QUANT-X

An End-to-End Architecture Guide

How to design and build a multi-agent AI decision dashboard — from a single scoring function to a live, watchable UI.

This guide walks through the architecture behind QUANT-X, a browser-based trading dashboard that turns a handful of market inputs into a narrated, multi-agent decision. It's written for builders who want to understand — and reuse — the underlying pattern, not copy a specific implementation. Expect diagrams, design principles, and a build roadmap. Expect very little code.

A companion guide for builders · Architecture & design, not a copy-paste tutorial

Contents

SECTION 01

Introduction

This guide is not a tutorial you copy-paste from. It's a walkthrough of the architecture and design decisions behind QUANT-X — a multi-agent AI trading dashboard built as a live, browser-based demo. If you want to build something like it yourself, in trading or in any other domain where several signals need to add up to one confident decision, this is the map.

QUANT-X lets a user describe a market scenario with a handful of sliders, then watches four differently-named “AI agents” reason through it one at a time before a decision engine fires a BUY, SELL, or HOLD call with a confidence score, followed by a simulated trade execution.

It's built entirely with Next.js, React, and TypeScript, and it runs completely in the browser — there is no AI backend, no model API, no server-side inference. That's on purpose, and it's the first lesson in this guide: an interface can feel intelligent without literally calling an LLM four times per decision.

One note before we start: QUANT-X is a demo built for a screen recording, not a trading product. Nothing in this guide is financial advice — it's a walkthrough of software architecture.

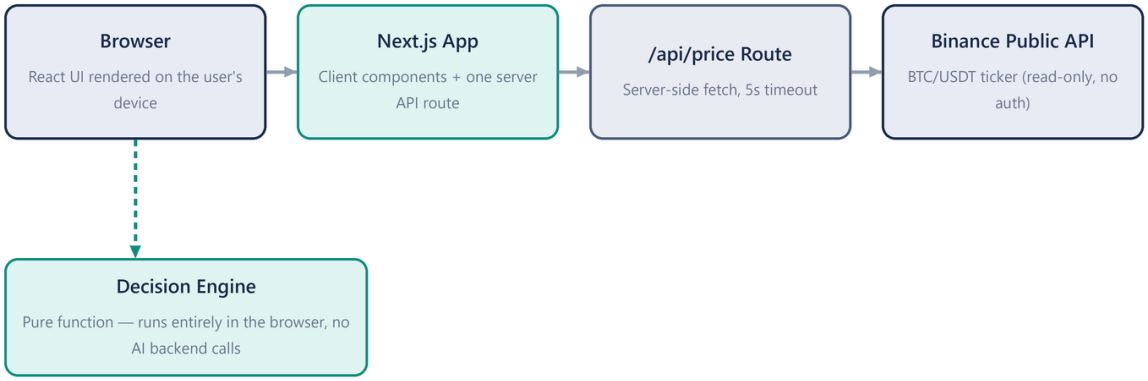
SECTION 02

The Big Picture

At the system level, QUANT-X is almost boring — which is the point. There is exactly one thing it talks to outside itself: a public price ticker. Everything that looks like “AI reasoning” happens entirely on the user's device.

DIAGRAM 1

High-Level System Architecture



Live price flows left → right on a 4-second poll. Everything else stays on-device.

The API route in the middle exists for one reason: browsers block direct cross-origin calls to the price API from client code, and a tiny server-side proxy sidesteps that cleanly, with its own timeout so a slow upstream response can never hang the UI. It carries no business logic — it fetches a price and returns it.

The decision engine is drawn as its own branch off the browser box specifically to make a point: it is not a step in the network chain. It's a function call, sitting in memory, that happens to run in a browser tab instead of a server.

