# Galyari: Building a Free, Linux-Native Order-Flow Heatmap from a Brokerage Feed

*Quinn Lambert*

## Abstract

Commercial order-flow tools—Bookmap, ATAS, Sierra Chart—are Windows-centric and gate live equity depth behind a paid market-data subscription, typically a Nasdaq TotalView feed layered on top of the software license. I already had the data. A Charles Schwab brokerage account with API access exposes Level 2 book data (`NASDAQ_BOOK`) over a streaming WebSocket, and that is the same class of input a liquidity heatmap is built from. So rather than pay for the software *and* the feed, I built the visualization from the feed I already had, on Linux, for free. Galyari renders a Bookmap-style view of the limit order book over time: price on the y-axis, time scrolling right-to-left on the x-axis, color intensity as resting size at each price level. The goal is to *see* the microstructure a candlestick chart hides—where liquidity rests, where walls build and get pulled, absorption, and imbalance. This document is the build log and the reasoning behind it: what the feed actually is (and is not), the architecture that keeps the data layer separate from the renderer, the phased build from a 270-line seed to a multi-symbol browser renderer, the visualization decisions that took the most iteration, and the bugs that cost the most time. The throughline is a single design bet—that the parsed order book is the stable core and the renderer is replaceable—and the report is largely the story of that bet paying off.

## 1. Introduction

The first question behind a lot of discretionary trading is the one a price chart can't answer: *where is the liquidity?* A candlestick tells you where price has been. It does not tell you that there are 4,000 shares resting twelve cents below the touch that have been sitting there for ninety seconds, or that the offer that was capping the move just vanished a half-second before the break. That resting intent, and the way it appears and disappears, is what an order-flow heatmap is for. It is a standard tool, and the standard tools are good—Bookmap in particular is the reference. They are also Windows-first and they assume you are paying for a depth feed on top of the application.

The motivation here was narrow and practical. I run Linux, I had a Schwab account with API access turned on, and Schwab's streamer carries `NASDAQ_BOOK`—a multi-venue aggregated Level 2 book—over WebSocket. That is the expensive input, already entitled to my account. The missing piece was the renderer, and a renderer is just software. So the project is the software: ingest the book, bin it into a rolling price×time matrix, and paint it.

There were two audiences for the output from the start, and the architecture had to serve both:

- **Discretionary use**—a live window to watch liquidity while trading or observing.
- **Systematic research**—the *parsed* book as a clean, importable data structure that other research projects can consume to build features (book imbalance, wall persistence, absorption events) without

dragging in a plotting library.

That second audience is the one that shaped the codebase. It meant the parsing and book-state layer could not be tangled into the renderer; it had to be a module you can `import` with no matplotlib in the dependency graph. I made that a hard rule, and most of the architectural decisions below are downstream of it. The bet was that if the data layer was genuinely renderer-agnostic, I could later swap matplotlib for something faster without touching ingestion—and by the end I did exactly that, which is the cleanest evidence the rule was worth keeping.

The rest of this report describes the feed, the architecture, the build phases in order, the visualization work, and the hard bugs—then the limitations, stated plainly, and what is still left to do.

## 2. The feed: what `NASDAQ_BOOK` actually is

Before any rendering, I had to know exactly what the wire gives you, because the whole matrix is built on assumptions about the message shape. schwab-py relabels the raw numeric streamer fields into readable names. Each message carries a `content` list with one entry per subscribed symbol, and each entry is a snapshot of one side's price levels with a nested per-venue breakdown. A real captured GOOG frame, trimmed, which I kept verbatim as the test fixture:

```
{
  "key": "GOOG",
  "BOOK_TIME": 1782144614895,
  "BIDS": [
    { "BID_PRICE": 344.80, "TOTAL_VOLUME": 120, "NUM_BIDS": 1,
      "BIDS": [ { "EXCHANGE": "NSDQ", "BID_VOLUME": 120, "SEQUENCE": 43814837 } ] },
    { "BID_PRICE": 344.67, "TOTAL_VOLUME": 200, "NUM_BIDS": 2,
      "BIDS": [ { "EXCHANGE": "edgx", "BID_VOLUME": 120, "SEQUENCE": 43814848 },
                { "EXCHANGE": "batx", "BID_VOLUME": 80,  "SEQUENCE": 43814848 } ] }
  ],
  "ASKS": [
    { "ASK_PRICE": 344.90, "TOTAL_VOLUME": 100, "NUM_ASKS": 1,
      "ASKS": [ { "EXCHANGE": "arcx", "ASK_VOLUME": 100, "SEQUENCE": 43814900 } ] }
  ]
}
```

