KSE | Kyiv School of Economics

Computer Science Department

Software Engineering and Business Analysis (SEBA-26)

**Bachelor's Thesis**

# Multi-Exchange Trading Journal for Cryptocurrency Markets

Tradeline: an affordable, automated analytics platform

Myroslav Ostashchuk

*Supervisor*
KSE, Mykhailo Moroz, m.moroz@kse.org.ua

*Submission date*
16 June 2026

# Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received (e.g. from peers, tutors, or online forums) is acknowledged in the acknowledgements section.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

| | |
|---|---|
| Place, date | Kyiv, 04.06.2026 |
| Signature | _____ |

# Contents

# Abstract

Most retail crypto traders work across several exchanges, and the tools meant to help them review their performance are either expensive or built for many asset classes and overloaded with features. This thesis presents Tradeline, a crypto-native trading journal that competes on price and ease of use rather than on the number of features.

Tradeline connects to Binance, Bybit, OKX and Bitget through read-only API keys, imports a trader's available history over REST and keeps it current over WebSocket, reconstructs the raw trades into positions, and turns those positions into the analytics a trader needs to judge and improve their results. It is built as a complete web application: an asynchronous FastAPI backend with a feature-sliced architecture, PostgreSQL with the TimescaleDB extension for the time-series data, a runtime analytics layer that avoids recomputing on every change, and a React single-page frontend, all deployed to production behind a pool of egress proxies.

The result is a working, deployed system: it gives an intermediate crypto trader a complete, automated journal, with the full analytics free and a low monthly price only for the limits that power users reach. The thesis covers the market analysis that motivated the design, the architecture and the trade-offs behind it, the implementation of the exchange integration, the positions engine and the analytics layer, and the validation of the system against its requirements.

***Keywords***:

*Trading journal*, *Cryptocurrency*, *Multi-exchange*, *FastAPI*, *TimescaleDB*, *Trading analytics*

# 1 Introduction

Cryptocurrency trading has grown from a niche activity into a market with hundreds of millions of participants. By 2024 around 562 million people owned cryptocurrency, close to 6.8% of the world's population and a third more than the year before [1]. Many of them trade, not just hold, and they rarely do it on a single venue. Liquidity, fees and the list of available pairs differ from one exchange to the next, so an active trader usually ends up with accounts on several of them at once.

That habit creates a simple but stubborn problem. Each exchange reports only its own activity, in its own layout, with its own notion of what a trade or a position is. A trader who wants one honest answer to "am I making money, and where?" has to pull statements from every account and stitch them together by hand, or pay for a dedicated tool. The cost of not having that answer is high. A 2024 survey of 1,005 retail crypto traders found that 84% lost money in their first year, and 58% lost almost all of it [2]. A journal does not fix a bad strategy on its own, but without a clear record of what worked and what did not, a beginner keeps repeating the same mistakes.

Tools that close this gap already exist, and they tend to sit at two awkward extremes. Multi-asset platforms such as TraderSync cover stocks, options, futures, forex and crypto together. That breadth makes them capable and, for a crypto-only trader, overloaded: hundreds of charts and settings that get in the way more than they help. Crypto-specific journals such as Coin Market Manager are far more focused, but expensive, with a paid plan near seventy dollars a month and free access limited to a single exchange [3]. For someone who is still learning, and by the numbers probably still losing money, that price is a barrier to even starting.

This thesis takes a deliberately narrow position. It does not try to beat the established products on feature count or on the number of supported exchanges. A single developer cannot out-build a funded company on that axis, and pretending otherwise would be dishonest. The argument is that the underserved need is different: an affordable, crypto-native journal that gives an intermediate trader the analytics they need, automatically, across their exchanges, in an interface that is pleasant to use. The concrete result of the work is Tradeline[1], a web platform that connects to Binance, Bybit, OKX and Bitget with read-only API keys, imports both historical and live trades, reconstructs them into positions, and turns them into the analytics a trader uses to see what is working and what is not. Its free tier gives away the full analytics, and the paid tier asks a low monthly price only for the limits that power users reach, instead of putting the basics behind a paywall as most of its competitors do.

The platform is built as a full web application, and the thesis treats it as an engineering project from end to end. The backend is an asynchronous FastAPI service with a clean, feature-sliced structure, backed by PostgreSQL extended with TimescaleDB for time-series data. The frontend is a React single-page application. The system talks to live exchange APIs over both REST and WebSocket, routes its outbound traffic through a pool

[1] Source code: https://github.com/justmiroslav/tradeline

of proxy servers, and runs in a production deployment rather than only on a developer's machine.

## 1.1 Objectives

This work set out to design, build and deploy a trading journal for the underserved case identified above: an intermediate crypto trader who needs automatic, multi-exchange analytics at a price they can justify before they are profitable. That aim breaks down into four concrete objectives:

- integrate with four major crypto exchanges through read-only API keys, importing both historical and live trades without manual work;
- reconstruct raw trades into positions and derive a complete set of performance analytics from them;
- present those analytics in a clean, responsive interface backed by a free tier that is a usable product in its own right;
- engineer the system to production standards, secure, within exchange rate limits, maintainable and actually deployed, rather than as a prototype.

Each objective is met in the system as built, and the chapters that follow show how. The work is therefore as much an exercise in system design and engineering as in product thinking, and the thesis is presented in that spirit.

## 1.2 Structure of the thesis

The remaining chapters follow the path of the project. Section 2 surveys the existing solutions, looks closely at the main competitors, and turns the gap between them and the trader's needs into a set of requirements. Section 3 presents the design: the architecture, the technology choices, the data model and the deployment topology. Section 4 describes how the system was implemented, with attention to the exchange integration, the positions engine and the analytics layer. Section 5 checks the result against those requirements. Section 6 summarises the work and outlines what comes next.

# 2 Research

The demand for crypto trading journals is large, and the tools that serve it are either expensive, overloaded, or dated. This chapter examines that gap closely and turns it into a clear set of requirements.

## Contents



## 2.1 The case for a trading journal

A trading journal is the record a trader keeps of their own decisions: every position taken, when and why it was opened, and what it returned. Put that way it sounds dull, but it is the difference between trading on memory and trading on facts, and that difference matters: most retail crypto traders lose money in their first year, and most never learn which of their own habits were responsible. An exchange shows a balance and a list of open orders; it does not show that a trader makes money on one pair and slowly loses it all back on another, that the losing positions are the ones held overnight, or that the win rate falls apart after a winning streak. Those patterns only surface once trades are gathered, rebuilt into positions, and measured against each other.

What turns that record into something useful is the analytics layer on top of it: realised profit and loss over time, win rate set against the average win and average loss, and performance broken down by symbol, by side, by hour of day and by holding time. None of this asks the trader to do anything except trade, because it is derived entirely from the history they already generate. So a good journal is less a notebook and more a tool that shows a trader their own behaviour in a form they can act on.

There is one more difficulty: a trader rarely stays on a single venue, so this record has to be pulled together from several exchanges at once, each with its own format. Doing that by hand means exporting statements from every account and matching them up in a spreadsheet, which is tedious enough that most people give up within weeks. Doing it automatically and all the time, so the analytics are always just there, is the core job of the product.

## 2.2 Where existing tools fall short

The products in this market are mature, and several are genuinely good at what they do. Yet across them and their user feedback the same three weaknesses recur, and together they explain why so many traders still fall back on a spreadsheet.

The first is price. The crypto-native incumbent Coin Market Manager charges around seventy dollars a month for its main paid plan, and the established desktop journal Edgewonk asks for a lump sum of close to two hundred dollars a year [3], [4]. The multi-asset tools sit in the same range. Where a free tier exists at all it is usually a short trial or a single-account cap, not a state a trader can settle into. For a beginner who is, by the numbers, probably losing money, paying a monthly subscription before seeing any value is a barrier to even forming the habit.

The second is complexity. The broad platforms try to serve stocks, options, futures, forex and crypto from a single interface, and that breadth has a cost. A trader who only touches crypto is handed hundreds of statistics and settings built for asset classes they will never use, and the interface becomes something to study rather than something that answers a question. What looks like power on a feature list just becomes noise when you use the tool.

The third is the interface itself, and it is not confined to the multi-asset tools. Trader Make Money, the closest crypto-native competitor, is respected for what it computes but criticised for how it feels; reviewers describe its interface as dated and harder to learn than it should be [5]. Edgewonk, despite a devoted following, still behaves like a desktop application from the late 2010s and depends on manual entry or CSV import because its live integrations are thin [6]. In both cases a trader new to journaling spends their first hour fighting the tool instead of reading their results.

Table 1 sets the four closest products side by side. Because the choice a trader actually faces is whether to pay month to month or commit to a year, it lists both the monthly cost and the effective monthly cost when billed annually.

| Product | Focus | Free tier | Monthly | Annual |
|---|---|---|---|---|
| Trader Make Money | Crypto | 1 exchange | $9-29/mo | $6-22/mo |
| Coin Market Manager | Crypto | 1 exchange | $70-90/mo | $58-75/mo |
| TraderSync | Multi-asset | Trial only | $30-80/mo | $22-60/mo |
| Edgewonk | Multi-asset | Trial only | n/a | $16/mo |
| Tradeline | Crypto | Full analytics | $8.99/mo | $7.50/mo |

*Table 1: Crypto-relevant trading journals: per-month cost on monthly versus annual billing*

## 2.3 The competitive landscape

Each of these four is instructive for a different reason. Trader Make Money is the closest peer and the hardest one to beat. It does the hard part right: it connects to crypto exchanges and imports trades automatically, and it offers a wide range of features at a genuinely low price, from a free tier up to roughly twenty to thirty dollars a month at the top [7]. It cannot be beaten on features or on price alone. Its weakness is the one a single developer can realistically address, which is the experience: the interface is capable but unfriendly, and the free tier is thin. Tradeline's answer is not more features but a cleaner interface and a free tier that gives away the full analytics rather than a preview.

