below a minimum raised after sealing cannot be revealed. Solana's clock can differ from the user's, and congestion or outages around 09:00–09:29 ET can delay reveals and cranks.

Operator risk. The API and web app are run by the team. They never hold user keys or see salts before the reveal, but an outage or a faulty build could keep users from building transactions. The

program and its state stay on chain and can be used through the SDK directly.

Token risks

Not implemented. The fee, the buyback, the staking registry and the governance handover do not exist. They need program changes, audits and, for governance, a working distribution. Any may change or never ship, and $LULL may have no function for a long time, or ever.

No claim and no return. $LULL gives no right to revenue, fees, assets, the treasury, the buyback or any payment. A buyback, if it ever happens, depends on volume through Lull, which is zero today, and it can be paused, changed or ended.

Bonding-curve risk. A curve launch is open to anyone, including automated buyers faster than people. Early buyers pay less than later ones. Prices can move sharply on the curve and after it completes. The curve, its fees and its completion rules belong to the launch platform, which carries its own operational and smart-contract risk. Liquidity after completion can be thin.

Concentration and governance capture. The distribution is whatever the curve produces. A few holders can end up with much of the supply and the votes. Governance can be captured, can approve harmful changes including a harmful upgrade, or can fail to reach quorum.

Market and execution risk. $LULL can lose all its value, be volatile, or trade on few venues or none. Scheduled buybacks can be front-run, suffer slippage, and depend on xStock and $LULL liquidity.

Staking risk. A delegate's stake can be slashed for a missed duty, including through faults in its own infrastructure. Slashing rules may contain errors, and unstaking is delayed.

Regulatory and legal risks

The legal treatment of tokenized equities, of auctions for them, and of utility and governance tokens is uncertain, differs between jurisdictions and can change. Authorities could view $LULL, particularly a revenue-funded buyback, as a security or another regulated instrument. That could force design changes, restrict who may hold or use the token, or end parts of the project. Operating an auction venue for tokenized equities may be regulated activity in some places. Lull and $LULL are not intended for US persons or for persons in jurisdictions where they are restricted. The Lull app asks each visitor once whether they are a US person and shows US persons a notice instead of the app. That answer is a self-declaration kept in the visitor's browser, not identity verification, and the program itself, like any Solana program, can be called without the app. Users are responsible for complying with the laws that apply to them, including tax law.

Forward-looking statements

This paper describes plans: a program deployment, upgrades, a fee, a buyback, a staking registry, governance and new markets. They depend on decisions not yet made, audits not yet done, third parties such as Pyth, the xStocks issuer, Arcium and launch platforms, and regulatory developments. Actual outcomes may differ materially, and the team has no obligation to update this paper.

22

Legal notice

Nature of this document

This paper describes software and a token design for information purposes only. It is not a prospectus, an offering document, a solicitation, or a recommendation. It may be changed or withdrawn at any time without notice. No part of it forms the basis of any contract or commitment.

Not an offer of securities

Nothing in this paper is an offer to sell, or a solicitation of an offer to buy, any security, token, derivative or other financial instrument in any jurisdiction. $LULL is designed as a utility and governance token for the Lull protocol. It is not intended to represent equity, debt, a share of profits or revenue, a right to dividends or distributions, a claim on assets, a right of redemption, or any other financial interest in any person or protocol.

No expectation of profit

Nobody should acquire $LULL with an expectation of profit. The planned protocol fee and buyback are not implemented. If they are ever implemented, their purpose is to connect the protocol's use to its governance token, not to produce returns for holders. The buyback may never start, and it can be paused, reduced, redirected or ended. The team makes no promise about the price, liquidity or value of $LULL, or about any feature that is designed but not built.

Not investment, legal or tax advice

Nothing in this paper is investment, financial, legal, accounting or tax advice. The authors are not licensed advisors. Consult your own advisors before taking any action related to Lull, xStocks or $LULL.

Restricted persons and jurisdictions

Lull and $LULL are not intended for, and must not be used by, US persons, or persons located in or resident of any jurisdiction where their use, purchase or holding is prohibited or restricted, or where it would require a registration, licence or approval that has not been obtained. xStocks are not offered to US persons by their issuer, and the issuer's eligibility rules apply to anyone who holds them. The Lull app asks each visitor once whether they are a US person and shows US persons a notice instead of the app. The answer is the visitor's own declaration. It does not verify eligibility and does not relieve anyone of the restrictions above. It is each person's responsibility to know and follow the laws that apply to them.

Regulatory uncertainty

The regulation of tokenized equities, of trading venues for them and of crypto tokens is unsettled and changing. Changes in law or in its interpretation could require changes to Lull or $LULL, restrict their availability, or end them. The buyback in particular may be treated by some regulators as a feature of a security, and it may be changed or abandoned for that reason.