A few facts mattered enough to lock down:

- The top level per side (`BIDS` / `ASKS`) is an array of **price levels**, sorted best-to-worse. Each level has its price, a `TOTAL_VOLUME` aggregated across venues, and a **nested same-named array** breaking that level down per venue (`EXCHANGE`, the venue volume, a `SEQUENCE`).
- `BOOK_TIME` is epoch milliseconds.
- The observed venues span `NSDQ, arcx, edgx, batx, nyse, memx, miax`. So despite the name, this is a **multi-venue aggregated book** surfaced through Schwab, not a Nasdaq-only feed. I preserve the per-venue breakdown on the parsed `Level` even though the heatmap only uses the total—it may matter for research (venue-specific persistence, for one).

### 2.1 Snapshot, not delta

The single most important question for the data model: is each message a full book, or an incremental update? If it's a delta feed I'd need add/modify/delete keying and sequence reconciliation; if it's a snapshot I can discard the previous book and rebuild on every message, which is much simpler and impossible to get subtly out of sync.

**The finding is that each message is a full snapshot.** The `BIDS`/`ASKS` arrays represent the complete current visible book at `BOOK_TIME`. There is no per-level update flag and no stream-wide sequence cursor—the only `SEQUENCE` numbers are per-venue identifiers *inside* a level, not a delta cursor across messages. This matches the behavior Schwab inherited from the old TD Ameritrade book services. So full-replace is correct, and `OrderBook.apply()` just rebuilds. I wrote down what would falsify it—levels that stick stale across frames, volumes that only ever grow, single-changed-level messages—so a future live session has a clear test if the assumption ever breaks.

### 2.2 There is no time-and-sales tape

This was the unwelcome discovery, and it shaped the entire trades layer. Schwab's streamer has **no time-of-sale service.** The legacy TD Ameritrade `TIMESALE_EQUITY` stream was dropped; schwab-py exposes only `level_one_equity`, `chart_equity`, and the book services. So there is no tick-by-tick trade tape to subscribe to.

The workaround is to **derive prints from `LEVEL_ONE_EQUITY`.** Level-one is a field-delta feed—a field shows up only when it changes—so the producer caches the last-known `(trade_ms, price, size)` per symbol and emits a synthetic `Trade` when the last-trade timestamp advances. Aggressor side is then inferred against the prevailing book: a print at or above the best ask is buyer-initiated, at or below the best bid is seller-initiated, otherwise unknown.

I want to be honest about what this is. Level-one last-trade is the *consolidated* last sale sampled at the feed's update cadence—it is **not** a true tick-by-tick tape. Rapid same-price prints can coalesce into one, and odd-lot or auction prints may be filtered out by the consolidated feed. For order-flow *visualization* it is entirely adequate; you can see aggression and direction. For trade-level *research* the CVD and bubble layers should be treated as an approximation, not ground truth. The redeeming design point is that the `Trade` struct and everything downstream of it know nothing about where prints came from, so if I ever find a genuine T&S source, I swap one producer function and the rest is unchanged.

## 3. Architecture

### 3.1 One sentence

A **background producer thread** (Schwab stream, simulator, or replay) parses incoming frames into book state and writes the latest snapshot into a **shared, lock-guarded store**; the **renderer** snapshots that store a few times a second, bins it into a rolling price×time matrix, and draws it.