Coin Market Manager illustrates the price axis in its sharpest form. It is competent and crypto-native, with real-time portfolio tracking and automatic imports, but its paid plans run from about seventy dollars a month to ninety, and even billed annually they stay near sixty [3]. That positions it for established traders who already make money and treat the subscription as a cost of business, not for the much larger group still trying to become profitable. The contrast with an under-nine-dollar plan is the entire point.

TraderSync shows the complexity problem. It is a polished, professional journal, but a multi-asset one: crypto is a single tab among stocks, options and futures, and its plans run from thirty to eighty dollars a month [8]. A crypto-only trader pays both in money and in attention for breadth they do not need, and there is no lasting free tier to ease them in. It is strong evidence that capability and fit are not the same thing.

Edgewonk shows the limits of a manual, desktop workflow. Its journaling depth and trading-psychology focus are real, and at a flat fee of a hundred and ninety-seven dollars a year it is, measured per feature, inexpensive [4]. But it is a desktop-era product: crypto is a secondary asset, the integrations are thin, and getting data in usually means manual entry or CSV files rather than a live connection. It answers a different question - deep manual review for a disciplined trader - not the automatic, crypto-first one this thesis is about.

Together, the four show where Tradeline fits. It matches the automatic crypto sync of Trader Make Money and Coin Market Manager, avoids the multi-asset overload of TraderSync, and replaces Edgewonk's manual, dated workflow with a live and modern one. Above all it gives the full analytics away for free and charges under nine dollars a month only for the limits that heavy users reach, where every competitor either charges from the start or restricts the basics. The bet of this work is that price, fit and interface, not raw feature count, are where this market is underserved.

## 2.4 The target user

All of this is aimed at one person in particular. The user Tradeline is built for is an intermediate crypto trader: someone in their twenties who has been trading for a year or two, mostly perpetual futures on two or three exchanges, with a modest portfolio and a win rate sitting just below break-even. They have already tried the alternatives, a spreadsheet they found too tedious to maintain and one or two paid tools they found either too expensive or too complex, and what they want is a simple, crypto-native product with a modern interface that they can start using for free before committing any money. They are

technical enough to create a read-only API key when shown how, but they will not read a manual. Every requirement below exists to serve this person rather than a professional desk or a multi-asset trader.

## 2.5 Requirements

The analysis turns into two sets of requirements.

The functional requirements describe what the system must do:

- connect to Binance, Bybit, OKX and Bitget through read-only API keys, covering both spot and perpetual-futures accounts;
- import each user's available trade history, up to three months back, and then keep it current from live exchange feeds without any manual export;
- reconstruct raw trades into positions per exchange and market, and compute a full set of performance analytics from them;
- present those analytics through a dashboard, a journal of trades and positions, and a daily calendar, all filterable by exchange, symbol and date;
- support user accounts and two subscription plans, where the free tier includes the full analytics rather than a limited preview.

The non-functional requirements describe how it must behave:

- never hold the right to move funds: keys stay read-only and encrypted at rest, and the system rejects any key that carries trade or withdrawal permissions;
- stay affordable, with a free tier that is a complete product and a single paid tier under nine dollars a month;
- respect each exchange's rate limits while many users and exchanges synchronise at once;
- present a clean, responsive interface that an intermediate trader can navigate without instructions;
- rest on a clean, maintainable architecture and run as a deployed, publicly reachable service.

These requirements are what the design has to satisfy. The next chapter turns them into an architecture and the decisions behind it.

# 3 Design

This chapter presents the design of Tradeline: how its parts are arranged, the patterns that hold them together, and the decisions behind each, with their trade-offs. It works from the system as a whole down to the single components, and explains every choice as an answer to a requirement from Section 2, not on its own.

## Contents



## 3.1 Architecture overview

Tradeline is two applications and a few supporting services. A React single-page application is the interface, an asynchronous FastAPI backend does the work, and a PostgreSQL database with the TimescaleDB extension stores the data. The backend reaches the four exchanges through a pool of proxy servers, and calls two outside services: NVIDIA NIM for the AI insight and Resend for email. Figure 1 shows these parts, where they run, and how they talk.

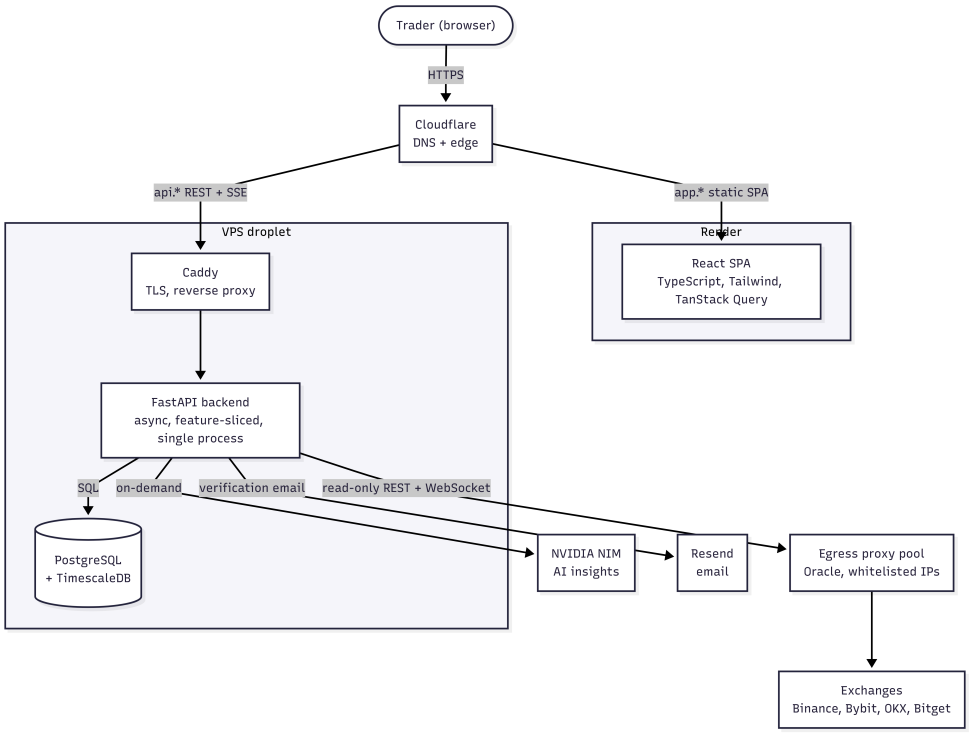


*Figure 1: System overview: the applications, where they run, and the services they depend on*

Both applications are built the same way: as feature slices over a small shared core. This is usually called clean, or layered, architecture. Each feature - authentication, API keys, trades, positions, analytics, plans, the AI insight - is its own slice, with its own routes, logic and data access. A slice depends only on the small shared core, not on the other slices. On the backend a slice reaches the database only through a repository, and shares one domain layer for the models, the fixed value sets and the configuration. On the frontend the same split holds the pages, hooks and components, over a shared interface kit. This is why the style was chosen: a change to the analytics view stays inside one slice, each slice can be read and tested on its own, and a new feature is a new slice instead of edits all over the code.

## 3.2 Backend structure and patterns

The backend does two kinds of work, and the design keeps them apart. The first is serving a request. A request enters a feature router, which checks the input and calls a service. The service holds the logic and reaches the database only through a repository, and the repository is the only place that writes SQL. Figure 2 shows this path. Because only the repository touches the database, a query can be changed or sped up in one file, without touching the logic above it. Every slice follows this same router - service - repository order, so the code looks the same wherever you open it.

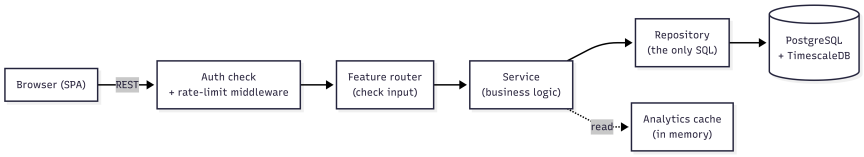


*Figure 2: Serving a request: router, service, repository, database*

The second kind of work runs in the background, for the whole life of the process. Figure 3 shows it. The exchange side is one package: a gateway that gives out a client per exchange, the per-exchange clients and connection handlers, the proxy and rate limiter, and a cache of the tradable symbols. Three workers run all the time: one keeps API keys healthy and synced, one holds a live connection per key, and one fetches market data and adds context to positions. A trade ingester sits in the middle and turns incoming trades into stored trades, positions and analytics updates, and an event bus pushes those changes out to the browser. All of these are built once at startup and shared through the application's state. This is the framework's form of dependency injection: no part reaches for a global, and each request or worker is given exactly the parts it needs. Because the workers hold all the live state - the connections, the buffers, the in-memory caches - the backend runs as one process on purpose.

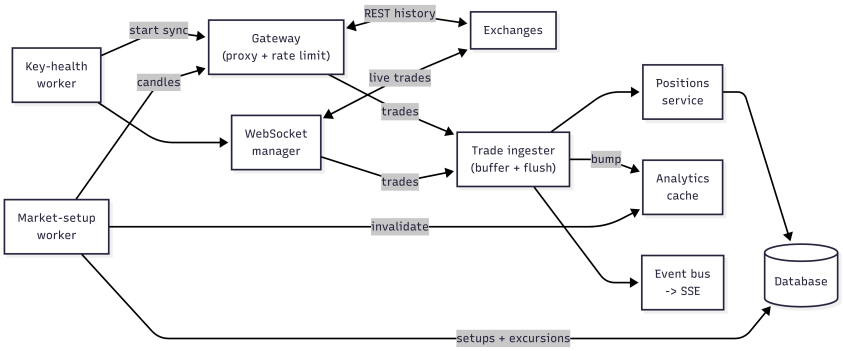


*Figure 3: The live path: how data flows in from the exchanges to the browser*

## 3.3 The API surface

The backend offers the frontend a small, normal REST interface, split by the same features as the code. Table 2 lists the endpoints. The shape is easy to guess: a list lives at its plural noun, one item at that noun plus an id, and the HTTP verb says what happens - GET reads, POST creates or runs an action, PATCH edits, DELETE removes. Lists are paged and filtered through query parameters, and the two export endpoints stream their result as a CSV file instead of building it all in memory.