No warranty

The software is provided as is, without warranty of any kind, express or implied, including warranties of merchantability, fitness for a particular purpose and non-infringement. The program has not been audited. To the fullest extent permitted by law, the authors and contributors accept no liability for any loss arising from the use of the software, the token, or this paper.

Third parties

Lull relies on software and services it does not control, including Solana, the Token-2022 and SPL Token programs, Pyth and its receiver, Wormhole, the xStocks issuer, USDC, Jupiter's price API (used for analytics only), Railway, and, in its design, Arcium and SPL Governance. This paper does not claim any affiliation with, or endorsement by, any of them, or by Apple Inc. Names and marks belong to their owners and are used only to identify the systems Lull interacts with.

Forward-looking statements

Statements about plans, designs, phases, upgrades, parameters and future markets are forward-looking. They involve risks and uncertainties, listed in chapter 21, that could cause actual outcomes to differ materially. They speak only as of the date of this paper.

Figures

Figures in this paper come from the project's repository, from its recorded test runs, from its read-only API as fetched on the date stated, or from worked examples that are labelled as such. Past or test figures do not indicate future results.

Appendix A

Instruction reference

Conventions

The program has 15 instructions, listed here by area. "Signer" names the account that must sign beyond the fee payer, and "anyone" means no particular signer. Accounts follow the Anchor structs in `programs/lull/src/instructions`. PDAs are derived under the program id `8xf4NfkWse8YRYghY1femurYsZaBaSEoNgssweT6oGWd` (appendix B).

Administration

Instruction	Signer	Checks (errors)
`init_market(price_feed_id, params)`	the program's upgrade authority, who pays and becomes the market authority	the signer equals the upgrade authority rea from the program data account (`NotUpgradeAuthority`); params within bounds (`BadParams`); each mint matches its token program
`update_params(params)`	the market authority	params within bounds (`BadParams`)
`create_lull(open_ts, close_ts, reveal_start, reveal_end)`	the market authority, who pays	`close < reveal_start < reveal_end ≤ op` and `now < reveal_start` (`BadWindows`)
`cancel_lull`	anyone; the authority at any time	status `Pending` or `Sampling` (`BadLullStatus`); non-authority only after t pricing deadline (`DeadlineNotReached`)
`close_lull`	anyone	status `Finalized` or `Cancelled` (`BadLullStatus`); `claimed + forfeits == commits` (`TicketsOutstanding`); the rent recipient is the market authority

The parameter bounds, from `validate`, are `max_lp_spread_bps < 10,000`, `max_conf_bps ≤ 10,000`, `twap_min_samples ≥ 1`, `max_price_lag_secs ≥ 1`, `open_delay_secs ≤ 3,600`, `min_buy_quote > 0` and `min_sell_base > 0`.

Balances

Instruction	Signer	Checks (errors)	Effect	Event
`deposit(amount)`	the owner	`amount > 0` (`ZeroAmount`); the mint is the market's base or quote mint (`WrongMint`) and the vault is the matching market vault (`WrongVault`)	creates the balance account if needed; `transfer_checked` owner → vault; credits `free` with what the vault received	`Deposited`
`withdraw(amount)`	the owner	as for deposit; `free` ≥ `amount` (`InsufficientBalance`)	debits `free`; `transfer_checked` vault → owner, signed by the market PDA	`Withdrawn`

Tickets

Instruction	Signer	Checks (errors)	Effect	E
`commit(id, commitment)`	the owner, who pays rent and bond	status `Pending` (`BadLullStatus`); `now < reveal_start` (`CommitClosed`)	creates the ticket; stores the commitment, bond and time; moves `bond_lamports` into the ticket; increments `commits`	(
`reveal_order(side, amount, limit, salt)`	anyone	`Pending` ; in the window (`NotInRevealWindow`); ticket `Committed` (`BadTicketState`); valid side (`BadSide`); hash matches (`CommitmentMismatch`); minimum met (`OrderTooSmall`); free balance covers it (`InsufficientBalance`)	locks the amount; stores the fields; ticket `Revealed` ; increments `reveals`	(
`reveal_quote(spread, max_base, max_quote, salt)`	anyone	the same window, state and hash checks; `spread ≤ max_lp_spread_bps` (`SpreadTooWide`); non-empty (`EmptyQuote`); fewer than 8 quotes (`QuoteSlotsFull`); free balances cover both maximums	locks both maximums; writes the next quote slot; updates `spread_hi_ask` / `spread_hi_bid` ; ticket `Revealed`	(
`forfeit`	anyone	`now ≥ reveal_end` (`RevealNotEnded`); ticket `Committed` (`BadTicketState`); treasury is the market's	bond → treasury; ticket closed, rent → owner; increments `forfeits`	F