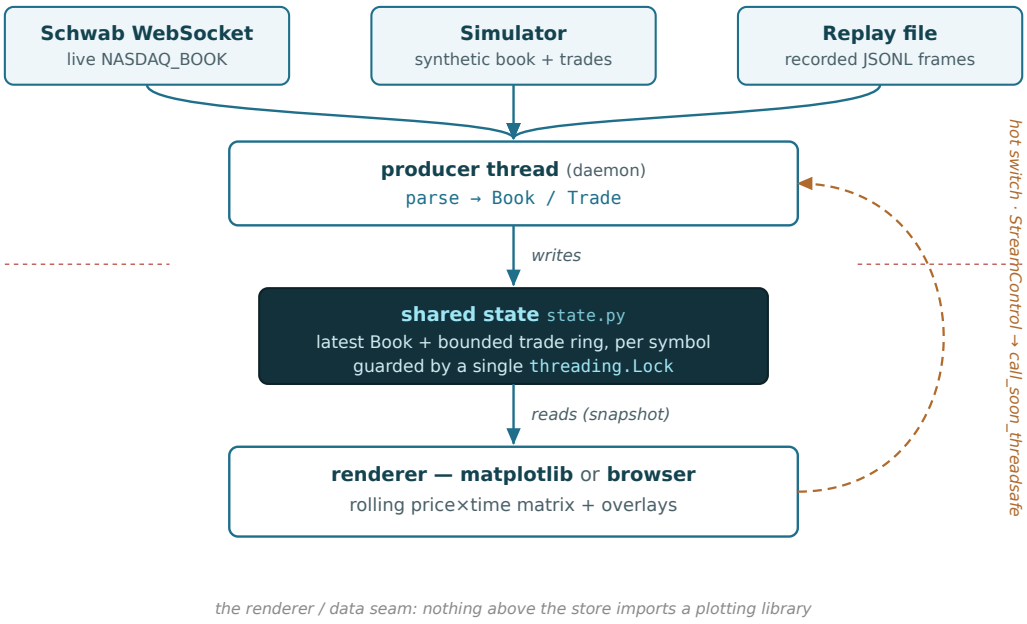


***Figure 1.*** *The producer / seam / renderer model. Three interchangeable producers (live stream, simulator, replay) write a full book snapshot into a single lock-guarded store; the renderer reads snapshots and paints them. The only backward signal is a hot-ticker switch, marshalled onto the stream's own asyncio loop. Nothing above the store imports a plotting library.*

### 3.2 Why threads, and why this exact split

The split is forced by three things that each insist on owning a loop. matplotlib must own the main thread—its event loop and GUI backend require it. Schwab's stream client is asyncio and blocks on `handle_message()`. The simulator sleeps in a loop. None of those can share the GUI thread, so each producer runs on its own daemon thread and talks to the renderer *only* through the shared store. The render loop never blocks on the network; the producer never touches matplotlib. I called this the **renderer/data seam**, and enforcing it is the spine of the whole project: the producer side knows nothing about how, or whether, data is drawn.

The one signal that flows *backward* (renderer → producer) is a hot ticker switch, and even that never reaches into schwab-py directly. The renderer posts a command through a `StreamControl` object, which hops it onto the stream's asyncio loop via `loop.call_soon_threadsafe`. The command is consumed *inside* the message loop so every socket operation—subscribe, unsubscribe, receive—stays on one task. That is not optional politeness: schwab-py guards the socket with a single `asyncio.Lock`, so a subscribe issued from the wrong thread would deadlock or race.

### 3.3 The shared-state contract

`state.py` is deliberately tiny—the latest `Book` per symbol and a bounded ring of recent trades per symbol, behind one `threading.Lock`. Writers (`set_book`, `add_trade`) are routed by `book.symbol`; the reader uses `get_book()` / `drain_trades()`, whose no-arg form returns the *focused* symbol so the single-symbol renderer never had to change when multi-symbol arrived. The rules that keep it correct:

- The store keeps only the **latest** book per symbol (it's a full snapshot, so there's nothing to accumulate) and a small ring of trades. No history lives here—history lives in the renderer's rolling matrix and, optionally, the recording file.
- Everything crossing the boundary is **immutable by convention**: producers build a fresh `Book` and hand it over; the reader never mutates what it reads. This keeps the lock hold-time down to a single assignment or list append.
- An **empty book is a valid state, never an error.** Market closed or no entitlement looks exactly like silence after the `subscribed` log line, and treating that as a failure would be wrong.

