TECHNICAL REPORT

Tradeski: A Real-Time Market Dashboard

Rate-limit budgeting, indicators correct at their boundary conditions, and grounding a language model in one user's actual portfolio.

Daniel Lichtenberger · 2026
tradeski.dev · Source on GitHub

A real-time market dashboard where the interesting engineering is everything except the charts.

Summary

Tradeski streams live prices, macroeconomic series, and scored news into a single browser dashboard, and answers questions about the user's own portfolio rather than the market in general. Plotting a price series is a solved problem. The parts that took real work are the ones a screenshot doesn't show: staying inside third-party API rate limits, computing indicators that are correct at their boundary conditions, and grounding a language model in a specific user's holdings so its answers are about them.

This report describes the architecture as built, the decisions worth defending, and — in §9 — the things it does not do and the places I would build it differently.

Backend	Python 3.12, Flask + Flask-SocketIO on gevent, 25 HTTP endpoints
Frontend	Static HTML/CSS/JS, 2,299 lines, Plotly.js, no framework and no build step
Indicators	10, hand-implemented on NumPy arrays — no TA-Lib, no pandas
Tests	75, across 11 files, run in GitHub Actions with flake8
Deployment	Backend on Render (gunicorn + gevent), frontend static, Postgres

1. The problem: a dashboard is a rate-limit budget

A market dashboard looks like a data-visualization problem and is actually an API-quota problem. Every panel wants fresh data; every upstream provider meters requests; and the naive implementation — fetch on every page load — exhausts a free-tier news or macro quota within an hour of a handful of users arriving.

So the real design question is not *how do I draw this* but *how stale is each number allowed to be*, and the answer differs per data source by three orders of magnitude. A price can be a minute old. An unemployment rate cannot meaningfully change within a month. Treating those identically is the mistake.

Tradeski assigns every data source its own time-to-live, chosen from how fast the underlying quantity actually moves:

Source	TTL	Why
Price history / stats	5 min	Intraday movement matters; a minute of staleness does not
Screener fundamentals	10 min	Ratios update on filings, not ticks
News + sentiment	30 min	Headline flow is slow; scoring is the expensive part
FRED macro series	1 hour	Most series are monthly or quarterly
Correlation matrix	1 hour	Computed over 90 days; a fresh day barely moves it

The cache itself is deliberately unremarkable — a dict of `(value, expires_at)` with lazy eviction on read (`cache.py`). Its limitations are real and are discussed in §9. What matters is that the TTL is a per-source decision rather than a global constant, and that per-route rate limiting sits in front of the expensive endpoints: registration at 10/hour, login at 10/minute, the assistant at 20/hour and 50/day.

The screener fans out across a 29-symbol universe — 27 stocks spanning technology, financials, healthcare, energy and consumer staples, plus SPY and QQQ — through an eight-worker thread pool, so 29 sequential network round-trips become roughly four waves.

---

2. Architecture

```
browser (static HTML/JS, Plotly.js)
    │
    ├── HTTP  ──►  Flask app  ──►  SimpleCache  ──►  upstream APIs
    │                  │                             (prices, FRED, NewsAPI)
    │                  ├── Postgres  (users, portfolios, alerts, price history)
    │                  └── Anthropic API  (assistant, grounded — §7)
    │
    └── Socket.IO  ◄──  background tracker task (price_update events)
```

The frontend holds all UI state and is served as static files; the backend is a pure JSON and event layer with no templating. That split is what allows the frontend to deploy independently of the backend, and it keeps the server free of rendering concerns entirely.

Two entry-point details are load-bearing. `wsgi.py` calls `gevent.monkey.patch_all()` as its literal first statement, before any other import — patching the standard library's sockets after `requests` has already bound them produces a server that appears to work and then deadlocks under concurrency. And because the backend package directory contains a space in its name (`Plotly dashboard/`), it cannot be imported as a module; `wsgi.py` inserts it onto `sys.path` explicitly so the process works regardless of working directory.