The cross

Instruction	Signer	Checks (errors)	Effect	Event
`capture_price`	anyone	status `Pending` or `Sampling`; receiver-owned `PriceUpdateV2` (`BadPriceAccount`), fully verified (`PriceNotFullyVerified`), the market's feed (`WrongFeed`), the market's base mint; the capture rule (`TooEarly`, `NonPositivePrice`, `PriceBeforeOpen`, `PriceTooLate`, `StaleSample`)	records a TWAP sample (status `Sampling`), or stores price, confidence, exponent, publish time, source, exactness, multiplier and `px` (status `Priced`)	`PriceSampled` or `PriceCaptured`
`tally` (tickets as remaining accounts)	anyone	status `Priced`; each ticket belongs to the lull (`WrongLull`)	classifies each revealed, untallied order, adds it to the totals, marks it `Tallied`; skips everything else	none
`finalize`	anyone	status `Priced`; `tallied == reveals` (`TallyIncomplete`)	runs `cross`; stores the heavy side, class totals, LP totals and fills; status `Finalized`	`Crossed`
`claim`	anyone	status `Finalized` or `Cancelled` (`BadLullStatus`); ticket not `Committed` (`BadTicketState`); a classed order in a finalized lull must be `Tallied`	releases the lock; applies the order's debit and credit or the quote's fill; returns the rest to `free`; closes the ticket, bond and rent → owner	`Claimed`, with base and quote received and paid

Error codes

The program defines 32 error codes, numbered 6000 to 6031 in this order: `MathOverflow`, `BadWindows`, `CommitClosed`, `NotInRevealWindow`, `CommitmentMismatch`, `BadTicketState`, `WrongLull`, `BadLullStatus`, `InsufficientBalance`, `OrderTooSmall`, `BadSide`, `SpreadTooWide`, `EmptyQuote`, `QuoteSlotsFull`, `BadPriceAccount`, `PriceNotFullyVerified`, `WrongFeed`, `TooEarly`, `PriceBeforeOpen`, `PriceTooLate`, `StaleSample`, `NonPositivePrice`, `TallyIncomplete`, `DeadlineNotReached`, `RevealNotEnded`, `WrongMint`, `WrongVault`, `BadMint`, `ZeroAmount`, `BadParams`, `TicketsOutstanding` and `NotUpgradeAuthority`. `docs/FRONTEND-INTEGRATION.md` and the user documentation map the user-facing ones to plain-language messages.

Appendix B

Account layouts and sizes

Conventions

Sizes are in bytes and include Anchor's 8-byte discriminator. Integers are little-endian. `Pubkey` is 32 bytes. The totals match the sizes stated in `ARCHITECTURE.md` and follow from `InitSpace` in `programs/lull/src/state.rs`.

Program-derived addresses

Account	Seeds
`Market`	`"market"`, base mint, quote mint
`Lull`	`"lull"`, market, `open_ts` (i64 LE)
`Ticket`	`"ticket"`, lull, owner, `id` (u32 LE)
`Balance`	`"balance"`, market, owner
Base vault	associated token account of the market PDA for the base mint, under the base token program (Token-2022 for AAPLx)
Quote vault	associated token account of the market PDA for the quote mint, under the quote token program (SPL Token for USDC)

`Market` (348 bytes)

Field	Type	Bytes
discriminator		8
`authority`	Pubkey	32
`treasury`	Pubkey	32
`base_mint`, `quote_mint`	Pubkey × 2	64
`base_vault`, `quote_vault`	Pubkey × 2	64
`base_token_program`, `quote_token_program`	Pubkey × 2	64
`price_feed_id`	[u8; 32]	32
`base_decimals`, `quote_decimals`, `bump`	u8 × 3	3
`params`	`MarketParams`	41
`lull_count`	u64	8

`MarketParams` (41 bytes)

Field	Type	Bytes	Meaning
`open_delay_secs`	u32	4	P_{open} is published at or after open + delay
`max_price_lag_secs`	u32	4	the latest publish time, relative to open + delay
`max_conf_bps`	u16	2	confidence limit for a direct capture
`twap_window_secs`	u32	4	TWAP window after the first sample
`twap_min_samples`	u8	1	minimum TWAP samples
`max_lp_spread_bps`	u16	2	widest spread a quote may reveal
`bond_lamports`	u64	8	bond per ticket
`min_buy_quote`	u64	8	smallest buy, raw quote
`min_sell_base`	u64	8	smallest sell, raw base

`Lull` (835 bytes)