Login draws a clear line through the table. A few public endpoints handle sign-up, email verification, login and token refresh; everything else needs a bearer token, checked once before the route runs. A few endpoints are not resources and sit a little apart: a stream the browser keeps open to get live updates, a small public endpoint that returns the IP addresses a user must whitelist, and a health check. The plan endpoints are where billing will attach later: changing a plan works today, but the payment step behind it is future work.

| Method | Path | Purpose | Auth |
|---|---|---|---|
| **Authentication** | | | |
| POST | `/auth/register` | Begin sign-up, email a code | no |
| POST | `/auth/register/verify` | Confirm the code, issue tokens | no |
| POST | `/auth/login` | Log in | no |
| POST | `/auth/refresh` | Refresh the access token | no |
| POST | `/auth/logout` | Revoke the session | yes |
| GET | `/auth/me` | Current profile | yes |
| PATCH | `/auth/me` | Rename the account | yes |
| DELETE | `/auth/me` | Delete the account | yes |
| **API keys** | | | |
| GET | `/api-keys` | List connected keys | yes |
| POST | `/api-keys` | Add a key | yes |
| GET | `/api-keys/{id}/sync-progress` | Poll the first sync | yes |
| POST | `/api-keys/{id}/recheck` | Revalidate and resync | yes |
| PATCH | `/api-keys/{id}` | Rename a key | yes |
| DELETE | `/api-keys/{id}` | Remove a key | yes |
| **Trades and positions** | | | |
| GET | `/trades` | List trades, filtered | yes |
| GET | `/trades/symbols` | Symbols the user has traded | yes |
| GET | `/trades/range` | First and last trade dates | yes |
| GET | `/trades/export` | Stream trades as CSV | yes |
| GET | `/positions` | List positions, filtered | yes |
| GET | `/positions/export` | Stream positions as CSV | yes |
| **Analytics, plans and AI** | | | |
| GET | `/analytics/overview` | The full analytics payload | yes |
| GET | `/plans` | List the plans | no |
| PUT | `/plans/me` | Change the current plan | yes |
| GET | `/ai/insight` | Last insight and cooldown | yes |
| POST | `/ai/insight` | Generate a new insight | yes |
| **Live and service** | | | |
| GET | `/stream` | Live event stream over SSE | yes |
| GET | `/info` | Egress IPs to whitelist | no |
| GET | `/health` | Liveness probe | no |

*Table 2: The backend HTTP endpoints, grouped by feature*

## 3.4 Exchange integration

The four exchanges are alike enough to share one interface, and different enough that those differences shape much of the design. Each exchange is wrapped in a client that follows one common contract: validate a key, list trades, fetch candles. The rest of the backend asks the gateway for a client and works against this contract, never against a named exchange. This is the adapter pattern: the exchange-specific code is hidden behind a fixed shape, and the gateway is the facade that hands out the clients. Adding a fifth exchange means writing one more client, not changing the callers.

The differences live under the contract. Sign-in differs: two exchanges sign a request with a keyed hash over the query, the other two also add a passphrase. Paging the history differs too, and there are two ways to do it. Most exchanges have one trade-history endpoint that you walk with a cursor; one of them does not, so its history is collected by scanning the symbols a user actually traded - which is why the system keeps a cached list of that exchange's symbols and scans only a few at a time. Realised profit comes inline from some exchanges and from a separate call on others. Each client hides all of this and returns one normalised trade shape, so everything after it - positions and analytics - is written once and works the same way, whatever exchange the trade came from. All the variety between exchanges ends at this one layer.

## 3.5 Egress: proxy and rate limiting

The backend does not call the exchanges directly. It goes out through a pool of proxy servers, for two reasons that both matter. The first is identity. An exchange lets a user lock a key to certain IP addresses, and the proxies are the fixed, whitelisted addresses the backend goes out from. So the system can ask a user to whitelist a small, stable set of IPs, instead of a server address that might change later. The second reason is rate limiting, on the same path. Every outgoing call must take a token from a bucket before it leaves. The bucket is a token bucket sized to the exchange's published limit, and it also reads the rate-limit headers the exchange sends back and sets its own count to what is really left, so it slows down before the exchange refuses, not after.

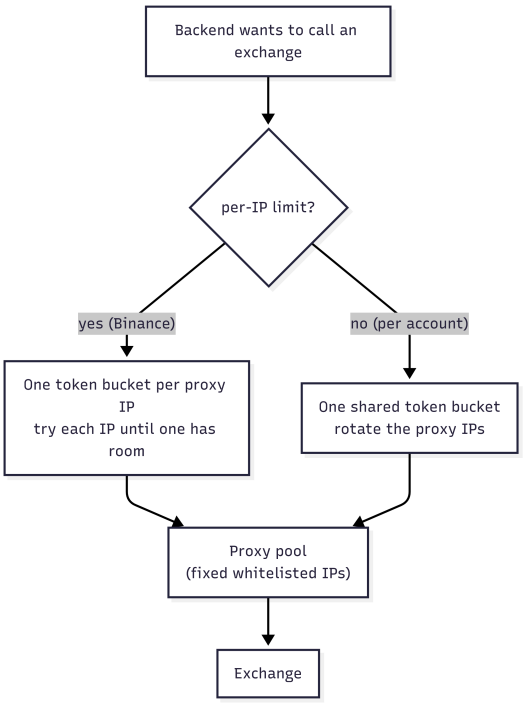


*Figure 4: Choosing the rate-limit strategy per exchange*

The buckets are kept per exchange and per limit type, because the exchanges count differently, and one choice here is worth saying plainly because it is easy to get wrong. Figure 4 shows it. Some exchanges count usage per IP address, others count it per account.

When the limit is per IP, the design keeps one bucket per proxy and tries each IP until one has room, which lets the pool carry more traffic. When the limit is per account, one shared bucket is correct, because spreading the same account's calls over many IPs would only push several lanes into the same single limit. A flag per exchange picks the right mode, and getting it wrong either wastes capacity or trips the limit.

## 3.6 From a new key to live analytics

The main flow of the system starts when a user adds an API key; Figure 5 follows it. Adding a key checks that it is read-only, encrypts it and stores it; anything that can trade or withdraw is rejected. Then the key-health worker takes over. On its next pass it confirms the key still works and starts the first sync: it backfills the available history over REST and opens a live WebSocket connection. Both paths feed the same trade ingester, which buffers the trades, writes them to the database, hands them to the positions service, and tells the analytics layer that something changed. New live trades also publish an event, which the frontend gets over the stream and uses to refresh only the affected views, without polling.

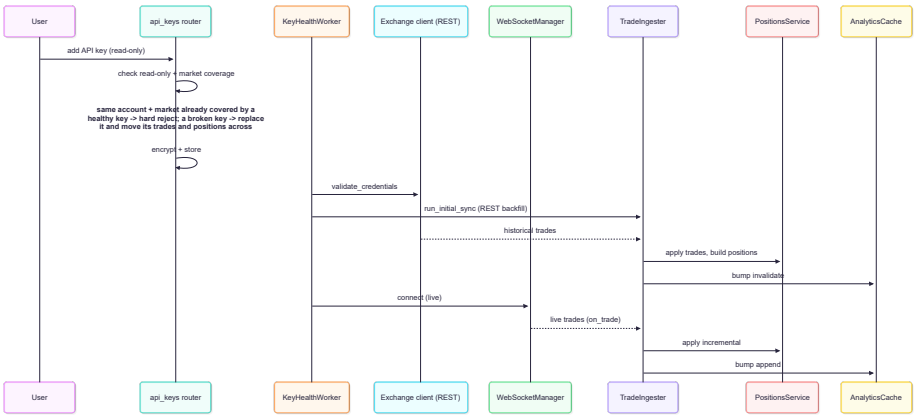


*Figure 5: From adding an API key to live analytics*

Adding a key is also where the account rules live. A user connects an account on an exchange, and a key covers one or both markets on it: spot, futures, or both. If a new key would cover a market that a working key already covers for that account, the system rejects it as a duplicate, so the same trades are never imported twice. A key that covers a market not yet connected for that account is simply added next to the others. The only time a key is replaced is when the key it would duplicate is broken: then the new, working key takes over, and the old key's trades and positions move across to it, so nothing is lost or doubled.

The same worker also heals the system. It rechecks keys on a schedule, takes a key offline when it stops working, and brings it back with a fresh sync once it recovers. The WebSocket manager reconnects on its own rolling schedule, so no connection is left open forever.

## 3.7 From trades to positions

A single trade is just one buy or sell. What a trader thinks in is a position: an entry, maybe a few adds, and an exit. The positions service builds positions from trades by matching them first-in, first-out inside each exchange, market and symbol, so a sell closes the oldest open buy and the result is counted correctly. Live trades are folded in as they arrive. When the history changes underneath - after a backfill or a fix - the service rebuilds the affected positions by replaying the trades from the start. The rebuild is deterministic: the same trades always give the same positions. Positions, not raw trades, are what the analytics, the journal and the calendar read.

A trade is never stored with a fixed link to its position. Instead, when the journal needs to show which position a trade belongs to, it finds it by matching the symbol and the time. This keeps trades and positions independent, which is what makes the rebuild safe: the service can throw away and rebuild every position without changing a single trade.

## 3.8 Market context: setups and excursions

A list of trades tells you what a trader did, but not what the market was doing at the time. To add that, the system pulls price candles for each symbol a user has traded. It only fetches candles for symbols that appear in the user's history, so it never downloads the whole market. Figure 6 shows these steps.

