A Technology Vision for Quantitative Trading

Thomas Schmelzer, Jebel Quant Research — May 2026 — thomas@jqr.ae

Jebel Quant Research builds the tools and infrastructure for systematic trading teams, from data access and portfolio construction to live execution and repo management. This document sets out the thinking behind that work.

These are my personal views on how a quantitative trading platform should be built. I have spent two decades working across systematic hedge funds, high-frequency trading, family offices and sovereign wealth funds. Before any of that, I trained as a professional mechanic at AUDI and spent a summer assisting in their quality labs. The practices I describe in the early sections were perfectly reasonable at the time. Technology has moved on, and I think the way we build and run these platforms should move on too. When I reach for analogies about how precision work gets done (kitchens, assembly lines, shared standards) I am drawing on direct experience, not metaphor.

The Old Approach

For most of the industry's history, quantitative research followed a familiar pattern. A small group of researchers, typically mathematicians, physicists and statisticians, would develop trading strategies in MATLAB or Python. These environments suited the work well: easy to iterate on, rich in numerical libraries and close to the mathematical notation of the models. A researcher could move from an idea to a backtest in a matter of days.

The outputs were scripts: dense, clever and personal. Variable names made sense to their authors. Logic accumulated in layers over months or years. The code worked, and it carried real insight about markets.

Reinventing the Wheel

Because research lived in personal scripts, knowledge did not accumulate. Each researcher wrote their own version of the same basic components: moving averages, momentum signals, mean-reversion filters, portfolio construction routines. There was no shared library, no common vocabulary in code. Two researchers on the same team might implement the same idea with different conventions, different edge case handling, different numerical choices. When results diverged, it was hard to know why.

This also made it difficult to build on what had come before. Onboarding a new researcher often meant starting from scratch. Institutional knowledge lived in people rather than in code, and walked

out the door when those people moved on.

There is a subtler problem too. Researchers working in isolation tend to gravitate towards strategies they can implement themselves. The limit is often their programming skills rather than their research ideas. A researcher with strong mathematical intuition but modest software engineering experience will keep returning to the same simple constructions, not because they are the best ideas available but because they are the ones within reach. A shared platform with well-built common tools raises that ceiling. The researcher's ambition is no longer bounded by what they can personally code from scratch.

The Handover Problem

When a strategy was ready for production, it was passed to a team of software engineers to reimplement in C++. The rationale made sense: C++ offered the performance, determinism and operational robustness that live trading required. But the process was expensive.

The handover was rarely clean. A MATLAB matrix operation that fit on one line might require careful memory management and numerical precision decisions in C++. Edge cases the script handled implicitly had to be made explicit. Bugs crept in during reimplementation and were hard to catch because the reference and production implementations diverged the moment the handover began.

The result was a slow pipeline. Research cycles were constrained by engineering capacity. Months could pass between an idea and a live strategy. Every change to a live strategy risked restarting the process.

Modern machine learning made this untenable. Libraries like PyTorch represent millions of engineering hours: automatic differentiation, GPU kernels, distributed training, a vast ecosystem of pretrained models. Reimplementing any meaningful fraction of that in C++ is not a project; it is a decade-long programme. Teams that held to the C++ mandate found themselves unable to use these tools, and fell behind those that did.

History has not been kind to the handover model. Across the industry, the pattern repeated itself: the central tools were never built. Teams focused on strategies and left the shared infrastructure as an afterthought. Essential tools (data access, portfolio construction, performance analytics, backtesting frameworks) were created independently by every team that needed them, in slightly different ways, with slightly different assumptions. The same wheel was reinvented dozens of times across the same organisation, and nobody had a complete picture of what existed or how reliable

any of it was. The handover model produced not one coherent platform but a sprawling collection of overlapping partial solutions, each owned by whoever happened to write it.

There is a human cost to the handover model that rarely gets discussed. When something goes wrong in a live strategy, the handover creates a ready-made alibi for everyone involved. The researcher points to the Python code and says it was correct. The engineer points to the C++ implementation and says it faithfully reproduced what was handed over. The operations team says they deployed exactly what they were given. Nobody is lying, and nobody fixes anything quickly. In a system built on a clean handover between separate groups, accountability diffuses precisely at the moment when it is most needed. The checkerboard structure is partly a response to this. When researchers and developers have built something together, they own it together.

The handover model can be made to work better. Shared interface definitions between research and engineering reduce ambiguity at the boundary. Strict versioning of the research artefact (the exact notebook, the exact data snapshot, the exact parameters) gives the engineering team something precise to reimplement rather than a moving target. Automated regression tests that compare research and production outputs on the same inputs catch divergence early, before it reaches live capital. These are real improvements and worth making. But they are patches. They reduce the cost of the translation step; they do not eliminate it. The fundamental problem is the translation step itself: the moment where one representation of an idea becomes another, and where something is always lost or changed in the crossing.

