LiveTape.ai

William Huynh

Production: https://app.livetape.ai **Landing Page:** https://livetape.ai

Code sample: https://gist.github.com/williammh/a1ccdc038f186247220c070aee1ed974

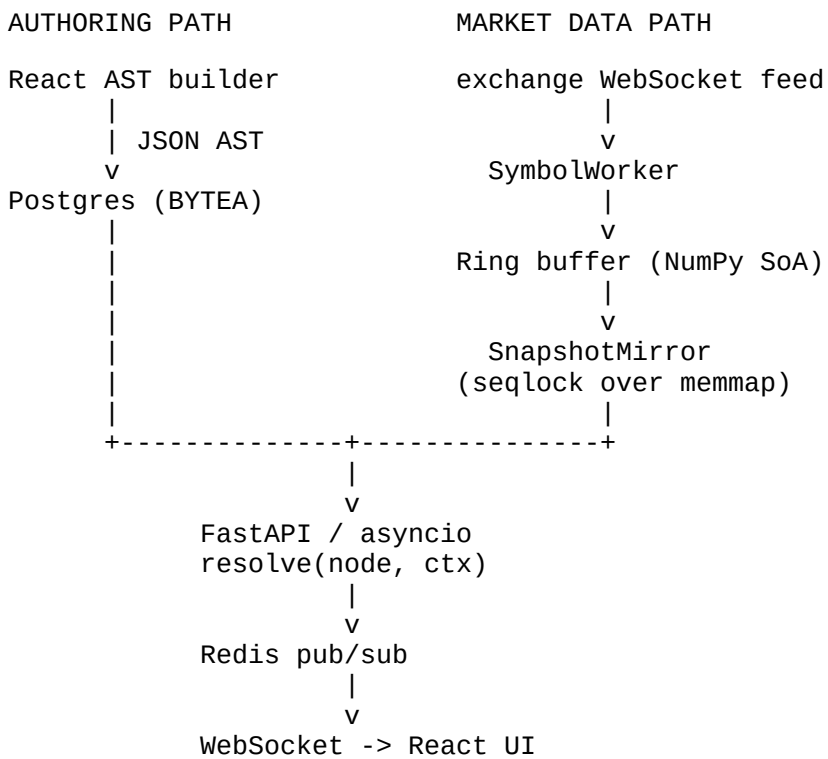
What it does

Any LLM can a trading strategy. Running it, correctly, live, against a live market data, without falling over, is a different problem, and that's where most of the engineering effort went: getting market data in fast and reliably, keeping time semantics sane when the network jitters, coordinating between processes without locking anything up, running multiple strategies simultaneously, and storing strategies in an extensible format.

Every tick gets evaluated, not just the close of each 1 minute period the way most no-code trading tools do. Internal latency is under 50ms: tick in, through the ring buffer, through a lock-free snapshot, through the AST, decision out, no database call anywhere in that path. What happens after that (how fast the broker fills the order, any signal provider upstream) is downstream of the engine and not counted here.

How it's wired together

Two things happen mostly independently and meet at evaluation. On one side, a user authors a strategy visually in the browser and it gets stored. On the other, live market data streams in continuously and gets kept warm in shared memory. Once a bar closes, the engine pulls both together and runs the strategy.



The left side only runs once, when an agent starts up. The right side never stops, and it crosses over into the evaluation process at SnapshotMirror without ever taking a lock.

Design tradeoffs

The market-data handoff is lock-free. Two processes need to share one continuously-updating OHLCV table: SymbolWorker ingests a live feed and writes it dozens of times a second; AgentWorker reads it just as often to evaluate every running strategy, in a separate process. A mutex was ruled out because the writer is on the critical path of a live feed and can't afford to ever wait on a slow reader. A queue was ruled out because readers only ever want the latest state, not a backlog of every update in between. A database round-trip is orders of magnitude too slow for a per-tick budget. Data comes in over WebSocket and gets bucketed into a fixed-size NumPy ring buffer: plain arrays, O(1) push, nothing ever shifts. The strategy-evaluation process reads that buffer through a memory-mapped file, both sides coordinating with a seqlock: the writer marks a counter odd before it starts overwriting the table and even when it's done, and a reader that sees an odd counter, or sees the counter change while it was copying, just discards what it read and retries. Neither side ever blocks the other. There's no shared-memory object and nothing brokering it; the file on disk is the boundary.