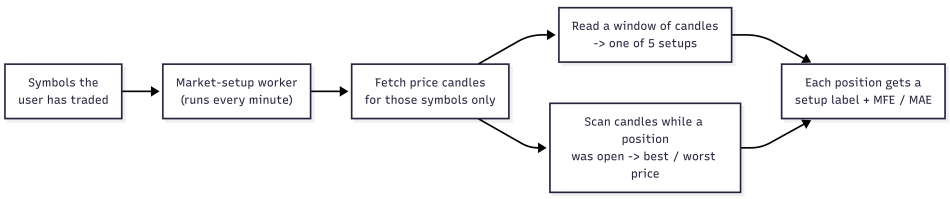


*Figure 6: Market context: from a user's symbols to a setup and excursion numbers on each position*

The candles are used in two ways. First, every position gets a setup - a short label for the kind of market it was opened into. A background worker reads a window of recent candles and sorts it into one of five simple types: a breakout, a pullback inside a trend, a fade inside a range, a reversal, or a burst of news-driven moves. It decides using basic measures like a moving average, the recent high-low range and the volume. The label is attached to the position afterwards, by matching the time the position opened to the nearest candle, so it does not slow down importing.

Second, the worker measures how far the price moved in the trader's favour and against them while a position was open - the best and the worst point it reached. These two numbers, often called the maximum favourable and maximum adverse excursion, show common mistakes, such as taking profit too early or holding a loser too long. Together, the setup and these two numbers turn a flat list of results into analytics a trader can use.

## 3.9 Analytics without wasted work

Analytics is the part a user looks at most, and redoing it on every change would be slow, so that is avoided. The computed analytics is never stored in the database. It lives only in

an in-memory cache, one per user and per active filter set, and it is built lazily, the first time it is asked for. Figure 7 shows how it works.

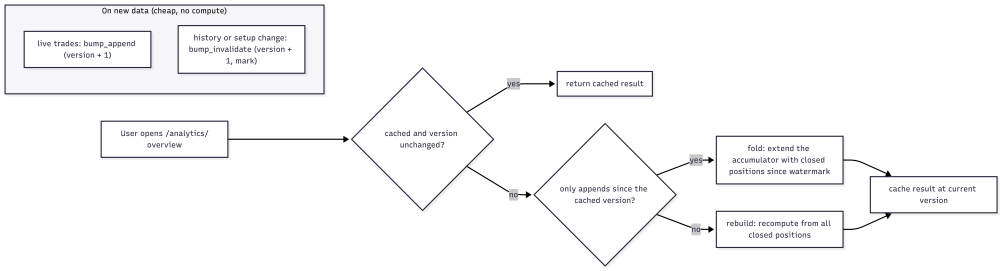


*Figure 7: The analytics cache: cheap version bumps on writes, a lazy fold or rebuild on read*

New data does not trigger a recompute. When live trades arrive, the ingester only adds one to the user's version counter. When the history is rewritten, or a position gets its setup, it adds one and also marks an invalidation. The real work happens only when the user opens the analytics. The service then compares the cached version with the current one. If nothing changed, it returns the cached result as is. If only live trades were added, it folds just the newly closed positions into the existing total, instead of scanning everything again. Only a real change to past data forces a full rebuild. Between page views the system does almost nothing, and a refresh usually costs a small fold, not a full pass over the user's trades.

## 3.10 Data model and indexing

The data model is small. Users own API keys, API keys bring in trades, trades are folded into positions, and positions are what the analytics reads. A few more tables hold market data: the list of symbols a user has traded, the price candles, and the setups found in them.

How the rows are indexed matters as much as the tables, because the same rows are read in very different ways. Table 3 lists the main indexes and why each one exists. Three are worth pointing out. The trade and position lists are read newest-first and filtered by user, so those indexes put the user first and the time last, which also matches how the time-series tables are split. One index is a partial unique index: it allows only one open position per user, key, exchange, market, symbol and side, which is the rule the matching depends on, while still allowing many closed positions. A second partial index covers only the positions that still need a setup, so the worker that fills them in scans a small index instead of the whole table.

| Table | Index | Why it exists |
|---|---|---|
| trades | `(user_id, time)` | List a user's trades, newest first |
| trades | `(user_id, symbol, time)` | A user's trades for one symbol |
| trades | `(user_id, trade_id, exchange, market, time)` unique | Block importing the same trade twice |
| positions | `(user, key, exchange, market, symbol, side)` unique, open only | Allow only one open position per instrument |
| positions | `(user_id, status)` | Split open and closed positions |
| positions | `(user_id, closed_at)` | Read closed positions in a date range |
| positions | `(exchange, market, symbol)` setup empty only | Find positions still needing a setup |
| api_keys | `(next_check_at)` | Find keys due for a health check |
| api_keys | `(api_key_hash)` | Block adding the same key twice |

*Table 3: The main database indexes and the access pattern each one serves*

Trades, candles and setups are time-series by nature, so they are stored as TimescaleDB hypertables. A hypertable is split by time underneath but stays a normal SQL table to query. Candles are the largest table and rarely change, so they are compressed after a week and dropped after a year. The schema and these rules are applied automatically when the backend starts.

## 3.11 Domain core and configuration

Under the slices sits a small shared core. It holds the data models, the fixed value sets (such as the four exchanges or the two market types) and one configuration object.

The data models are kept deliberately plain. The same classes are both the domain models and the database rows; there is no separate layer of copies in between. Only one small piece of translation exists, to turn a trade that comes from an exchange into a database row. Fewer layers means less code to keep in step.

The configuration is a single object, read once at startup. It holds the database connection, the proxy list, the plan limits, the exchange settings, and the keys for the AI and email services. Every part of the system reads its settings from this one place, so the whole system's behaviour can be checked in a single file instead of hunting through the code.

## 3.12 The AI insight

Tradeline can turn a user's analytics into a short, written review, using a large language model. This is one small feature, kept deliberately simple. When a user asks for an insight, the backend takes the analytics it has already computed - the headline numbers, the results by symbol, side, hour and weekday, the holding times and the excursions - and turns them into a prompt. It sends this, with a fixed instruction, to a hosted model (NVIDIA NIM, running Llama 3.3 70B) and asks for the answer in a fixed JSON shape: a one-word verdict, a headline, and three short lists - what is working, what is losing money, and what to do next. The frontend shows each of these as its own section.

Two choices keep this cheap and steady. The model is asked for low-creativity, structured output, so the answer is easy to parse and does not wander. And each user can ask for a new insight only once per cooldown - a week on the free plan, a day on the paid one - which controls the cost and stops the feature from being spammed. The last insight is stored, so opening the page is instant and does not call the model again.

## 3.13 Cross-cutting concerns

A few concerns run across every slice. Security comes first. API keys are read-only by rule and encrypted at rest, and the system rejects any key that could trade or withdraw. Sessions use short-lived tokens that carry a version number, so raising that number revokes every token at once; the logout and account-delete paths use it. The system also shows only the proxy addresses a user needs to whitelist, and nothing else about the servers.

Errors are handled in layers, so a failure deep inside an exchange call still reaches the user as something clear. Each exchange maps its own error responses, those are turned into one common set of exchange errors, and the API turns them into a single error shape the frontend can show. A Bybit rate-limit and a Binance one then look the same to the user. Rate limiting also runs on both sides: an inbound limiter caps how often one address can hit the sensitive routes, such as login, and the outbound per-exchange limiter from earlier protects the exchange calls. Email, for verification, goes out through one small sender interface, which uses Resend in production and just logs to the console in development. There are also clean seams for work that is planned but not built yet, most clearly a sign-in slice that a Google login can plug into without touching the rest.

## 3.14 Event bus and live updates

The backend pushes changes to the browser instead of letting it poll. Inside the process, a small event bus works as a simple publisher and subscriber: when a trade is stored or a sync finishes, the relevant part of the backend publishes an event, and every browser connection that belongs to that user receives it. Each connection has its own short queue, and if a burst of events fills it, the oldest is dropped instead of blocking, so a slow client can never hold up the rest.

The browser receives these events over a single long-lived stream, using Server-Sent Events. A trade event means new trades arrived; a sync event means a key finished its first import. The frontend does not refetch on every event. It collects the events for a short moment and then refreshes the affected views once. This pairing - a cheap publish on the backend and one combined refresh on the frontend - is what makes the interface feel live without flooding either side.

## 3.15 Frontend architecture

The frontend follows the same discipline as the backend: one single-page application, split into the same feature slices over a shared core. The shared core holds the interface kit, the common hooks, and a small set of design tokens - the colours, the spacing and so on - written once, so the light and dark themes stay consistent without styling each component by hand.

Server data is handled by a query library that caches each response and refetches it only when it goes stale, which keeps the components free of manual loading and caching code. Reads and writes follow a steady shape. A thin request helper adds the token and, when a token has expired, refreshes it once and retries. List and detail data are read through small hooks. Edits use optimistic updates: the change shows at once and is rolled back if the server refuses. Charts are the one heavy part, so they sit behind a thin wrapper and are loaded only on the pages that use them, which keeps the charting library out of the first load. That same wrapper is where the chart colours are read from the design tokens, so the charts follow the theme when it changes.

Reuse is built in rather than bolted on. The heavy pieces are written once and shaped by their inputs: one generic table component draws both the trades and the positions journals, one chart wrapper backs every chart, and one filter model - the set of exchanges, symbols, dates and so on - drives the journal, the analytics and the calendar together, so a filter chosen on one screen carries to the next. Smaller pieces are shared the same way, such as a single inline-edit hook used wherever a field can be renamed. A new screen is then mostly a composition of parts that already exist, not new code.

Live updates arrive over the stream from the backend, as described above: an event marks the matching cached data as stale, and the affected view refreshes on its own. Put together, the slices, the shared core, the query layer and the live stream let the interface stay simple on the surface while still showing new trades within a second of them arriving.