A New Direction

Our idea is built around a single environment for both research and production. Researchers express ideas in high-level terms and the platform carries them through to execution without a translation step.[1]

The old handover model borrowed, implicitly, from an outdated idea of the factory: one group does its work and passes the output down the line. It is worth being precise about which factory. Henry Ford's assembly line was built on fixed interfaces, specialised stations and inspection at the end. A modern car factory looks nothing like this. Toyota's production system (quality checks at

[1] The idea of a unified research-to-production environment is not new, and it would be remiss not to acknowledge the prior art. Its origins trace to SecDB, the integrated trading-and-risk system built at Goldman Sachs from the 1990s, and to its descendants Athena (JP Morgan) and Quartz (Bank of America Merrill Lynch). The most prominent platforms bringing this approach to the wider market are Beacon (founded in 2014 by SecDB and Athena alumni, Python-native, acquired by Clearwater Analytics in 2025) and Deltix (now part of EPAM Systems, which acquired it in 2020), each offering data, research, backtesting and execution within a single environment, with the explicit aim of carrying a strategy from idea to live trading without reimplementation. Where we differ is less in that goal, which we share, than in the means and the framing, discussed in the Build or Buy section below.

every station, workers empowered to stop the line, continuous feedback between stages) is far closer to what we advocate here than to anything Ford would recognise. The quant industry adopted a model that manufacturing itself had largely abandoned by the 1980s. Marcos Lopez de Prado has argued explicitly for the factory model in his work on the industrialisation of quantitative finance[2] — and the model was put into practice at ADIA's Team Q, where I had the opportunity to observe its strengths and its costs firsthand.

The deeper problem is not that the factory model is wrong in general, but that it does not fit knowledge work at all. In a factory, the interface between stations can be fully specified in advance: a part has a known shape and tolerance before it arrives at the next station. In research and engineering, the interface is exactly what is hardest to define, and it changes as understanding develops. Treating it like a fixed handoff creates the appearance of process while undermining the collaboration that actually gets things done.

Talking to practitioners since, I have been struck by how deeply the factory model is embedded, and not only in quant teams. Entire banks are organised around it. The front office produces a number, the middle office reconciles it, the back office reconciles it again, and risk and finance reconcile it once more against their own systems. Each department exists, in large part, to check the output of the one before it. Reconciliation is not a minor operational chore in such an institution; it is the organising principle. Armies of people spend their days confirming that one team's figures agree with another's.

It is worth asking what all that reconciliation is for. The honest answer is that it compensates for the handover. If every department worked from the same data, the same definitions and the same code, there would be far less to reconcile, because there would be far fewer independent versions of the truth to bring back into agreement. Reconciliation is the cost of having let the representations diverge in the first place. It looks like control. Much of it is the institutional scar tissue of a design that made divergence inevitable.

This is also why the model is so hard to leave. A quant team can adopt a shared environment in a quarter. A bank cannot easily dismantle the departmental boundaries that decades of process, headcount, regulation and audit have hardened around the handover. The reconciliation functions are not incidental; careers, controls and reporting lines rest on them. I do not underestimate this. But it is worth being clear-eyed that much of what looks like prudent control is the expensive upkeep of a problem that need not exist. The point of a shared environment is not to reconcile faster. It is

[2] Marcos Lopez de Prado, *Advances in Financial Machine Learning* (Wiley, 2018), Section 1.3. Working paper: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3104847

to have less to reconcile.

For those who cannot leave the model, there is at least a better version of it. If the factory must remain, let it be a modern one. The lesson of Toyota is that quality is not a stage but a property of every stage. An assembly line on which inspection happens only at the end is not really a factory at all; it is a slow route to a scrapyard, where defects accumulate unseen through every station and are discovered only when the finished car will not start. A trading platform is no different. Testing cannot be an afterthought, something performed once the strategy is "finished" and handed to whoever signs off on production. It has to be present at every stage and owned by whoever works at that stage, the same way a modern line stops the moment a fault appears rather than building twenty more cars around it.

A more useful image is the professional kitchen. Not the frantic Saturday-night service of a Michelin-starred restaurant; that version of the kitchen, with its noise and urgency, is the wrong picture. Think instead of the kitchen as atelier: calm, deliberate, precise. A place where skilled people work together with shared tools, each aware of what the others are doing, unhurried but never idle. A kitchen of that kind is not sequential. Every station is visible to every other. The head chef and the junior cook share the same space and the same standards. Quality is not inspected at the end; it is everyone's responsibility throughout. No serious dish leaves having passed through one pair of hands with no awareness of what came before or after.

A quant fund should work the same way. Researchers and developers in the same room, aware of each other's constraints, catching problems early. Quality is not a downstream function.

The kitchen analogy has a further implication. A chef is not expected to build the oven or construct the fridge. A professional kitchen assumes a working environment: reliable equipment, the right tools available. Asking a chef to fabricate their own appliances would produce worse food and worse appliances. The same applies here. Researchers and developers should not be building basic infrastructure from scratch: data pipelines, execution connectors, backtesting engines, monitoring tooling. That is what the platform provides. The team's energy belongs on the signals, the models, the risk framework. Everything else is the oven, and it should just work.