---

3. Indicators from first principles

Every technical indicator is implemented directly on NumPy arrays rather than pulled from TA-Lib or pandas: SMA, EMA, RSI with Wilder smoothing, MACD, Bollinger Bands, ATR, the Stochastic Oscillator, rolling Z-score, rolling volatility, and a least-squares linear regression forecast (`analyzer.py`).

The point of writing them by hand is not that the libraries are wrong. It is that the boundary conditions are where indicators are actually wrong, and you only meet them if you write the code.

RSI when there are no losses. Relative Strength is `avg_gain / avg_loss`. When the average loss is zero, RS diverges and the standard formula divides by zero. The common shortcut is to set `rs = 0`, which inverts the meaning: a window of pure gains — maximally overbought — reports RSI 0, maximally *oversold*. Tradeski handles the case explicitly: RSI is 100 when there were gains and no losses, and a neutral 50 for a perfectly flat window where both averages are zero.

The Stochastic Oscillator needs highs and lows. %K is defined against the intraday high and low of the lookback window. The real-time tracker stores closing prices only. Feeding the close in as all three of high, low and close makes the formula degenerate — the numerator and denominator collapse to the same quantity and %K pins to a constant. Rather than emit a plausible-looking number, `analyze_series` returns `stoch_k` and `stoch_d` as `None` when highs and lows were not supplied, and the dashboard omits the panel. A missing indicator is honest; a fabricated one is not.

---

4. Portfolio risk metrics

Risk metrics are computed on the user's actual holdings, weighted by current market value, from a year of daily returns.

The portfolio return series is built as the value-weighted sum of each holding's daily returns, aligned to a common length so that a symbol with a shorter history does not silently shift the series. From that: annualized volatility as the daily standard deviation scaled by $\sqrt{252}$; the Sharpe ratio against a 4.5% annual risk-free rate converted to a daily figure; and beta as `cov(portfolio, SPY) / var(SPY)`.

Two guards matter. A portfolio with zero return variance — a single holding entered today, or an untraded position — makes the Sharpe denominator zero, so the endpoint reports that it cannot be computed rather than dividing. And beta falls back to 1.0 if SPY's variance is degenerate. Both return an explicit reason to the client instead of a number.

The 90-day correlation heatmap covers 11 tracked symbols, computed as Pearson correlation across their aligned daily return series — not price levels, which would report near-1.0 correlation for any two assets that both drift upward.

---

5. News sentiment: why general-purpose scoring fails here

Sentiment is VADER, extended with roughly 45 finance-specific terms (`news.py`).

The extension is not decoration. VADER is tuned on social-media English, and financial headlines use a register it has no entries for. "Apple beats expectations" scores 0.0 out of the box — *beats* is not in the lexicon in its financial sense, and the headline reads as neutral. Likewise "misses", "downgrade", "buyback", "record high". The single most important event type in equity news is invisible to the default model.

So the lexicon adds the vocabulary with hand-assigned polarities: `beats +2.0` , `all-time high +2.2` , `downgrade -1.8` , `bankruptcy -3.0` , `tariffs -1.2` . Headlines are scored on title plus description, aggregated to a mean compound score with bullish / neutral / bearish counts at a ±0.05 threshold.

Query construction matters too. NewsAPI searched for the bare ticker `META` returns articles about metadata and metaphysics; searching `"Meta stock"` does not. Each tracked symbol carries an explicit descriptive query string rather than its ticker.

This is a hand-tuned heuristic, not a validated model. The polarities are my judgment, not fitted values, and there is no labeled financial-headline set here measuring whether the extended lexicon actually beats the default. It plainly fixes the specific failures above; the general claim is untested, and §9 says so.

---

6. Security

Passwords are hashed with PBKDF2 via werkzeug. Sessions are signed bearer tokens issued with `itsdangerous` under a versioned salt, so tokens can be invalidated as a class by rotating it. Every portfolio, alert, and watchlist query is scoped by `user_id` at the database layer rather than filtered after retrieval — the difference between a bug that returns nothing and a bug that returns another user's holdings. CORS is restricted to an explicit origin allowlist from the environment rather than a wildcard.