## 3.16 Deployment

The two applications are deployed separately, which matches how they are built; their layout is the one already shown in Figure 1. The single-page application is hosted as a static site behind a content network, and the backend runs on a small server behind a reverse proxy that handles TLS. A change to the frontend rebuilds and redeploys only the static site; a change to the backend rebuilds and restarts only the server, through CI pipelines that run the tests first and are filtered by which files changed. The backend runs as a single process on purpose: it holds the live WebSocket connections, the background workers and the in-memory analytics cache, none of which would survive being split across copies without extra coordination. An edge layer sits in front of the backend to hide its address and absorb abuse, and the egress proxies are the system’s other fixed point - the addresses the exchanges see and the user whitelists.

# 4 Implementation

This chapter shows how the design from Section 3 becomes working code. It follows the path the data takes, from the first call to an exchange, through the backend and into the browser, and ends with how the whole thing is tested and shipped.

## Contents



## 4.1 Method and conventions

A few rules run through the whole backend and explain the shape of the code that follows. Everything that touches the network or the database is asynchronous, from the routers down to the database driver, so one process can hold many live connections at once and still answer requests. Each feature is a slice in the same order: a router, then a service, then a repository. Every signature is typed, and small frozen dataclasses replace loose tuples, so the shape of the data is clear at each step. The code carries its meaning in names and small functions rather than comments, of which there are almost none; the snippets that follow are real code, trimmed to the lines that matter, with a cut marked by an ellipsis.

## 4.2 Talking to an exchange

The backend never names an exchange directly. It asks a gateway for the right client and works against one shared contract. The gateway builds one client and one request executor per exchange at startup and hands them out:

```
class ExchangeGateway:
    def __init__(self, session: aiohttp.ClientSession, router: ProxyRouter):
        self._executors = {exchange: RequestExecutor(exchange, session,
router) for exchange in _CLIENTS}
        self._public = {exchange: cls(self._executors[exchange]) for exchange,
cls in _CLIENTS.items()}

    def executor(self, exchange: Exchange) -> RequestExecutor:
        return self._executors[exchange]

    def client(self, exchange, api_key, api_secret, passphrase=None) ->
BaseExchangeClient:
        return _CLIENTS[exchange](self._executors[exchange],
DecryptedCredentials(api_key, api_secret, passphrase))
```

A client knows how to sign a request and which of the two history strategies to use, but it never opens a connection itself. Each of its calls funnels through one private method into the executor, and the executor is the single place where an HTTP call leaves the backend:

```
async def request(self, method, build_request, pool, weight=1):
    for attempt in range(self._MAX_RETRIES):
        proxy, bucket = await self._router.acquire(self._exchange, pool,
weight)
        req = build_request()
        async with self._session.request(method, req.url, headers=req.headers,
data=req.body, proxy=proxy) as r:
            if r.status in (418, 429):
                # ... penalise the bucket and retry ...
                continue
            await raise_for_status(self._exchange, r)
            bucket.observe(r.headers)
            return await r.json()
    # ...
```

One executor per exchange is shared by every client, so usage is counted in one place. Before each send it asks the router for a proxy address and a token bucket together, sends through that proxy, and reads the reply headers back into the bucket. Those two steps, the proxy and the bucket, are the next section.

## 4.3 Going out: proxies and rate limits

Every call out starts by asking the router for a proxy and a token bucket together. **`acquire`** chooses both, and it branches on the exchange's **`per_ip`** flag:

```
async def acquire(self, exchange, pool, weight=1.0):
    spec = self._spec(exchange, pool)
    slots = self._slots(exchange, pool)
    if not self._proxies:
        await slots[0].acquire(weight)
        return None, slots[0]

    start = self._cursor.get((exchange, pool), 0)
    self._cursor[(exchange, pool)] = start + 1

    if not spec.per_ip:
        await slots[0].acquire(weight)
        return self._proxies[start % len(self._proxies)], slots[0]

    for i in range(len(slots)):
        idx = (start + i) % len(slots)
        if await slots[idx].try_acquire(weight):
            return self._proxies[idx], slots[idx]
    # ... grow a new lane, or wait on the least-loaded address ...
```

When the limit is shared, one bucket guards the lane and only the proxy address rotates. When it is per address, there is one bucket per proxy, and the loop takes a token from the first proxy that has one free. The bucket is a token bucket, and its one trick is that it corrects itself from the exchange's own headers after each reply:

```
def observe(self, headers):
    spec = self.spec
    if spec.header_remaining_key:
        raw = headers.get(spec.header_remaining_key)
        if raw is not None:
            try:
                self.snap(int(raw)); return
            except (ValueError, TypeError):
                pass
    # ... otherwise read the used-quota header and snap to capacity minus
used ...
```

So the limiter follows the exchange's count instead of drifting from it. That covers the calls the backend makes. To receive trades the moment they happen, it also keeps a live connection open.

## 4.4 Staying connected

For live data the backend keeps one WebSocket connection open per key, and a single manager owns all of them. It is the one place that opens a connection when a key goes live, holds the running task and its handler, reconnects on a schedule, and shuts the connection down when the key is removed or fails, so nothing else in the backend ever touches a socket. To connect a key the manager decrypts the credentials, picks the handler for that exchange from a small registry, wires it with the callbacks it needs, and runs it as a task:

```
async def connect(self, api_key: ApiKey):
    async with self._lock_for(api_key.id):
        if api_key.last_error or api_key.id in self._handlers:
            return
        cls = _REGISTRY.get(api_key.exchange)
        if not cls:
            return

        creds = self._decrypt(api_key)
        handler = cls(
            creds=creds, http=self._ws_http,
            executor=self._gateway.executor(api_key.exchange),
            api_key_id=api_key.id, user_id=api_key.user_id,
capabilities=api_key.capabilities,
            on_trade=self._on_trade, on_status=self._on_status,
on_auth_fatal=self._on_auth_fatal
        )
        handler.attach_proxy_router(self._proxy_router)
        task = asyncio.create_task(handler.run())
        self._handlers[api_key.id] = handler
        self._tasks[api_key.id] = task
        self._started_at[api_key.id] = monotonic()
```

The important wire is **`on_trade`**. Every handler, whatever exchange it speaks to, calls that same callback with each trade it receives, so the rest of the backend never sees a raw socket. A key with a stored error is skipped here and left to the health worker, and an exchange that drops the connection ends the task, which a done-callback cleans up. The manager also reconnects each socket on a rolling eight-hour schedule, so no connection is left open indefinitely. The keys it skips here, and any whose connection fails, are the job of a second worker.

## 4.5 Keeping keys healthy

A live connection is not the whole story; keys expire, have their permissions changed, or hit a brief outage. A second worker watches for that. On a schedule it revalidates each key, and the three outcomes are all there is to it:

```
async def _check_one(self, key: ApiKey) -> None:
    client = client_for_key(self._gateway, key)
    try:
        caps = await client.validate_credentials()
        ensure_ip_restricted(caps, key.exchange)
    except (RateLimitError, ExchangeUnavailableError, *FETCH_ERRORS):
        # ...
        return
    except ExchangeError as e:
        async with self._sf() as session:
            await ApiKeyRepository(session).mark_checked(key.id, ...,
str(e.detail)[:512])
        if self._ws.is_live(key.id):
            await self._ws.disconnect(key.id)
        return
```

```
    recovering = key.last_error is not None
    # ...
    if not recovering and self._ws.is_live(key.id):
        return
    task = asyncio.create_task(run_initial_sync(self._ws, self._ingester,
self._cache, fresh, client, user.plan, ...))
```

A transient failure, a rate limit or a short outage, is left alone and tried again later, so a flaky moment never marks a good key as broken. A real error, a revoked or downgraded key, is written down and its socket is dropped. A key that works again, or one that is simply not live yet, is sent through the same initial sync the onboarding uses: it backfills what it missed and reopens the connection. So the system repairs itself between checks, with nobody watching it. That sync is also what fills the ingester, alongside the live feed from the handlers.

## 4.6 Ingesting trades

The ingester is the hub. The history backfill and the live handlers both feed it the same way, and it does not write each trade on its own: it buffers per key and flushes a batch on a short timer or when the buffer fills. The flush is where storage, positions, analytics and the live stream meet:

```
async def _flush(self, api_key_id: UUID) -> None:
    if api_key_id in self._backfilling:
        return
    async with self._buf_lock:
        entry = self._buffers.pop(api_key_id, None)
    if entry is None:
        return
    user_id, trades = entry
    if not trades:
        return

    async with self._sf() as session:
        inserted = await TradeRepository(session).upsert_many(user_id,
api_key_id, trades)

    if inserted:
        await self.apply_live(user_id, api_key_id, inserted)
        for trade in inserted:
            self._bus.publish(user_id, "trade", trade.to_payload())
```

**`upsert_many`** returns only the rows that were genuinely new, dropping duplicates on the table's unique trade constraint. So only new trades are folded into positions and analytics by **`apply_live`**, and only new trades become events. The buffer is held back while a key is still backfilling, so the first import does not race the live feed. Each new trade now becomes a position.