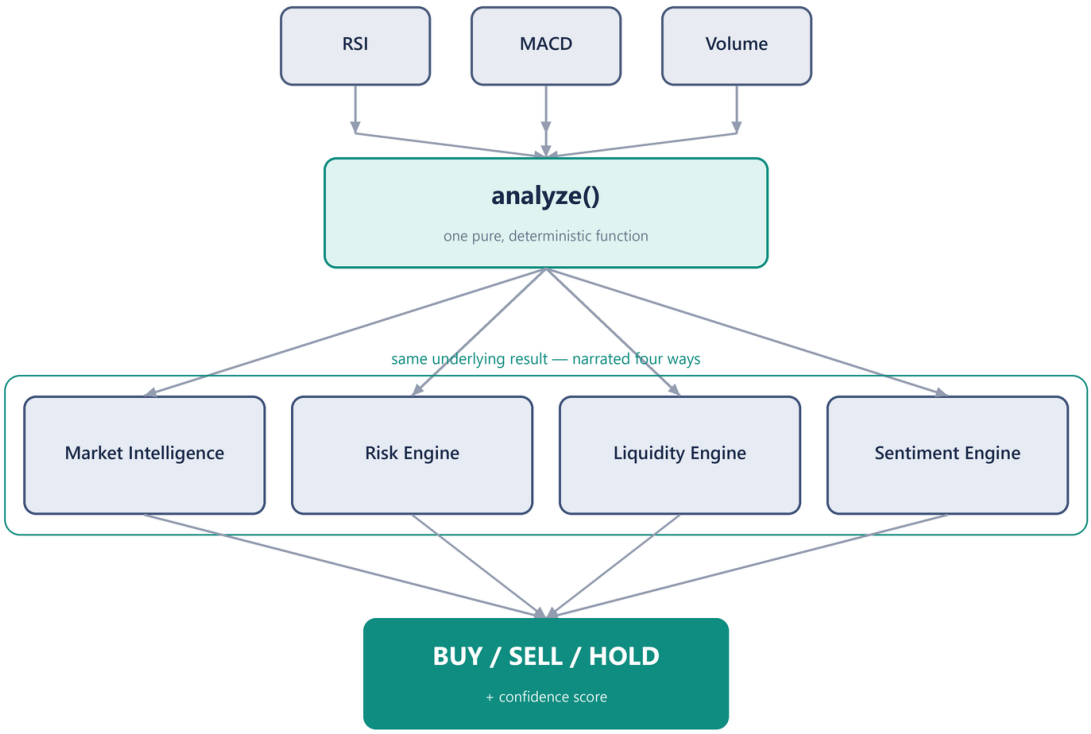
SECTION 03

The Core Pattern: One Brain, Four Voices

This is the idea worth taking with you, independent of trading, dashboards, or any of the specifics of this build.

DIAGRAM 2

One Brain, Four Voices



Underneath the four “agent” cards is a single deterministic function. It runs once, produces one result, and that result contains four labeled slices — each slice is a different narrated view of numbers that were already computed together, in the same pass, by the same logic.

Why bother presenting one result as four? Three reasons, and they generalize well beyond trading:

- **Explainability.** A single opaque score reads as a black box. Four named specialists, each reporting on one slice of the picture, reads as reasoning — even when the computation underneath is identical either way.
- **Pacing.** Four labeled slices give a UI natural checkpoints to reveal information gradually, instead of dumping a wall of text the moment a result exists.
- **Extensibility.** Because each “agent” is just a labeled entry in an array, adding a fifth voice later means adding one more entry — not building a new subsystem.

Nothing about this pattern requires the underlying logic to stay rule-based, either. Swapping any one “agent” for a genuine LLM call — with its own prompt and its own model — is a natural next step, and the pattern doesn't care which agents are rules and which are model calls, as long as they all still resolve into one shared decision.

SECTION 04

Layered Architecture

Four layers, each with exactly one job, each depending only on the layer below it.

DIAGRAM 3

Layered Architecture

Domain logic has zero dependency on React — swap the UI, keep the brain.

The layer worth designing carefully first is **Domain Logic**. In this build it's two functions that take plain data in and return plain data out — no React, no DOM, no timers, no knowledge that a UI exists at all. That constraint is what makes it trivially testable (call it with inputs, check the output) and trivially portable: the same two functions could power a CLI tool, a Slack bot, or a completely different frontend without a rewrite.

Orchestration is the layer that's allowed to know about time — it owns the hooks that poll for live data and the state machine that sequences a run. **Presentation** is only allowed to know about “what does the current state look like,” never “what should happen next.” Keeping that boundary strict is what keeps the component tree simple even as the UI gets more animated.