### 3.4 The module map

The hard rule—*nothing in the stable core or producers imports matplotlib*—is visible in the dependency graph:

| File | Role | matplotlib? |
|---|---|---|
| `orderbook.py` | **Stable core.** Typed `Book`/`Level`/`Trade`, `parse_nasdaq_book`, `OrderBook` state. | No |
| `state.py` | Thread-safe shared store between producer and renderer. | No |
| `feeds.py` | Producers: live stream, simulator, replay; `StreamControl` for hot switching. | No |
| `recorder.py` | Append-only JSONL recording of raw frames. | No |
| `microstructure.py` | Wall / pull / iceberg detection over the `Book`/`Trade` stream. Stdlib only. | No |
| `schwab_orderflow_heatmap.py` | CLI entrypoint + matplotlib renderer. | Yes |
| `bridge.py + web/` | Alternative frontend: WebSocket publisher + browser canvas. | No |

That the data layer is renderer-agnostic stopped being a hypothesis the moment `bridge.py` existed: it is a second, complete frontend that reuses `feeds`/`state`/`orderbook` unchanged and never imports matplotlib. The bet from Section 1 is the line in this table where two very different renderers share the same six no-matplotlib modules.

## 4. The build, phase by phase

The project started as a single ~270-line file that had been debugged to a working state in a live session—it ingested the book live or simulated, wrote to a shared dict under a lock, and drew an `imshow` heatmap with a mid-price line at ~3 Hz. That seed worked end-to-end, and the rule was to preserve its behavior except where a phase explicitly changed it. I grew it in coherent, reviewable phases.

**Phase 0 — Documentation and hygiene.** Before touching behavior, I wrote the docs from scratch: a README, an architecture doc, and a data-schema doc that pinned down everything in Section 2. Added

`requirements.txt` and a `.gitignore` for `token.json`, `.env`, and `__pycache__`. The point of doing docs first was to force myself to actually understand the feed and the seam before building on them.

**Phase 1 — Extract the reusable core.** Pulled parsing and book state into `orderbook.py`: the frozen dataclasses, `parse_nasdaq_book(msg) -> Book | None`, and an `OrderBook` holder. Added pytest unit tests using the captured GOOG frame as a fixture. This is the module the other research projects import, so it had to be clean and dependency-light. Behavior of the main script was unchanged; it just imported from the new module.

**Phase 2 — Price-window recentering / auto-zoom.** This was the highest-value fix. The seed anchored its price window on the first observed mid and never moved it, so as price drifted the action would clip off the top or bottom of the view. The fix was to store liquidity keyed by **absolute price bin** rather than by row offset, so the y-axis can follow price drift by *shifting rows* instead of losing history. Recentering became a vertical roll of the stored columns; zoom became a lossless crop, because every level is already stored at its true price. The simulator's random walk drifts out of the old window fast, which made it the perfect test.

**Phase 3 — Trades layer.** Given the no-T&S finding (Section 2.2), this is the level-one-derived prints rendered as **bubbles** (area ∝ size, color by inferred aggressor side) plus a **CVD** panel (running Σ of signed trade size). I extended the simulator to emit synthetic trades so the whole layer is testable with no credentials.

**Phase 4 — Record and replay.** An append-only JSONL event log, one frame per line, never rewritten. The key design choice: `--replay` feeds recorded frames back through the **same shared-state plumbing** the simulator uses, so the renderer is byte-for-byte identical across live, sim, and replay. That uniformity is why a bug seen live can be reproduced offline from a tape.

**Phase 5A Tier 1 — Hot ticker switching.** Live retargeting with no restart: `n`/`p` cycle a `--watchlist`, and a `go to ▸` text box jumps to any symbol. This is where the `StreamControl` → `call_soon_threadsafe` machinery from Section 3.2 came in, plus a **stale-frame guard**—the renderer arms a guard on `state` *before* the stream resubscribes, so a late book from the old symbol cannot flash on the new symbol's chart.