The one subtlety is that a seqlock like this depends on CPU store ordering that Python never states explicitly, and the two platforms this runs on don't give it the same guarantee for free — x86-64 does, ARM64 (the production instance) doesn't. It's safe in practice for a specific reason: the writer's last column write and its counter update are separated by real interpreter work — NumPy teardown, refcounting, bytecode dispatch — and that incidental overhead acts as the memory barrier the protocol needs, even though nothing asked for one. That argument holds only as long as CPython sits between those two writes, which is a real constraint on future changes (porting the writer to Cython or a C extension would remove it) and not one that shrinking the write size affects. The full reasoning, the retry counters, and a runtime coherence check that turns "this has never gone wrong" into something actually measured all live at the top of snapshot_mirror.py, next to the code it's protecting.

The frontend never runs any trading logic. A condition built in the visual editor (SMA(50)[1] > SMA(200)[1]) gets turned into a JSON tree (an ExpressionNode), and that's all the frontend ever does with it: build the tree, render the tree. The backend's resolve(node, ctx) walks that exact same tree shape recursively. Both sides check the tree against the same schema (Zod on one side, Pydantic on the other), so there's one grammar and the two halves can't drift apart from each other.

Bar references are relative, not absolute. An indicator doesn't point at a fixed slot in an array. It's parent_offset + node.bar_index, and that adds up as the tree nests. So SMA(50)[1] inside a bigger expression means "one bar back from wherever the parent landed," and the same piece of logic can be reused at a different offset without touching it. This one rule is what the whole resolver hangs off of.

The clock is deterministic, never time.time(). Every evaluation runs against ctx.current_time_s, an internal clock that only moves forward and only moves in response to real events (a bar closing, a tick, or, if the market goes quiet, a synthetic heartbeat). That's what lets backtests and live runs share the exact same code path: in a backtest the clock is just set straight from the replayed timestamps instead. Nothing in the resolver ever calls the system clock, which is the only reason a backtest is reproducible at all.

No Python loop ever touches the full buffer. There's a rule on this project: nothing may loop in plain Python over a full ring-buffer-sized array (172,800 elements) on the event loop. A loop that size would block long enough to time out any HTTP request coming in at the same moment. So every bulk operation on market data is vectorized NumPy: fancy indexing, reduceat, slice assignment. Python glues things together; the array math runs in C.

A strategy exists in three formats on purpose. It's authored as a plain dict over the API, stored as UTF-8 JSON bytes in Postgres, and converted once at agent start into a np.uint8 array, the form the engine actually runs on. Three small, tested functions sit at that boundary (rules_to_bytes, rules_from_bytes, rules_to_uint8), so the storage format, the wire format, and the in-memory format can each change on their own without a migration touching the other two.

One event loop runs up to 50 strategies at once. Each one gets its own runtime state and never hits the database on a per-cycle basis. State loads once when the agent starts and just lives in memory after that. Logs get written out asynchronously, off to the side, so they don't slow anything down.

Stack

Backend: Python 3.13, FastAPI, asyncpg, SQLModel, Redis, NumPy. Frontend: TypeScript, React, Zustand, Zod, lightweight-charts. Docker Compose for dev/preview/prod, running on OCI ARM64, with Cloudflare Workers serving the static frontend.

Testing

The backend tests focus on the stuff that's actually easy to get wrong here: does the seqlock correctly detect a torn read, does the bar pointer move to the right slot after a close, does the clock always move forward and never jump backward, does a crashed publish clean up after itself and recover. There's also a set of guided end-to-end flows that run in Playwright against the real UI.