Field	Type	Bytes
discriminator		8
`market`	Pubkey	32
`open_ts`, `close_ts`, `reveal_start`, `reveal_end`	i64 × 4	32
`status`	enum (u8)	1
`bump`	u8	1
`commits`, `reveals`, `tallied`, `claimed`, `forfeits`	u32 × 5	20
`quote_count`	u8	1
`spread_hi_ask`, `spread_hi_bid`	u16 × 2	4
`price`	i64	8
`conf`	u64	8
`expo`	i32	4
`publish_time`	i64	8
`price_source` (0 none, 1 direct, 2 TWAP)	u8	1
`price_exact`	bool	1
`twap_sum`	i128	16
`twap_samples`	u8	1
`twap_first_ts`, `twap_last_ts`	i64 × 2	16
`multiplier` (× 10^{12})	u128	16
`px` (raw quote per raw base × 10^{12})	u128	16
`tally`	`TallyState` (u64 × 6)	48
`heavy` (0 none, 1 buy, 2 sell)	u8	1
`class_a`, `class_b`	`ClassState` (u64 × 3) × 2	48
`lp_base`, `lp_quote`	u64 × 2	16
`quotes`	`QuoteSlot` × 8	528

`QuoteSlot` (66 bytes)

Field	Type	Bytes
`owner`	Pubkey	32
`spread_bps`	u16	2
`max_base`, `max_quote`	u64 × 2	16
`fill_base`, `fill_quote`	u64 × 2	16

`reveals` counts revealed orders only. Quotes are counted by `quote_count`. `finalize` compares `tallied` with `reveals`.

`Ticket` (154 bytes)

Field	Type	Bytes
discriminator		8
`lull`, `owner`	Pubkey × 2	64
`id`	u32	4
`bump`	u8	1
`commitment`	[u8; 32]	32
`bond`	u64	8
`committed_at`	i64	8
`state` (0 committed, 1 revealed, 2 tallied)	u8	1
`kind` (0 order, 1 quote)	u8	1
`side` (0 buy, 1 sell)	u8	1
`class` (0 none, 1 A, 2 B)	u8	1
`amount` (raw quote for buys, raw base for sells)	u64	8
`limit_price` (raw quote per share; 0 = market on open)	u64	8
`slot` (quote slot)	u8	1
`revealed_at`	i64	8

The `lull` field sits right after the discriminator, so an RPC `memcmp` filter at offset 8 lists every ticket of a lull. The SDK's `fetchTickets` uses it.

`Balance` (105 bytes)

Field	Type	Bytes
discriminator		8
`market`, `owner`	Pubkey × 2	64
`bump`	u8	1
`base_free`, `base_locked`	u64 × 2	16
`quote_free`, `quote_locked`	u64 × 2	16

External accounts read by the program

Pyth `PriceUpdateV2`, as parsed in `oracle.rs`

Offset	Field	Type
0	discriminator `22 f1 23 63 9d 7e f4 cd`	8 bytes
8	write authority	Pubkey
40	verification level: 0 = Partial (followed by a u8 signature count), 1 = Full	u8
41 (Full)	feed id	[u8; 32]
73	price	i64
81	conf	u64
89	expo	i32
93	publish_time	i64
101	prev_publish_time	i64
109	ema_price, ema_conf, posted_slot	i64, u64, u64

The program rejects a Partial update before reading further, so the offsets from 41 onward are those of a fully verified update.

Token-2022 Scaled UI Amount extension

The mint's base layout is padded to 165 bytes, followed by an account-type byte (1 for a mint), followed by a list of extensions, each `type u16 | length u16 | value`. The Scaled UI Amount extension is type 25:

Offset in value	Field	Type
0	authority	Pubkey
32	multiplier	f64
40	new multiplier effective timestamp	i64
48	new multiplier	f64

The program's own program data account

`init_market` reads the program's upgrade authority from its program data account, the PDA of `[program id]` under the upgradeable BPF loader. The program parses the loader's `ProgramData` state by hand:

Offset	Field	Type
0	state tag, which must be 3 (`ProgramData`)	u32
4	slot of the last deployment	u64
12	1 if an upgrade authority is set, 0 if the program is immutable	u8
13	upgrade authority	Pubkey

If no upgrade authority is set, or the signer differs from it, `init_market` fails with `NotUpgradeAuthority`.

Rent

`docs/LAUNCH.md` gives about 0.0049 SOL of mainnet rent for a lull account. That implies about 5,080 lamports per byte, counting Solana's 128-byte account overhead, and the same rate gives the other figures below (derived):

Account	Bytes	Rent	Returned
`Market`	348	about 0.0024 SOL	no
`Lull`	835	about 0.0049 SOL	to the market authority by `close_lull`
`Ticket`	154	about 0.0014 SOL	to the owner at claim or forfeit
`Balance`	105	about 0.0012 SOL	no

On a local validator with the default rent, a lull account holds 6,702,480 lamports, the amount `check:e2e` sees returned by `close_lull`.