**Phase 5C — Wall / iceberg / pull detection.** A pure consumer of the parsed stream in `microstructure.py` (stdlib only, importable by research code without the renderer). A sliding window of per-price state yields three signals:

- **Wall**—a level that is large *now* (≥ a multiple of the book's median size) **and** has been large for several frames. Requiring "big now" means a pulled wall stops being reported immediately instead of lingering.
- **Pull**—a level that was a wall last tick and whose size just collapsed. Discrete, one-shot, written to the event log under `--record`. Often precedes a fast move.
- **Iceberg**—a level whose cumulative *executed* volume far exceeds its displayed size, with repeated refills. Crucially a refill only counts when it follows **execution** at that price within a short window—

tying refills to trading is what rejects pure book-size noise, which was the main false-positive source.

I tuned the whole module for **precision over recall**: better to miss a marginal wall than to clutter the view with maybes.

**Phase 5D — Analytics panels.** All derived in the renderer from the same `Book`/`Trade` stream, no core changes: a **volume profile** with point-of-control in the right margin (where business actually happened, versus where intent rests), a **delta footprint** strip (net buy−sell per time bucket), a **bid/ask touch staircase with a translucent spread band**, a legend, and a conservative text-only **absorption flag** in the header.

**Phase 5B — Browser renderer.** The payoff of the seam. `bridge.py` is a *peer* of the matplotlib renderer, not a layer on top: it launches the same producer thread filling `state.py`, then runs an asyncio WebSocket server instead of a GUI loop, broadcasting one JSON frame per column tick. `web/app.js` is dependency-free vanilla canvas—no build step, no charting library, nothing from a CDN—that keeps its own rolling matrix client-side and paints the heatmap the way GPU heatmaps do (a 1px-per-cell offscreen image scaled up with smoothing off), overlaying everything the matplotlib view has as vectors.

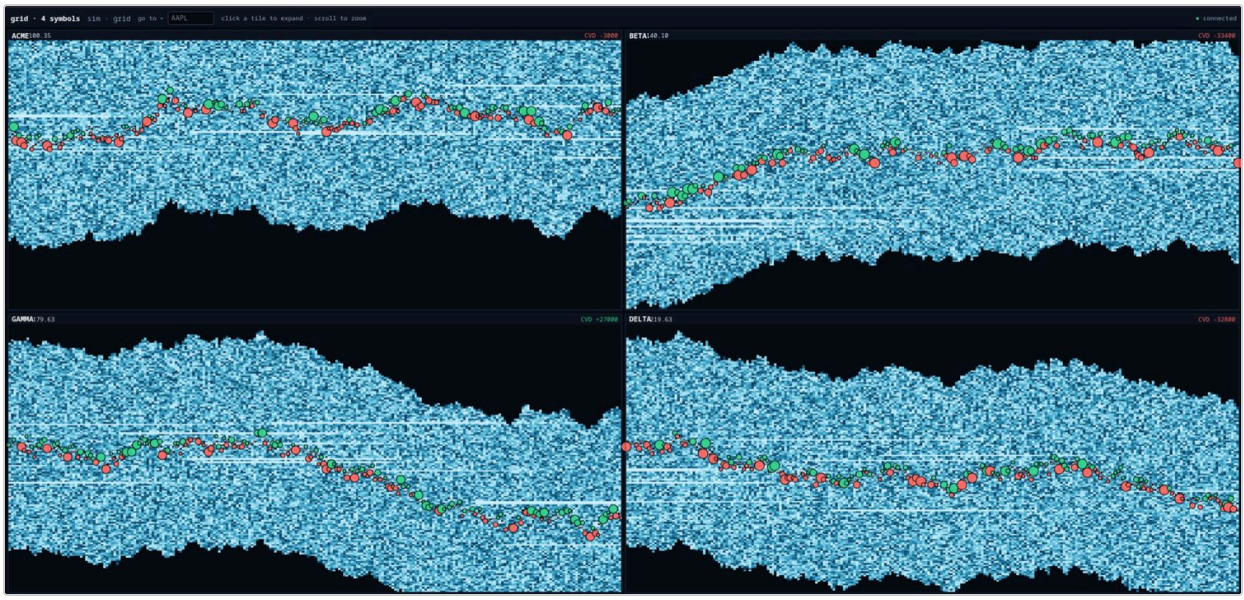

***Figure 2.*** *The browser renderer in multi-symbol grid mode: four live heatmaps (ACME, BETA, GAMMA, DELTA) streaming at once, each with its own mid-line, trade bubbles, and running CVD. Because every symbol streams continuously, clicking a tile to expand to the full single-symbol view is purely client-side. The grid—rather than the originally planned focus-plus-sidebar—was built because the actual goal turned out to be seeing every symbol at once.*