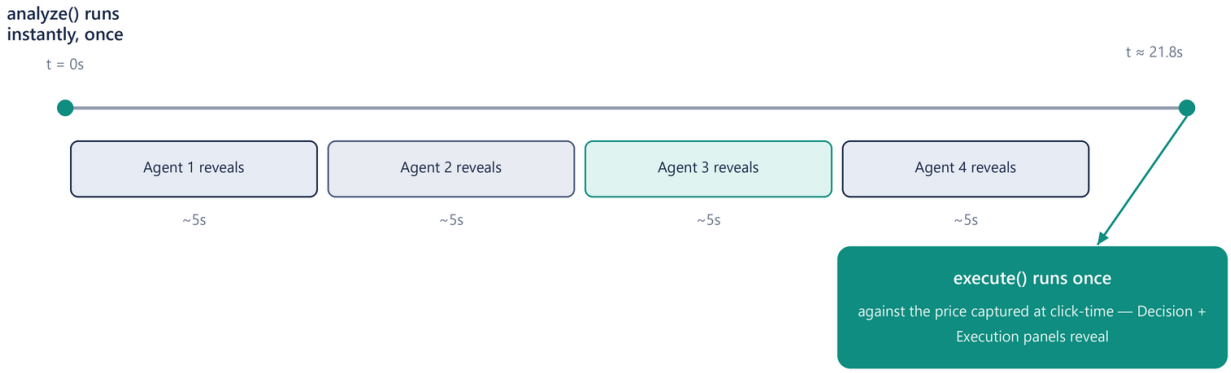
SECTION 05

Data Flow: One Full Run

Here's what actually happens, in order, from the moment a user clicks the button:

DIAGRAM 4

One Full Run — Timeline



The UI reveal is timer-driven; the underlying decision was already final at t = 0.

The detail worth internalizing: the decision is **fully computed at t = 0**. Every second after that is choreography — a timer revealing information the system already knows, at a pace a human can actually follow. The trade itself only executes once, right at the end, against the price that was current the moment the user clicked (not re-fetched mid-run, which would make the outcome depend on unrelated network timing).

This separation — *when something becomes true* versus *when the user is shown it's true* — is a reusable technique any time you're building an interface meant to be watched, not just used.

SECTION 06

Designing a Trustworthy Decision Engine

Three principles, in order of how expensive they are to skip:

- **One source of truth.** A single function owns the decision. Nothing downstream recomputes it, second-guesses it, or has its own opinion.
- **Derive, don't ask.** If a value can be computed from other inputs, don't make the user set it separately — every extra manually-set knob is another way for the inputs to quietly contradict each other.
- **No modifier may reverse a decision, only scale it.** A secondary factor is allowed to make the engine act on a signal more or less strongly — it should never be able to flip which decision comes out.

That third one is a scar, not a guess. An earlier version of this exact engine let two different readings of the same underlying number pull the final score in opposite directions. For a real slice of the input space, that produced a decision that contradicted the label sitting right next to it on screen — confusing at best, and the kind of inconsistency that quietly destroys trust in anything presenting itself as “AI reasoning.” The fix wasn't a UI patch. It was the rule above: a modifier scales conviction, it never flips direction.

The insurance policy is cheap: write a small script that sweeps a wide grid of plausible inputs through your scoring function and asserts basic self-consistency, before you ship it and definitely before you put it in front of a camera.

```
for every combination of (signal_a, signal_b, signal_c) in a realistic range:

    result = decision_engine(signal_a, signal_b, signal_c)

    assert result.decision agrees with result.explanation

    assert result.confidence is within its documented bounds
```

That's pseudocode, deliberately — the exact scoring rules aren't the point of this guide (or particularly interesting on their own). The habit of sweeping your own logic for self-contradictions before you trust it is the transferable part.

SECTION 07

Designing for Watchability

The UX side of the same coin. A few choices that mattered more than they might look:

- **Reveal sequentially, not instantly.** Even though the result exists immediately, showing it all at once reads as a wall of text nobody actually reads. Revealing it in labeled stages reads as reasoning.
- **Let each step's status reflect its own real progress** — not just an outer sequencer. If a card's “done” state is driven purely by a shared timer, it can claim to be finished while its own content is still visibly animating. Small inconsistency, but it quietly undercuts trust the same way the decision-engine bug in Section 6 did.
- **Use connectors as a progress metaphor.** A simple line or arrow between steps, lit up as the process passes through it, does a lot of work to make a multi-stage process feel like a pipeline instead of four unrelated widgets.
- **Pace deliberately.** Character-by-character reveal, a few hundred milliseconds of pause between beats — slower than feels necessary while you're building it, and usually correct once you watch someone unfamiliar see it for the first time.

The underlying reason to care: a product that's also going to be screen-recorded benefits from being built to be watched, not merely to be used. Those aren't always the same design goal, and it's worth deciding on purpose which one you're optimizing for at each step.