Appendix C

Derivations

Notation

`SCALE = 10¹²` is the scale of `px` and of the multiplier `M`, and `BPS = 10,000`. `⌈·⌉` and `⌊·⌋` are integer ceiling and floor division on non-negative operands with 128-bit intermediates. For a heavy-side class `c`, `D_c`, `C_c` and `W_c` are its debit, credit and weight, and `a_i` is order `i`'s locked amount.

C.1 The conversion rate `px`

A Pyth price is `price × 10^expo` USD per share. With `m` the multiplier as a real number (shares per unscaled token), `M = m × SCALE`, `q` quote decimals and `b` base decimals:

```
raw quote per share       = price × 10^(expo + q)
shares per raw base       = m × 10^(-b)
px = SCALE × raw quote per raw base
   = price × (m × SCALE) × 10^(expo + q − b) = price × M × 10^(expo + q − b)
```

In the end-to-end cross, `px = ⌊33,575,101 × 1,003,269,012,540 / 10^7⌋ = 3,368,485,842,620`.

C.2 Light-side conversions

A light-side sell of `b` receives `⌊b × px / SCALE⌋`, and `sell_quote_at_px` is the sum of exactly these floors. A light-side buy of `Q` receives `⌊Q × SCALE / px⌋`, and `buy_base_at_px` is the sum of exactly these. The light side's settlements therefore equal the tallied totals to the raw unit.

C.3 Exact limit comparison

A limit `L` is raw quote per share, and P_{open} per share is `price × 10^e` with `e = expo + q`. To compare `L` with `P_open × f / BPS` without division:

```
compare  L × BPS          with  price × f × 10^e        if e ≥ 0
compare  L × BPS × 10^-e  with  price × f               if e < 0
```

For AAPL, `expo = −5` and `q = 6`, so `e = 1`. For a buy, `L < P_open` gives None and `L < P_open × (1 + s_hi_ask)` gives A. For a sell, `L > P_open` gives None and `L > P_open × (1 − s_hi_bid)` gives A. Otherwise the order is B, and a zero limit is always B. Because `s ≥ 0`, a class can only fall as a buy's limit falls or a sell's limit rises.

C.4 Conservation, buy-heavy

Quote. A class-`c` buyer pays `⌈a_i × D_c / W_c⌉ ≥ a_i × D_c / W_c`, so with `Σ a_i = W_c` the class pays at least `D_c`. The quote paid out is `qm` to sellers (the sum of their floors) plus `lp_quote` to LPs, which equals `pay_A + pay_B_matched + lp_quote = D_A + D_B`. No quote is created, and each order leaves less than one raw unit behind.

Base. Sellers deliver `s` and LPs `lp_base`. Buyers receive `⌊a_i × C_c / W_c⌋`, at most `C_A + C_B = base_A + (s − base_A + lp_base) = s + lp_base` in total. No base is created, and again each order leaves less than one raw unit.

No order pays more than it locked. `pay_A = ⌈qm × q_A / q_all⌉ ≤ q_A` because `qm ≤ q_all` when buys are heavy. For class B, `left = q_B − pay_B_matched ≥ 0` and the LP leg spends at most `left`, so `D_B ≤ W_B`. In both cases `⌈a_i × D_c / W_c⌉ ≤ a_i`, and the `min(debit, amount)` in `settle_order` is a redundant guard.

C.5 Conservation, sell-heavy

The argument is symmetric. Buyers pay in full and receive `bm` in total. Class A gives `⌈bm × s_A / s_all⌉` and receives `⌊q × s_A / s_all⌋`. Class B gives `give_B_matched + lp_base` and receives `q − recv_A + lp_quote`. Sellers give at least what buyers and LPs receive in base, and receive at most what buyers and LPs pay in quote.

C.6 LPs get at least their quoted price

Buy-heavy: the ask `px_k = ⌈px × (BPS + s_k) / BPS⌉ ≥ px × (1 + s_k)`, and the LP receives `⌈b × px_k / SCALE⌉` for `b` base. A partial fill uses `b = ⌊left × SCALE / px_k⌋`, whose cost never exceeds `left`. Sell-heavy: the bid `px_k = ⌊px × (BPS − s_k) / BPS⌋ ≤ px × (1 − s_k)`, and the LP pays `⌊b × px_k / SCALE⌋`, with `b ≤ ⌊max_quote × SCALE / px_k⌋`, so it never pays more than `max_quote`.