**Phase 5A Tier 2 — Simultaneous multi-symbol.** This generalized `state.py` from a single book to per-symbol dicts plus a subscribed-set guard and a focus, with the single-symbol API preserved verbatim so the matplotlib path and every test were untouched. Each broadcast frame carries *all* watched symbols in full, so the browser renders a **grid of live heatmaps**; click a tile to expand to the full single-symbol view, Esc to return. Because every symbol streams continuously, expanding is purely client-side. The

practical cap is 8 symbols—book data is heavy per symbol and Schwab caps the overall streaming rate, so a handful is the sweet spot, not dozens.

## 5. Visualization decisions

A surprising fraction of the work was not plumbing—it was making the picture *legible* on real equity books, which are sparse and heavy-tailed in a way that defeats naive choices.


***Figure 3.*** *The full single-symbol view, here on simulated flow. The heatmap uses a power-law color norm so small resting sizes stay visible while walls (the bright horizontal streaks) still pop; the mid-line walks through it with buy/sell trade bubbles sized relative to the recent typical print. Right margin: the volume profile with its point-of-control (gold). Lower panels: the delta footprint and cumulative volume delta. The header carries live mid, spread, imbalance, CVD, and wall / ice / pull counts.*

**Color scaling.** A linear color map makes a thin book look empty: one wall saturates and everything else is black. Equity depth is heavy-tailed, so I use a power-law norm (`PowerNorm`, gamma ≈ 0.45) with a per-frame 97th-percentile `vmax`. Small resting sizes stay visible while walls still pop, and the auto-`vmax` keeps the scale sane as liquidity ebbs and floods.

**The visible price band.** The instinct is to fit the y-axis to the full book depth. That is wrong for equities—the book is sparse and wide, so fitting the furthest level buries all the action in a thin strip near the middle. Instead the visible band is a fixed window, roughly 0.15% of price on each side, centered on the touch, with `+/−` to widen or tighten it. Because liquidity is stored at absolute price bins, zooming out is lossless: the history is already there, never rebuilt. Expect ~10–15 streaks near the price on a normal name, not a dense wall.

**Bubble sizing.** Absolute trade size doesn't translate across names—a 500-share print is enormous in one symbol and noise in another. So bubble area scales with size *relative to the recent typical print*, which keeps small-lot names expressive instead of flat.

**Absorption flag.** Rather than spray markers, absorption is a single quiet header flag. It fires only when a one-sided delta burst (≥ 8× the typical print) is met by ≤ 2 ticks of price movement—aggression hitting a wall that holds. Deliberately conservative; a flag you see twice a session and trust beats one that's always on.

## 6. Bugs worth remembering

Two bugs cost real time and are worth writing down because both were non-obvious and both were about matplotlib's drawing model, not the data.

**The invisible heatmap.** At one point the heatmap rendered as a black background while the mid line, bubbles, and overlays drew fine. The cause: the `imshow` artist had been marked `animated=True`. The animation runs with `blit=False` (full redraws), and on matplotlib 3.x an animated artist is *excluded* from a normal full draw—it only paints on the blit path. With blit off, the one animated artist never rendered while every non-animated artist did. The fix was one flag, but finding it meant understanding why only *one* artist was missing.

**Coordinate-system drift on zoom/recenter.** The image extent and the axes y-limits must be re-locked to the current row count *every frame*. If they drift apart, the heatmap, the mid line, and the bubbles end up in mismatched coordinate systems and the overlays float off the liquidity they describe. Re-asserting `n_rows` on both, every tick, is what keeps them welded together through zoom and recenter.