## 4.7 Reconstructing positions

A position is built from trades one at a time. **`apply`** looks for an open position on the other side to close, then opens or extends one on the trade's own side:

```
async def apply(self, trade: TradeEvent) -> list[CloseEvent]:
    own_side = PositionSide.LONG if trade.side == TradeSide.BUY else
PositionSide.SHORT
    opp_side = PositionSide.SHORT if trade.side == TradeSide.BUY else
PositionSide.LONG
    qty_left = trade.quantity
    fee = trade.fee or _D0
    events: list[CloseEvent] = []

    opposite = await self._repo.find_open(
        trade.user_id, trade.api_key_id, trade.exchange, trade.market_type,
trade.symbol, opp_side
    )
    if opposite is not None:
        consumed, pnl = self._reduce(opposite, trade, qty_left, fee)
        events.append(CloseEvent(opp_side, pnl, consumed, trade.time))
        qty_left -= consumed
        fee = _D0
    # ...

    if qty_left <= _D0:
        return events

    if self._can_open(own_side, trade.market_type):
        await self._open_or_extend(trade, own_side, qty_left, fee)
    return events
```

**`_reduce`** is where spot and futures part. A futures close trusts the realised number the exchange already gave; a spot close has none, so it matches first-in, first-out against the open entries the position keeps:

```
@staticmethod
def _fifo_consume(position: Position, qty_left: Decimal, exit_price: Decimal)
-> tuple[Decimal, Decimal]:
    fills = [[Decimal(p), Decimal(q)] for p, q in (position.entry_fills or
[])]
    consumed = _D0
    pnl = _D0

    while qty_left > 0 and fills:
        entry_price, entry_qty = fills[0]
        matched = min(qty_left, entry_qty)
        pnl += (exit_price - entry_price) * matched
        fills[0][1] -= matched
        qty_left -= matched
        consumed += matched
        if fills[0][1] <= _D0:
            fills.pop(0)
```

```
    position.entry_fills = [[str(p), str(q)] for p, q in fills]
    position.qty = position.qty - consumed
    return consumed, pnl
```

The oldest open entry is matched first and sets the profit on the quantity it covers. Because the rule reads only the trade and the position's own entries, replaying the same trades rebuilds the same positions, which is what makes a full rebuild from the trades safe. Positions, not trades, are what the analytics reads next.

## 4.8 Incremental analytics

Reading the analytics goes through one method. It returns the cached result untouched when nothing has changed, and otherwise does the smallest amount of work that is still correct:

```
async def get_overview(self, user_id: UUID, filters: AnalyticsFilters) ->
AnalyticsOut:
    cached = self._cache.get(user_id, filters)
    if cached is not None and cached.version == self._cache.version(user_id):
        return cached.out

    async with self._cache.lock(user_id):
        cached = self._cache.get(user_id, filters)
        version = self._cache.version(user_id)
        if cached is not None and cached.version == version:
            return cached.out

        repo = AnalyticsRepository(self.session)
        accumulator = await self._folded(repo, user_id, filters, cached)
        if accumulator is None:
            accumulator = await self._rebuilt(repo, user_id, filters)

        out = accumulator.result(await repo.open_positions(user_id, filters))
        self._cache.put(user_id, filters, CacheEntry(accumulator, out,
version))
        return out
```

The first check runs outside the lock, so an unchanged read costs almost nothing; the same check runs again inside the lock, so two requests that arrive together do not both rebuild. The real choice is fold against rebuild:

```
async def _folded(self, repo, user_id, filters, cached):
    if cached is None or cached.version <
self._cache.invalidate_version(user_id):
        return None
    accumulator = cached.accumulator
    watermark = accumulator.watermark
    if watermark is None:
        return None

    fresh = await repo.closed_positions_since(user_id, filters, watermark -
```

```
REPLAY_MARGIN)
    if not accumulator.extend(fresh):
        return None
    return accumulator
```

A full rebuild is taken only when a real invalidation has been marked since the cached version. Otherwise the accumulator is extended with just the positions that closed since its watermark, reaching back by a small **`REPLAY_MARGIN`** so a position that closed slightly out of order is not missed. A refresh after a few new trades reads a handful of rows, not the whole history. What the analytics shows is enriched by one more worker.

## 4.9 Market context

The market-setup worker adds context to positions, and it does two things with the candles it fetches. First, it labels each candle window with a setup. A small pure function reduces the window to a few signals and matches them against fixed rules, read from the loudest pattern to the quietest:

```
def _label(s: _Signals) -> str:
    if s.vol_ratio >= Decimal(3) and s.atr > _ZERO and s.bar_range >= s.atr *
Decimal(2):
        return "news_volatility"
    if s.vol_ratio >= Decimal("1.5") and (s.close_pos >= Decimal("0.9") or
s.close_pos <= Decimal("0.1")):
        return "breakout"
    if s.close_pos <= Decimal("0.3") and s.above_sma and s.bullish:
        return "reversal"
    if s.close_pos >= Decimal("0.7") and not s.above_sma and not s.bullish:
        return "reversal"
    if s.atr > _ZERO and (s.donchian_high - s.donchian_low) >= s.atr *
Decimal(4) \
            and Decimal("0.3") < s.close_pos < Decimal("0.7"):
        return "trend_pullback"
    return "range_fade"
```

The signals come from a twenty-bar window: a moving average, an average true range, the high-low channel, where the last close sits inside it, and how the last bar's volume compares to the window. The first rule that matches wins; a window that matches nothing is a **`range_fade`**. The function is deterministic and cheap, and the worker runs it in a separate thread so it never blocks the event loop.

Second, it measures how far price moved while a position was open. Given the candles between the open and the close, it takes the extreme high and low and turns them into two ratios off the entry price:

```
def _excursion(position: Position, klines: list[MarketKline]) ->
tuple[Optional[Decimal], Optional[Decimal]]:
    entry = position.entry_price
    if entry <= 0 or not klines:
        return None, None
```

```
    zero = Decimal(0)
    high = max(k.high for k in klines)
    low = min(k.low for k in klines)
    if position.side == PositionSide.LONG:
        return max(zero, (high - entry) / entry), max(zero, (entry - low) /
entry)
    return max(zero, (entry - low) / entry), max(zero, (high - entry) / entry)
```

For a long position the best point is the high above entry and the worst is the low below it; for a short the two swap. These are the maximum favourable and adverse excursions. Both jobs only ever touch symbols a user has traded, and both mark the affected analytics as stale. Any change a user can see is then pushed to their browser.

## 4.10 Pushing updates to the browser

The events from the flush reach the browser through a small in-memory bus. Each open connection is one queue under the user's id, and a publish drops the envelope into every queue for that user:

```
def publish(self, user_id: UUID, event_type: str, payload: dict) -> None:
    subs = self._subscribers.get(user_id)
    if not subs:
        return
    envelope = {"type": event_type, "data": payload}
    for q in subs:
        try:
            q.put_nowait(envelope)
        except asyncio.QueueFull:
            q.get_nowait()
            q.put_nowait(envelope)
    # ...
```

The queues are bounded. If one fills, because a client is slow or gone, the oldest event is dropped to make room rather than blocking the publisher; a slow consumer loses a little history but never holds up the trade being stored. The browser holds one Server-Sent Events stream open, reads these envelopes off its queue, and refreshes only the affected views. All of this data is time-series, and it lives in hypertables.

## 4.11 The data underneath

The time-series tables become hypertables when the backend starts, after the migrations have run. One step creates the extension and the hypertables; a second, on an autocommit connection because TimescaleDB needs it, attaches the compression and retention policies:

```
async def _ensure_timescale() -> None:
    async with engine.connect() as conn:
        await conn.execute(text("CREATE EXTENSION IF NOT EXISTS timescaledb
CASCADE"))
        await conn.execute(text(
```

```
            "SELECT create_hypertable('trades', 'time', if_not_exists => TRUE,
migrate_data => TRUE)"
        ))
        # ... market_klines and market_setups the same way ...
        await conn.commit()

    auto = await (await
engine.connect()).execution_options(isolation_level="AUTOCOMMIT")
    await auto.execute(text(_KLINES_COMPRESSION))
    await auto.execute(text(_KLINES_COMPRESSION_POLICY))
    await auto.execute(text(_KLINES_RETENTION_POLICY))
    # ...
```

**`create_hypertable`** is idempotent through **`if_not_exists`**, so the step is safe to run on every boot. The policies compress candles older than a week, segmented by symbol, and drop them after a year. Because this runs at startup, the database needs no manual setup, and a fresh deployment becomes a working time-series store on its own. One feature sits on top of all this.

## 4.12 The AI insight

The AI insight is one request, made on demand and guarded by a cooldown. The service checks the cooldown, reuses the analytics it already has, refuses if there is nothing to analyse, then calls the model and stores the result:

```
async def generate(self, user: User) -> AiInsightOut:
    if not self.config.nvidia_api_key:
        raise AiUnavailableError("AI insights are not configured")

    ready_at = self._ready_at(user)
    if ready_at is not None and datetime.now(timezone.utc) < ready_at:
        raise TooManyRequestsError(f"Your next AI insight unlocks
{ready_at.isoformat()}")

    analytics = await AnalyticsService(self.session,
self.analytics_cache).get_overview(user.id, AnalyticsFilters())
    if analytics.kpis.total_positions == 0:
        raise BadRequestError(...)

    raw = await complete(
        self.http, self.config.nvidia_base_url, self.config.nvidia_api_key,
self.config.nvidia_model,
        load_skill(), build_user_message(analytics)
    )
    body = _parse_insight(raw)
    await UserRepository(self.session).set_ai_insight(user,
body.model_dump_json(), datetime.now(timezone.utc))
    return self._out(user)
```

The cooldown comes from the plan: **`_ready_at`** adds the plan's cooldown hours to the time of the last insight, so the free and paid difference is just a number. The model gets

two messages, a fixed system instruction and the user's analytics rendered as a short block of text, and the call asks for a JSON object directly:

```
payload = {
    "model": model,
    "messages": [{"role": "system", "content": system}, {"role": "user",
"content": user}],
    "temperature": 0.15,
    "top_p": 0.7,
    "frequency_penalty": 0.3,
    "max_tokens": 700,
    "response_format": {"type": "json_object"},
    "stream": False
}
```

That system instruction is a file, **`skill.md`**, loaded once and kept next to the code. It is the model's whole brief: it sets the role of a trading-performance analyst, fixes the exact JSON schema the answer must use, and lists the rules that make the output worth reading - cite a real figure in every point, read cause and effect instead of repeating numbers, and no hedging, hype or filler. It also tells the model to treat the statistics as data only and to ignore anything inside them that looks like an instruction, which is the guard against prompt injection when user-derived text is sent to a model. Keeping the brief in its own file means its wording can be tuned without touching the code.