C.7 Limits hold for class B

In a buy-heavy cross, class B buys the sellers' remaining base at `px` and LP base at `px_k ≤ px × (1 + s_hi_ask)` plus rounding, so its average price is at most `px × (1 + s_hi_ask)`. Every class-B buy has `L ≥ P_open × (1 + s_hi_ask)`. Each order's share preserves the class ratio up to one raw unit on each side, and the class totals add one more, so the property test allows two raw units of each token. Class A pays the light side's price, `px`, up to the same rounding, and its limit is at least P_{open}. The sell-heavy case is symmetric.

C.8 The TWAP

```
accept sample k    iff  t_k > t_(k-1)  and  t_k ≤ t_1 + twap_window + max_lag
finish at sample n iff  t_n ≥ t_1 + twap_window  and  n ≥ twap_min_samples
P_open             =    ⌊(p_1 + … + p_n) / n⌋
```

C.9 The pricing deadline

The last direct capture may be published at `start + lag`. If it starts a TWAP, the window must pass and the last sample may be up to `lag` later:

```
deadline = open + delay + lag + twap_window + lag = open + delay + 2 × lag + twap_window
         = open + 150 s                                    (defaults)
```

C.10 The exposure bound

The first reveal lands no earlier than `reveal_start`. A direct capture is sent no earlier than `open + delay`, and follows the first eligible update by the capture latency:

```
exposure ≤ (open + delay + latency) − reveal_start = 1,800 + 30 + latency seconds     (defaults)
```

A TWAP extends this by up to the window and the lag.

C.11 Average price per share

For a buy that paid `Q` raw quote for `B` raw base, with multiplier `M`: shares `= B × M / 10^(b + 12)`, and the average is `(Q / 10^q) / shares`. For alice in the end-to-end cross, `Q = 1,999,999,998` and `B = 592,934,847`, giving 5.94873158 shares at about $336.2061.

Appendix D

Default parameters

Market parameters

The SDK's `DEFAULT_PARAMS` (`sdk/src/client.ts`) are the recommended mainnet parameters in `docs/LAUNCH.md`, and `scripts/mainnet-init.ts` initializes AAPLx/USDC with them.

Parameter	Default	Unit	Program bound	Effect
`openDelaySecs`	30	s	≤ 3,600	P_{open} is published at or after open + delay
`maxPriceLagSecs`	30	s	≥ 1	latest publish time of a direct P_{open} after open + delay; bounds the cranker's discretion
`maxConfBps`	25	bps	≤ 10,000	confidence limit for a direct capture
`twapWindowSecs`	60	s	none	TWAP window after the first sample
`twapMinSamples`	3	samples	≥ 1	minimum TWAP samples
`maxLpSpreadBps`	200	bps	< 10,000	widest LP spread; worst premium or discount for a market-on-open order
`bondLamports`	10,000,000	lamports (0.01 SOL)	none	bond per ticket, fixed at commit
`minBuyQuote`	1,000,000	raw quote (1 USDC)	> 0	smallest buy
`minSellBase`	100,000	raw base (0.001 unscaled AAPLx)	> 0	smallest sell

With the defaults, the capture window is `[open + 30 s, open + 60 s]`, the pricing deadline is open + 150 s, a market-on-open buy pays at most $P_{open} \times 1.02$ and a sell receives at least $P_{open} \times 0.98$, and the imbalance exposure is bounded by about 30.5 minutes.

`check:e2e` compresses a lull into about a minute, so it overrides the timing parameters (`openDelaySecs` 2, `maxPriceLagSecs` 180, `twapWindowSecs` 30, `twapMinSamples` 2) and keeps the rest. `check:exact` uses `DEFAULT_PARAMS` unchanged.

Worker settings

Variable	Default	Meaning
`REVEAL_LEAD_SECS` / `REVEAL_CLOSE_SECS`	1,800 / 60	reveal window start and end, before the open
`CALENDAR_DAYS`	14	days of lulls computed ahead
`PUBLISH_AHEAD_SECS`	3,600	how long before a lull's close it is published
`LULL_MARKETS`	`AAPLx/USDC`	markets served
`BENCHMARKS_URL`	`https://benchmarks.pyth.network`	historical Pyth prices for the drift backfill
`PYTH_PRO_API_KEY`	unset	enables the exact P_{open} and the backfill; never logged
`AUTHORITY_KEYPAIR` / `CRANK_KEYPAIR`	unset	enable publishing lulls / permissionless cranks

The crank waits 15 seconds after open + delay for the exact update before using the push account (`EXACT_GRACE_SECS`), pays the Pyth receiver's update fee to treasury 0, and batches 5 forfeits, 8 tallied tickets or 4 claims per transaction. It finds the first Hermes update in the capture window with one request for the window's end and a binary search, pausing 150 ms between search steps, and after a rate-limit answer it waits 1.5 s and asks again, up to five times per search. Once a lull's tickets are all gone, a later pass saves its LP quotes to the database and calls `close_lull`.