---

7. The assistant, and what "grounded" means

The dashboard includes Ski, a Q&A assistant backed by Claude. The engineering that makes it useful is not the prompt; it is what goes into the context window before the prompt.

A general financial chatbot can explain what a P/E ratio is. That is worth very little inside a dashboard, because the user is looking at their own screen and their question is about *their* position. So each request assembles a context block first: the user's current holdings with market values, the live macro snapshot from FRED, and the scored news headlines for the relevant symbols — each formatted by a dedicated `format_*_context` function. Only then is the model called.

The result is that "am I overexposed to tech?" is answered against the user's actual allocation. The model is doing language work; the application is doing the retrieval.

---

8. Testing

121 tests across 16 files run on every push through GitHub Actions, alongside flake8 linting. Coverage is weighted toward the code where correctness is checkable rather than the code that is merely tedious: indicator boundary conditions, authentication and token handling, database access and user scoping, the FRED and news clients against recorded payloads, the screener, and portfolio math.

Network-dependent modules are tested against fixtures, not live APIs. A test suite that fails because a third-party service is down teaches you nothing and trains you to ignore it.

That style of testing has a blind spot, and it cost this project its headline feature. The background tracker emitted `price_update` on the `/stream` namespace while the browser connected to the default namespace and listened there. Each half was correct read on its own; the socket connected, the UI reported a healthy connection, and no tick ever arrived — prices moved only on the 60-second polling refresh. Every test passed the entire time, because every test exercised a pure function in isolation. A contract *between* two components is invisible to that.

`tests/test_stream_contract.py` now parses `app.py` and `dashboard.js` and asserts that every event the client subscribes to is emitted on the namespace the client is actually on. Worth recording that the first version of that test also passed with the bug present: it compared the *set* of namespaces each event was emitted on, and a dead code path that emitted `price_update` correctly on the default namespace masked the live one that did not. Checking each emit site individually, rather than unioning them per event, is the whole difference between a test that catches this and a test that looks like it does.

---

9. Limitations

Stated plainly, because a report that lists only strengths is marketing.

The cache is per-process and in-memory. It does not survive a restart, and under multiple gunicorn workers each holds an independent copy — so the effective upstream request rate is the intended rate multiplied by the worker count, and two users can see figures that differ by one TTL. Correct for a single-worker deployment, which is what runs; wrong the moment it scales horizontally. Redis is the obvious fix and is not implemented.

The risk-free rate is hardcoded at 4.5%. It should come from the FRED series the application is already fetching. Every Sharpe ratio is therefore slightly wrong in a direction that changes with the rate environment.

Sentiment is unvalidated. See §5. The lexicon extension is a fix for named failures, not a measured improvement.

There is no backtesting and no strategy evaluation. Tradeski shows you the present. The question of whether any indicator on the screen predicts anything is deliberately out of scope here — that question is the entire subject of AlphaGlyph, and the honest answer it arrives at is mostly "no". Reading the two projects together is the intended experience: this one displays the indicators, that one measures whether displaying them means anything.

`tracker/main.py` is dead code. It is a legacy standalone CLI tracker from before the web dashboard existed, and it no longer imports — it references a `dashboard` module that does not exist under that name. It is 467 lines of the repository that no execution path reaches, and it should be deleted rather than left to imply it runs.

The linear-regression "prediction" is a trend extrapolation, not a forecast. It fits a least-squares line to the recent window and reads off the next point. It is labeled as a prediction in the UI, which oversells it; it carries no error bars and no evaluation of whether it beats the naive "tomorrow equals today" baseline. It almost certainly does not.

---

10. What I would do differently

Move the cache to Redis and make TTLs configuration rather than constants. Pull the risk-free rate from FRED. Delete `tracker/main.py`. Either validate the sentiment lexicon against a labeled set or relabel the panel as a heuristic. And either give the regression forecast an honest error estimate or remove it — of everything on the dashboard it is the element that most looks like a claim and least is one.