The low temperature and the **`json_object`** format keep the answer steady and easy to parse. Even so, the parser is defensive: it slices from the first brace to the last and validates the result against a typed model, so a little stray text around the JSON does not break the feature. Anything that cannot be reached or read becomes a plain **`AiUnavailableError`** instead of a stack trace.

## 4.13 Fetching and refreshing

The other half of the system runs in the browser. Its data is followed the same way: fetched, kept live, and shown, starting with fetching.

One typed helper makes every request, and the careful part is what it does on an expired token. Many requests can come back with a 401 at the same moment, so the refresh is shared: the first caller starts it, the rest wait on the same promise, and none of them refresh twice.

```
let refreshInFlight: Promise<string | null> | null = null

async function tryRefresh(): Promise<string | null> {
  if (refreshInFlight) return refreshInFlight
  const refreshToken = authStorage.getRefresh()
  if (!refreshToken) return null

  refreshInFlight = (async () => {
    try {
      const res = await fetch(`${env.apiBaseUrl}/auth/refresh`, { /* ... */ })
      if (!res.ok) return null
```

```
      const data = (await res.json()) as TokenOut
      authStorage.set(data.access_token, data.refresh_token)
      return data.access_token
    } catch {
      return null
    } finally {
      refreshInFlight = null
    }
  })()

  return refreshInFlight
}
```

Around it, a request runs once, and only when it comes back 401 does it refresh and run again:

```
const res = await exec(token)
if (res.status !== 401 || !auth || !token) return res
const fresh = await tryRefresh()
if (fresh) return exec(fresh)
authStorage.clear()
return res
```

If the refresh fails, the tokens are cleared and the app falls back to the login screen. A query library sits on top of this helper, caches each response, and refetches only when it goes stale. What marks it stale, most of the time, is the live stream.

## 4.14 Staying live

The browser keeps one stream open to the backend and turns it into cache invalidations. A small reader splits the stream into Server-Sent Events frames and parses each into an event. Rather than refetch on every event, the hook collects the affected query keys and flushes them once after a short pause:

```
const onEvent = (e: StreamEvent) => {
  if (e.type === "trade") schedule("trades", "positions", "analytics")
  else if (e.type === "sync") schedule("trades", "positions", "analytics",
"api-keys")
}

const schedule = (...keys: string[]) => {
  for (const k of keys) pending.add(k)
  if (flushTimer == null) flushTimer = setTimeout(flush, COALESCE_MS)
}
```

So a burst of trades during a sync becomes one refresh instead of one per trade. The stream also reconnects on its own, backing off a little further each time it fails:

```
const delay = attempts === 0
  ? RECONNECT_BASE_MS
  : Math.min(RECONNECT_BASE_MS * 2 ** (attempts - 1), RECONNECT_MAX_MS)
```

The whole thing is tied to whether a token is present, through a small hook built on React's external-store API, so the stream starts at login and stops at logout on its own. Before any of this is useful, the user has to connect a key.

## 4.15 Connecting a key

Connecting a key is a three-step dialog: pick the exchange, enter the credentials and choose the markets, then watch it validate. The dialog holds the answer to the last step in a discriminated union, so the screen can only ever be in one of three clear states:

```
type ValidationState =
  | { kind: "pending" }
  | { kind: "success"; result: ApiKeyValidationResult }
  | { kind: "error"; variant: ApiKeyErrorVariant; detail: string | null; code:
string | null }
```

When validation runs, the result of the request sets that state: a success carries what the backend detected, an error carries a mapped variant. The error mapping does the real work. A backend error code becomes a small typed object that carries both the message and what the screen should do with it:

```
export type ApiKeyErrorVariant = {
  title: string
  hint: string
  recoverable: boolean
  goBackTo: "credentials" | "plan" | "close"
}

export function mapApiKeyError(err: unknown): ApiKeyErrorVariant {
  if (!(err instanceof ApiError)) return fallback
  return variants[err.code] ?? fallback
}
```

So a wrong key sends the user back to the credentials step, a plan limit sends them to the plan, and a duplicate just closes; `recoverable` decides whether the retry button appears at all. The backend's error contract becomes a guided recovery instead of a raw message. Once a key is connected and trades flow, they are shown, and the charts are the interesting part.

## 4.16 Charts and theming

Every chart is a thin wrapper over the charting library, and none of them hardcode a colour. They read their colours from the same design tokens the rest of the app uses, at runtime:

```
function readColors(): ChartColors {
  const s = getComputedStyle(document.documentElement)
  const v = (token: string) => s.getPropertyValue(token).trim()
  return { accent: v("--accent"), profit: v("--profit"), loss: v("--loss"),
           grid: v("--line"), axis: v("--fg-3") /* ... pie palettes ... */ }
}

export function useChartColors(): ChartColors {
  const [colors, setColors] = useState<ChartColors>(readColors)
  useEffect(() => {
    const observer = new MutationObserver(() => setColors(readColors()))
    observer.observe(document.documentElement, { attributes: true,
attributeFilter: ["data-theme"] })
    return () => observer.disconnect()
  }, [])
  return colors
}
```

The tokens are defined once, in OKLCH, as a dark set and a light set. Because OKLCH keeps the hue and moves only the lightness, the light theme matches the dark one without a second hand-picked palette. A `MutationObserver` watches the `data-theme` attribute, so when the theme flips, every chart re-reads its colours and recolours without remounting. The flip itself is animated as one snapshot of the page, with a plain fallback when the browser cannot do it or the user asked for less motion:

```
const reduceMotion = window.matchMedia("(prefers-reduced-motion:
reduce)").matches
if (!reduceMotion && typeof document.startViewTransition === "function") {
  const transition = document.startViewTransition(apply)
  void transition.finished.then(release)
} else {
  apply()
  requestAnimationFrame(() => requestAnimationFrame(release))
}
```

The other heavy view is the journal, which is a filtered table.

## 4.17 Tables and filters

One table component draws both the trades and the positions journals. It rolls its own scrollbar, which is pure geometry: the thumb’s size and position are the viewport scaled against the content, and the header is kept in step with horizontal scroll by a transform rather than a re-render.

```
const sync = useCallback(() => {
  const vp = vpRef.current
  if (!vp) return
  if (headRef.current) headRef.current.style.transform = `translate3d(${-
vp.scrollLeft}px,0,0)`
  const { scrollTop, scrollHeight, clientHeight } = vp
  if (scrollHeight > clientHeight + 1) {
```

```
    const travel = clientHeight - 2 * PAD
    const size = Math.max(24, (clientHeight / scrollHeight) * travel)
    const offset = PAD + (scrollTop / (scrollHeight - clientHeight)) * (travel
- size)
    setV({ show: true, offset, size })
  } else {
    setV(HIDDEN)
  }
  // ... the horizontal track the same way ...
}, [])
```

What the table shows is decided by one filter model, shared by the journal, the analytics and the calendar, so a filter set on one screen carries to the next. A pair of pure functions turns that model into the query the backend expects:

```
export function toTradeFilters(f: JournalFilterState, apiKeyIds: string[]):
ListFilters {
  const params: ListFilters = {}
  if (f.exchanges.length) params.exchanges = f.exchanges
  if (apiKeyIds.length) params.api_key_ids = apiKeyIds
  if (f.market) params.market_types = [f.market]
  if (f.tickers.length) params.symbols = f.tickers
  if (f.period === "custom") {
    if (f.customFrom) params.date_from = f.customFrom
    if (f.customTo) params.date_to = f.customTo
  } else {
    const from = periodFrom(f.period)
    if (from) params.date_from = from
  }
  return params
}
```

**`toPositionFilters`** adds the side and status the positions list also takes. That is the whole client: fetch, stay live, connect, show. What ties the two halves together is that both are checked before they ship.

## 4.18 Testing

The two sides are checked in different ways, because they fail in different ways. The backend holds the parts where a wrong line is a wrong number, so those are unit-tested: the positions engine and its first-in, first-out accounting, the analytics fold, the rate-limit bucket, the setup classifier and the excursion maths, the event bus, and the key validation and IP policy. The suite is about twenty small modules, run with an async-aware test runner against fake repositories, so it needs no database. None of it talks to a live exchange; the exchange-specific parsing is tested against recorded responses instead.

The frontend has little logic of that kind, so it is held to a different bar. The type checker and the linter both have to pass with no errors and no warnings before a build is even produced, so a type error or an unused import fails the build. That catches the class of mistake that matters most in a large typed codebase. Whether the system meets its requirements is checked in Section 5. These checks are the first thing the pipeline runs.

## 4.19 Containers, proxy and delivery

The two applications ship on their own pipelines, split by which files changed, so a frontend change never rebuilds the backend. Each pipeline runs its checks first, the tests for the backend and the type check and linter for the frontend, and only a change on the main branch goes on to deploy.

The backend ships as a container, run as a single process behind a reverse proxy. The proxy is the one configuration file that matters here, because it carries a detail the live stream depends on:

```
{$API_DOMAIN::80} {
    reverse_proxy backend:8000 {
        flush_interval -1
    }
    encode gzip
}
```

**`flush_interval -1`** turns off response buffering, so the Server-Sent Events stream reaches the browser as each event is written rather than in batches, and the proxy handles TLS on its own. Deployment is deliberately plain: the pipeline connects to the server over SSH, pulls the new commit, and rebuilds the running containers in place:

```
git pull --ff-only
docker compose -f docker-compose.prod.yml up -d --build
```

There is no orchestration layer, because the backend runs as a single process by design; one server with a few containers behind the proxy is all it needs. The single-page app is built to static files and served separately from the edge, which is why a frontend release never touches the backend at all.

# 5 Validation

This chapter checks whether the finished Tradeline meets the requirements from Section 2. The system is checked in three ways.