Market constants

Constant	Value
AAPLx mint	`XsbEhLAtcf6HdfpFZ5xEMdqW8nfAvcsP5bdudRLJzJp` (Token-2022, 8 decimals)
USDC mint	`EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v` (SPL Token, 6 decimals)
Price feed	`Equity.US.AAPL/USD`, `49f6b65cb1de6b10eaf75e7c03ca029c306d0357e91b5311b175084a5ad55688`
Push account (shard 1)	`D9uk39pqZMcnmtPP9WeC8cREUpKZmyXLga9mSQ79SphW`
Pyth receiver	`rec5EKMGg6MxZYaMdyBfgwp4d5rB9T1VQH5pJv5LtFJ`
Lull program id (not deployed)	`8xf4NfkWse8YRYghY1femurYsZaBaSEoNgssweT6oGWd`
AAPLx/USDC market address (derived)	`AkFq1LLZJCQ7yUJd23kcurQos79FSk5k8PRwJgnsakDp`

Planned parameters (not implemented)

Parameter	Design value
Protocol fee rate	5 bps of each filled order's crossed notional
Protocol fee cap	compiled into the program by the upgrade; value to be published
Buyback proceeds	burned by default; governance may redirect a share to the treasury
Registry stakes, slashing, compensation cap, unstaking delay	not set
Governance timelock	longer than the longest lull, 92.5 h in the current AAPL schedule

Appendix E

Glossary

Balance account. A user's per-market record of free and locked raw amounts of both tokens. Free balance can be withdrawn or locked. Locked balance is held by a revealed order or quote until claim.

Bond. SOL posted with each ticket, 0.01 SOL by default. It is returned at claim and forfeited if the ticket is never revealed.

Bounded discretion. The choice any capturer has among Pyth updates published inside the capture window.

Buyback. The planned use of protocol fee revenue to buy $LULL on the open market, burned by default. Not implemented.

Claim. Settling one ticket into its owner's balance and closing it.

Close (`close_lull`). Closing a finalized or cancelled lull once every ticket is claimed or forfeited. The rent returns to the market authority.

Class. An order's eligibility at P_{open}. None is refunded, A takes part in the matched volume only, and B also takes part in the LP leg.

Commit, commitment. Posting a SHA-256 hash over a domain tag, the ticket's fields, a 32-byte salt, the owner and the lull, together with a bond. No tokens move.

Confidence interval. Pyth's uncertainty band around a price. A direct P_{open} requires it to be at most 25 bps of the price by default.

Crank. The permissionless steps that move a lull forward: forfeit, capture, tally, finalize, claim, cancel after the deadline, and close the finished lull.

Exact path, exactness flag. The worker finds the first Hermes update at or after `start` (Hermes returns the last update at or before a time, so the worker searches the window), posts it through the Pyth receiver and captures it in the same transaction. `price_exact` is set when the captured update's previous publish time is before `start`, which proves it is the first update at or after `start`.

Exposure. The time from the first reveal to the price capture, during which the imbalance is public.

Forfeit. Closing an unrevealed ticket after the reveal window. The bond goes to the treasury, and the rent to the owner.

Free option. A committed participant's ability to decline to reveal after watching other reveals.

Heavy side, light side. The side with more eligible value at P_{open} is heavy. The light side fills in full at P_{open}.

Lull. The interval from one session's close to the next session's open: overnight, weekend or holiday.

LP quote. A sealed offer to absorb the imbalance, made of a spread, a maximum base to sell and a maximum quote to spend. It fills pay-as-quoted, at P_{open} plus or minus its own spread.

Market authority. The key that publishes lulls, sets parameters and can cancel a lull before pricing. It cannot move user funds.

MOO, LOO. Market-on-open (no limit) and limit-on-open (a limit in USD per share) orders.

P_{open}. The price a lull crosses at: the first fully verified Pyth price of the underlying published in `[start, start + lag]` whose confidence is within the limit, or else a TWAP of later verified samples.

Phase, status. The API's view of a lull (scheduled, unpublished, commit, reveal, sealed, pricing, crossing, settled, cancelled) and the program's (`Pending` , `Sampling` , `Priced` , `Finalized` , `Cancelled`).

Pricing deadline. `open + delay + 2 × lag + TWAP window` , 150 s after the open by default. After it, anyone can cancel an unpriced lull.

Push account, posted update. A Pyth-sponsored `PriceUpdateV2` account kept up to date at a fixed address, or a specific update posted through the receiver into a new account.

`px` . Raw quote per raw base × 10^{12}, with the Scaled UI multiplier included.

Raw base, raw quote, quote per share. The xStock's Token-2022 amount before the multiplier (8 decimals for AAPLx), USDC base units (6 decimals), and raw USDC per one share, the unit of limits.