SECTION 08

Tech Stack & Why

Choice	Why
Next.js (App Router)	One project serves the UI and the one small server route it needs — no separate backend to stand up or deploy.
React 19 + TypeScript	A component model for the UI, and types that catch “which state are we actually in” bugs before they ship.
Tailwind CSS v4	Utility styling keeps markup and styling co-located, which matters for a dense, animation-heavy interface iterated on quickly.
Framer Motion	Declarative animation driven by React state, so the UI stays in sync with the state machine instead of hand-rolled timers scattered across components.

No charting library, no state-management library, no backend framework — four runtime dependencies, total. Worth noticing: the interesting engineering in this project is almost entirely in plain functions and state design, not in tooling.

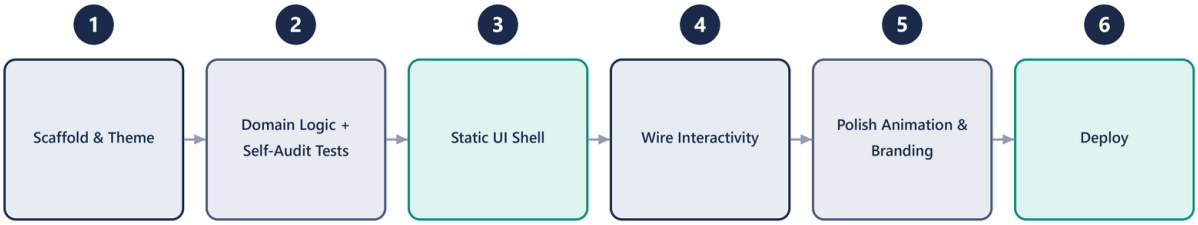
SECTION 09

Build Roadmap

Six phases, in an order that keeps the expensive-to-change decisions early:

DIAGRAM 5

Build Roadmap



1. Scaffold & Theme — Stand the project up and establish the visual language — palette, type, spacing — before any real component exists, so everything after slots into a system instead of being designed alone.

2. Domain Logic + Self-Audit Tests — Build the decision engine first, as pure functions, before any UI exists. Write the sweep-test from Section 6 before you trust it.

3. Static UI Shell — Build every screen against fake, fixed data — no interactivity yet. Layout and hierarchy are cheapest to change before real state is wired in.

4. Wire Interactivity — Connect real state, real timers, real sequencing. This is where the orchestration layer — the hooks — gets built.

5. Polish Animation & Branding — The pass that turns “functional” into “watchable” — pacing, motion, the small details from Section 7.

6. Deploy — Ship it.

SECTION 10

Deploying & Extending

This class of app — a Next.js frontend with one lightweight server route — deploys cleanly to Vercel or any Node-capable host, with essentially no configuration. If you add real external API keys later, that's the one place secrets belong: server-side environment variables, never shipped to the client bundle.

Credible next steps, roughly in order of effort:

- Swap one or more “agents” for a genuine LLM call, behind the same shared result shape the rest of the UI already consumes.
- Add more instruments or indicators — the pattern doesn't care how many inputs feed the decision.
- Persist run history, so a user (or you) can see a track record instead of one run at a time.
- Add a backtesting mode that replays historical data through the same engine — free, since the engine never depended on live data to begin with.

One practical guardrail worth planning for early: if the free public API you're polling has a rate limit, a client that polls every few seconds per visitor will hit it far sooner than a single developer testing locally ever notices. Caching the latest price for a few seconds at the API-route layer (rather than each client refetching upstream) costs almost nothing to add and is the difference between a demo that works once and one that survives being shared.

SECTION 11

Closing

None of this requires exotic tools — a scoring function, a sequencing state machine, and a UI willing to take its time revealing what it already knows. If you build your own version of this, the architecture in this guide is the part worth keeping. Everything else — the domain, the inputs, the exact visual language — is yours to make your own.

If you take only three ideas from this guide, make them these: one function can honestly present itself as several specialists, as long as it never contradicts itself doing so; separate the moment a decision becomes true from the moment your UI shows it's true; and before you trust any rule-based system enough to put it on camera, sweep it across a wide range of inputs and check it never argues with its own explanation. Everything else in this build — the palette, the exact indicators, the trading framing — is just one way of dressing up those three ideas.

Further reading

- Next.js documentation — nextjs.org/docs
- React documentation — react.dev
- TypeScript handbook — typescriptlang.org/docs
- Framer Motion documentation — motion.dev
- Tailwind CSS documentation — tailwindcss.com/docs

Thanks for reading this far. Now go build the version that's actually yours.