## 7. Verification status

Galyari has been run against the live Schwab socket across full regular-session flow, and every layer has been watched under real data rather than only simulated data. Confirmed by direct observation, against the real socket: token reuse with no re-auth prompt; live `NASDAQ_BOOK` streaming and rendering at the target cadence; hot symbol-switching via the `go to ▸` box (runtime resubscribe plus the stale-frame guard); the level-one-derived trades layer (CVD and bubbles) tracking real aggression and direction; and the wall / iceberg / pull overlays firing on genuine walls, real refills, and real pulls—not only the simulated ones. The one caveat that survives this is a property of the *source*, not of the testing: the trades layer is a consolidated-level-one derivation (Section 2.2), so it remains an approximation by construction even though it has now been observed to read correctly under live flow.

The OAuth detail that bites everyone—the callback URL must match the app registration character-for-character including the trailing slash, or you get "We are unable to complete your request" *after* logging in—is settled, and the refresh-token cadence (access token auto-refreshes; the ~7-day refresh token means re-pasting the URL about once a week) is confirmed. Offline, the full test suite across `orderbook`, `heatmap`, `live_trades`, `microstructure`, `switching`, `multisymbol`, `bridge`, and `replay` passes, and the captured GOOG tapes give a faithful replay path for regression against a known-good frame. In short, the system is battle-tested end to end: live ingestion, the seam, both renderers, and the overlay stack have all run on real market flow.

## 8. Limitations

Stated plainly:

- **The trades layer is an approximation.** It is derived from consolidated level-one, not a true tick tape (Section 2.2). Fine for visualization, a caveat for research.
- **matplotlib is a seed renderer, not a high-FPS one.** It was always the ceiling—no smooth pan/zoom, redraw cost grows with panel count. The browser renderer is the answer, and it exists, but matplotlib stays the default so the project never *depends* on a running browser.
- **The matplotlib path is single-symbol** by design; simultaneous multi-symbol lives only in the browser.
- **The per-venue breakdown is parsed but not visualized.** It is preserved on `Level` and waiting for a use (venue-specific persistence is the obvious one).
- **The snapshot-vs-delta model is confirmed against live frames**, and the falsification tests (stale levels, monotonically growing volumes, single-changed-level messages) stay written down should the feed's behavior ever change.
- **Book data flows only during the regular session and only with a Level 2 entitlement.** Silence is normal, not a bug.

## 9. Future work

The data layer is renderer-agnostic, so everything here is additive—none of it requires touching `orderbook.py`, `state.py`, or `feeds.py`:

- **A Lightweight-Charts frontend** for a free time axis, crosshair, price scale, and panes, with the heatmap as a custom series primitive—if the vanilla-canvas renderer's polish or pan/zoom ever becomes the bottleneck.
- **Binary frame payloads** (msgpack / typed arrays) if JSON bandwidth bites at higher symbol counts or refresh rates. The frame schema stays identical.
- **Runtime add/remove of watched symbols** in the browser grid.
- **A genuine time-and-sales source**, if one surfaces, to replace the level-one derivation and upgrade the trades layer from approximation to ground truth. One producer function changes; nothing downstream does.
- **Visualizing the per-venue breakdown**—the first real consumer would likely be venue-level wall persistence in `microstructure.py`.

## 10. Conclusion

Galyari started from a small, narrow bet: that a free brokerage depth feed plus a renderer I wrote myself could stand in for a Windows-only commercial tool and its paid data feed, on Linux. The bet held, and the part that made it hold was an architectural discipline—keep the parsed order book as the stable core,

keep the renderer replaceable, and never let matplotlib leak into the data layer. The clearest proof is that a vanilla-canvas browser renderer and a matplotlib window now sit on top of the same six modules, neither of which the data layer knows about. Everything else—recentering, the power-law color scale, the level-one trade derivation, wall/iceberg detection, the analytics panels, multi-symbol grids—is built on that seam. What's left is mostly polish and one honest gap (a real trade tape), and both are reachable without disturbing the core. The tool does the thing it was built to do: it lets you *see* the liquidity.

---