## Contents



## 5.1 How the system was checked

The three are unit tests, the frontend's type and lint gates, and the system running live, and each fits a different part. The backend has a core of pure, deterministic logic, where a wrong line means a wrong number, and that core is unit-tested. The frontend is mostly presentation, so the compiler and the linter catch the mistakes that matter there. The integration is the part no unit test can stand in for: talking to four real exchanges over REST and WebSocket, staying inside their rate limits, and surviving their quirks. That is checked by running the deployed system against live accounts on Binance, Bybit, OKX and Bitget.

## 5.2 Testing the core logic

The unit-test suite is about twenty small modules. They cover the positions engine and its first-in, first-out accounting, the analytics fold, the rate-limit bucket, the setup classifier and the excursion maths, the event bus, and the key and IP validation. They run against fake repositories, so they need no database and finish in seconds, and they run on every change before anything is allowed to deploy.

One example shows their shape. The positions engine is fed a known sequence of trades, and the resulting position is asserted in full:

```
async def test_fifo_pnl_nets_across_fills_and_flags_averaging_down(self):
    positions = await _run([
        _trade(MarketType.SPOT, "BTC", TradeSide.BUY, "100", "1", 1),
        _trade(MarketType.SPOT, "BTC", TradeSide.BUY, "90", "1", 2),
        _trade(MarketType.SPOT, "BTC", TradeSide.SELL, "95", "2", 3)
    ])
    assert len(positions) == 1
    p = positions[0]
    assert p.status == PositionStatus.CLOSED
    assert p.realized_pnl == D(0)
```

```
    assert p.averaged_down is True
    assert p.entry_notional == D(190)
```

Two buys at 100 and 90, then one sell of the whole size at 95: first-in, first-out books a five-dollar loss on the first unit against a five-dollar gain on the second, so the realised profit is exactly zero, the position is flagged as averaged down, and the entry value is 190. Other tests in the same file cover partial closes, short positions, the futures path that trusts the exchange's own profit number, and the netting flip where one trade closes one side and opens another.

The frontend carries little logic of this kind, so it is held to a different bar. The type checker and the linter both have to pass with no errors and no warnings before a build is produced, which catches the class of mistake that matters most in a large typed codebase.

## 5.3 Meeting the requirements

Table 4 and Table 5 line up each requirement from Section 2 with how the system meets it and the evidence for it, the functional requirements first and the non-functional ones after.

| Functional requirement | Met by | Evidence |
|---|---|---|
| Connect to all four exchanges with read-only keys, spot and futures | Per-exchange clients behind one contract; read-only validation on add | Key and IP-policy tests; live accounts on all four |
| Import up to three months of history, then stay current from live feeds | REST backfill and a WebSocket per key, joined by the ingester | Ingester tests; the running deployment |
| Turn raw trades into positions and a full set of analytics | The FIFO and mark-to-market engine; the analytics accumulator | Engine and analytics tests |
| A dashboard, a journal and a calendar, all filterable | The React frontend over one analytics endpoint | The type and lint gates; the running interface |
| Accounts and two plans, free tier with the full analytics | Auth with revocable tokens; the plan model | Auth tests; the plan catalogue |

*Table 4: The functional requirements, how each is met, and the evidence for it*

| Non-functional requirement | Met by | Evidence |
|---|---|---|
| Never hold the right to move funds | Read-only keys, encrypted at rest; trade or withdrawal keys rejected | Validation, IP-policy and security tests |
| Affordable: a complete free tier, paid under nine dollars a month | The plan limits and the $8.99 price | The plan model |
| Stay inside each exchange's rate limits under concurrent sync | The token bucket and per-address or per-account sharding | Rate-limiter tests; many keys run without limit errors |
| A clean, responsive interface usable without a manual | Design tokens over a shared interface kit | The running interface |
| A clean architecture, deployed as a reachable service | Feature slices; path-filtered CI and the deployment | Green CI; the live service |

*Table 5: The non-functional requirements, how each is met, and the evidence for it*

Each functional requirement is met by the part built for it and confirmed by the tests and the running system. The non-functional ones matter more here. Funds stay safe by rule: keys must be read-only and are encrypted at rest. The plans are affordable, with full analytics free and the paid tier under nine dollars a month, though paying for it is still future work. And the rate limits are what running proves best, since many keys have synced across the four exchanges without tripping them.

## 5.4 The system in use

The clearest validation is that the whole path works from end to end on live data. A user adds a read-only key; in the first sync the backend backfills the available history and the live connection takes over; the trades become positions; the dashboard, the journal and the calendar fill in; and an AI insight can be generated from the result. Each of these parts was tested or reasoned about on its own, and running them together against real accounts is what shows they hold up as a system rather than as separate pieces.

## 5.5 Limitations

A few limits are worth stating plainly. Usability comes from real use of the interface, not a formal user study, so there are no measured task-success numbers. There is no load testing, so behaviour under many users at once is reasoned from the design, not measured. The tests cover the core logic, not every branch of every feature. Billing is half-built: the plans and the free tier work, but the paid tier cannot yet be bought. And each exchange integration is written against today's API, which the exchange can change. None of these take away from the result; they just mark where it rests on tested evidence and where on the design.

# 6 Conclusion

The goal set out in Section 1 was a journal for an intermediate crypto trader: affordable, automatic, and built around their own exchanges instead of a spreadsheet. That system was built, and it runs in production. The four objectives are met: the exchanges are integrated, trades are rebuilt into positions and a full analytics layer, the interface ships with a free tier that stands on its own, and the whole system is deployed as a live service rather than a prototype.

The hardest part of the work sat in the gaps between the features, not in the features themselves. Connecting four exchanges that agree on almost nothing, staying inside rate limits that none of them count the same way, and keeping a live feed and months of history in one consistent picture were each harder than any screen the user ends up looking at. None of it was deep in theory, but all of it had to be right at the same time, and most of the mistakes only surfaced under live data. That is where the real engineering of this project went.

Some work remains, and it falls into three groups. The first is to make the paid tier real: a hosted checkout and the billing behind it, with subscriptions, renewals and failed-payment handling driven by the payment provider's webhooks, so that changing a plan, which already works, becomes a real purchase. The second is to deepen the analytics. The system already collects price candles and measures the best and worst point each position reached, and the next step is to use that price path directly - a continuous, candle-based equity curve that moves with the market while a position is open instead of stepping only at the close, a per-position price overlay, and richer measures such as funding costs, fee breakdowns and risk-adjusted returns. The third is a set of smaller items that round the product off: signing in with Google, sending verification email to any address rather than a single test recipient, differentiating history limits by plan and exchange, and adding logging and metrics for running it at scale.

For the trader it was built for, the result is already worth having: connect a single read-only key, and within the first sync the trading they could only half-remember comes back as clear, honest numbers, free and across every exchange they use. The point was never to add one more expensive dashboard, but to let an ordinary crypto trader stop guessing at their own results and start reading them - and that part works today.

# Glossary

## Engineering

***Hypertable*** – **TimescaleDB hypertable**: A table partitioned by time underneath while still queried as one ordinary SQL table.

***OKLCH*** – **OKLCH colour space**: A perceptually uniform colour space of lightness, chroma and hue, used for the interface tokens.

***Token bucket*** – **Token bucket**: A rate-limiting structure of tokens that refill over time; each call spends one, and an empty bucket waits.

## Trading

***FIFO*** – **First-In, First-Out**: Settling a closing trade against the oldest open entry first, used to attribute profit on spot positions.

***MAE*** – **Maximum Adverse Excursion**: The furthest the price moved against the trader while a position was open.

***MFE*** – **Maximum Favourable Excursion**: The furthest the price moved in the trader's favour while a position was open.

***Perp*** – **Perpetual futures**: A derivative that tracks an asset's price with no expiry, the main instrument crypto traders use, often with leverage.

***Position*** – **Trading position**: One or more trades in a single symbol on one side, long or short, from the first entry to the exit.

***Read-only key*** – **Read-only API key**: An exchange key that can read account data but cannot place trades or move funds.

***Spot*** – **Spot trading**: Buying or selling the asset itself for immediate settlement, without leverage or an expiry.

## Web and protocols

***JWT*** – **JSON Web Token**: A signed token that carries a user's session; a version number on it can revoke every token at once.

***REST*** – **Representational State Transfer**: The request-and-response HTTP style used to read history and submit actions.

***SSE*** – **Server-Sent Events**: A one-way stream from the server to the browser, used to push live updates without polling.

***WebSocket*** – **WebSocket connection**: A persistent two-way connection, used to receive trades the moment they happen.

# Bibliography

[1] Triple-A, "Global Crypto Ownership Reaches 562 Million People in 2024." Accessed: June 03, 2026. [Online]. Available: https://www.triple-a.io/cryptocurrency-ownership-data

[2] NFTevening, "Study: 84% of Retail Crypto Traders Lose Money in Their First Year." Accessed: June 03, 2026. [Online]. Available: https://nftevening.com/84-percent-of-retail-crypto-traders-lose-money-in-their-first-year/

[3] Coin Market Manager, "Pricing." Accessed: June 03, 2026. [Online]. Available: https://coinmarketman.com/pricing/

[4] Edgewonk, "Pricing." Accessed: June 03, 2026. [Online]. Available: https://edgewonk.com/pricing

[5] Coinmonks, "Crypto Trading Journal Review: CoinMarketMan vs TradersMake.Money." Accessed: June 03, 2026. [Online]. Available: https://medium.com/coinmonks/crypto-trading-journal-review-coinmarketman-com-vs-tradersmake-money-76d83d32c3e

[6] Bullish Bears, "Edgewonk Review 2026: Pricing, Pros, Cons and Alternatives." Accessed: June 03, 2026. [Online]. Available: https://bullishbears.com/edgewonk-review/

[7] Trader Make Money, "Pricing and Plans." Accessed: June 03, 2026. [Online]. Available: https://tradermake.money/prices

[8] TraderSync, "Pricing." Accessed: June 03, 2026. [Online]. Available: https://tradersync.com/pricing/