Registry. The planned list of staked delegates for reveal relaying and cranking. Not implemented.

Relayed reveal. A reveal sent by someone other than the owner who holds the salt.

Reveal, reveal window. Publishing a ticket's fields and salt, which the program checks and locks, inside `[reveal_start, reveal_end)` , by default from 30 minutes to 1 minute before the open.

Reveal template. The unsigned reveal transaction the API builds with every secret field zeroed, for the client to fill in.

RR feed. Pyth's redemption-rate feed for an xStock. It describes the same ratio as the mint's multiplier, and Lull monitors it but never multiplies by it.

Salt. 32 random bytes generated on the client. Knowing it authorizes the reveal.

Scaled UI multiplier. The mint's ratio of shares to unscaled tokens, stored by Lull × 10^{12}.

`s_hi` . The widest revealed LP spread on the side that would absorb an order.

Session. One regular trading period of the underlying: 09:30 to 16:00 ET on weekdays for AAPL, with holidays and early closes.

Start. The open plus the open delay: the earliest publish time a P_{open} can have.

Ticket. The account for one sealed order or LP quote.

Timelock. The planned delay between a passed governance vote and its execution.

Treasury. The account that receives forfeited bonds.

TWAP. The fallback P_{open}: the integer mean of verified samples over a short window.

Upgrade authority. The key that can replace the program's code. It is also the only key that can open a market.

VAA. A message signed by the Wormhole guardians. For Pyth it signs the Merkle root of a batch of price updates.

Appendix F

References

Repository documents

Document	Contents
`README.md`	overview, gate results, verified inputs, the sample cross
`ARCHITECTURE.md`	the implemented design: life of a lull, program, pricing, units, the cross, design decisions, trust assumptions, off-chain components, gates, known limitations
`docs/PRIVACY.md`	Stage 1 exposure and leaks; the Stage 2 Arcium design
`docs/LAUNCH.md`	deployment steps and costs, the Railway setup, prerequisites for real funds, open engineering items
`docs/FRONTEND-INTEGRATION.md`	the API and SDK contract, units, phases, flows, errors
`docs/e2e-sample-report.json`	the recorded end-to-end cross
`web/content/docs/`	the user documentation
`web/DESIGN.md`	the web design notes, including the lull clock

Source files

File	Contents
`programs/lull/src/math.rs`	the crossing math and its randomized test
`programs/lull/src/pricing.rs`	the P_{open} state machine and its tests
`programs/lull/src/oracle.rs`	the `PriceUpdateV2` and Scaled UI Amount readers
`programs/lull/src/state.rs`	account layouts and market parameters
`programs/lull/src/instructions/`	`admin.rs`, `funds.rs`, `tickets.rs`, `crossing.rs`
`programs/lull/src/errors.rs`, `events.rs`	error codes and events
`sdk/src/matching.ts`	the TypeScript mirror of the crossing math
`sdk/src/units.ts`	unit conversions
`sdk/src/commitment.ts`	commitments and salts
`sdk/src/client.ts`	instruction builders and `DEFAULT_PARAMS`
`sdk/src/receiver.ts`	posting a Hermes update through the Pyth receiver
`sdk/src/schedule.ts`	the schedule parser
`server/src/worker/cranks.ts`	publishing lulls, the crank, and the exact capture
`server/src/worker/drift.ts`, `samplers.ts`, `pythHistory.ts`	drift analytics, samplers, Hermes and Benchmarks by timestamp, and `firstUpdateFrom`
`tests/cross.test.ts`, `tests/schedule.test.ts`	the property and calendar gates
`scripts/e2e.ts`, `scripts/check-api.ts`, `scripts/check-exact.ts`	the end-to-end, API and pricing (exact and TWAP) gates
`scripts/dev-stack.ts`, `scripts/dev-fund.ts`	the persistent local stack for testing the web app, and its faucet

Live endpoints

All were fetched on 2026-09-26, when the server clock read 2026-09-25 23:46 UTC.

- `https://lull-api-production.up.railway.app/health`
- `https://lull-api-production.up.railway.app/v1/markets/AAPLx-USDC`
- `https://lull-api-production.up.railway.app/v1/markets/AAPLx-USDC/calendar?days=14`
- `https://lull-api-production.up.railway.app/v1/markets/AAPLx-USDC/drift`

External documentation

- Pyth Network: https://pyth.network
- Pyth documentation, including price feeds, the Solana receiver, Hermes and Benchmarks: https://docs.pyth.network
- Solana: https://solana.com
- Wormhole: https://wormhole.com

Literature

- Eric Budish, Peter Cramton and John Shim, "The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response", *The Quarterly Journal of Economics* 130(4), 2015.