The mechanism that makes this real is containerisation. A container packages not just code but the entire runtime environment (libraries, language version, dependencies) so the environment a researcher uses to develop a strategy is identical to the one that runs live. Moving between research, backtesting and production is a matter of changing configuration, not rewriting code. The familiar complaint, "it works on my machine," loses its meaning when every machine runs the same machine.

We organise the team in what we call a checkerboard structure rather than separating researchers and developers upstream and downstream. They sit together, alternate and collaborate continuously. A researcher working on a new signal works alongside the engineer responsible for the infrastructure that will run it. Knowledge moves in both directions. The result is code that is correct and deployable from the start.

In practice the boundary between researcher and developer is often blurry, and we think that is a good thing. The modern developer on this platform understands the models and contributes to the research process. The modern researcher writes production-quality code and takes responsibility for what they ship. We assume most researchers have strong development skills, and most developers have enough quantitative depth to engage seriously with the research.

Build or Buy

Commercial platforms such as Beacon and Deltix invite the obvious question. If platforms that unify research and production already exist, and can be bought, why build at all? It is the oldest question in enterprise software, and the honest answer is that it is the wrong question as posed. Build or buy is not a binary, and treating it as one is how teams reach both bad extremes: the team that builds everything, including the commodity plumbing that confers no advantage, and the team that buys everything, including the edge that was the only thing worth having.

The kitchen analogy already contains the principle. A chef buys the oven and does not cook the dish from a packet. The art is in knowing which is which. Commodity infrastructure (data storage, container orchestration, backtesting plumbing, broker connectivity) is the oven: hard to build well, conferring no edge once built, and no reason to build what can be adopted. The signals, the models, the risk framework are the dish. They are the only reason the team exists, and they are never bought.

The interesting category is the one in between, and it is where the enterprise platforms sit. Beacon and Deltix are not ovens; they are fitted kitchens, sold whole. Buying one solves a great deal at once, and for some institutions that is the right trade. But a fitted kitchen is built to its maker's plan. The cost is real, the pricing assumes an institution of a certain size, and the workflow becomes the vendor's workflow. You adapt to the platform rather than the platform to you, and the components you would most like to swap are the ones most welded in. Lock-in is not a failure of these products; it is their business model.

Our position is the third option the binary hides. Assemble the platform from open, container-based primitives: adopt the commodity layers, compose best-of-breed components at the edges, and build

only the parts that are genuinely yours. This is more work than buying a suite and far less than building one. It keeps cost proportionate to the team rather than to an enterprise licence, it avoids welding the team to one vendor's roadmap, and it leaves every boundary open so a better component can be substituted when it appears. It is also the only option that scales down: a small team priced out of an enterprise platform can still build this way.

There is one thing here that cannot be bought at all, and it is the most important. The operating model (the checkerboard, shared ownership, quality at every stage, the kitchen rather than the factory) is not a product. No licence confers it. A team can buy the best platform in the world and run the handover model around it, reconciling between departments exactly as before, and most of the value leaks away at the seams. Build or buy is a question about software. The harder question, the one this document is really about, is how the people work. That part is always built.

Building the Kitchen

Building the platform means deciding what the foundation looks like before the first strategy is written. The temptation is to start with the interesting parts: signal generation, portfolio construction, the models. Experience suggests this is the wrong order. Teams that skip the foundation and build strategies first end up with strategies that are hard to compare, hard to maintain and hard to trust. The foundation has to come first, even when it feels slow.

Data API. The data layer is where most platforms quietly fail. Not because data is technically hard to fetch, but because the failure modes are invisible. Point-in-time correctness (ensuring that a query returns only what would have been known at a given historical moment) is easy to get approximately right and difficult to get exactly right. Survivorship bias, look-ahead from revised data, misaligned timestamps: each of these can make a strategy look better in backtesting than it ever was in reality. A well-designed data API makes the correct behaviour the default and the incorrect behaviour structurally difficult. Getting this layer right is what everything else is built on.

Common strategy tooling. The building blocks of quantitative strategies (portfolio construction, signal combination, position sizing, transaction cost modelling) are not proprietary. Every team needs them; most teams build their own versions. The result is that the same components exist in dozens of slightly different forms across the same organisation, each with slightly different assumptions and edge case handling. When results diverge, nobody knows why. Implementing these components once, testing them thoroughly and making them available to everyone is not a convenience. It is what allows results to be compared and trusted.

Performance analytics. A strategy's returns tell you what happened. Analytics tell you why. Without a standard set of tools covering returns decomposition, drawdown analysis, factor attribution and cost accounting, each researcher develops their own view of what a good strategy looks like. Those views are not comparable. A strategy that looks excellent on one researcher's metrics and mediocre on another's cannot be evaluated honestly. Shared analytics create a shared vocabulary. They also create shared accountability: a strategy that passes a common standard is harder to champion with private metrics that nobody else uses.

Live monitoring. Problems in production are not announced. A strategy that drifts (signals weakening, execution quality degrading, positions creeping outside intended bounds) will continue to run until something forces attention to it. By then the cost is real. Continuous observation of positions, P&L attribution, signal behaviour and execution quality is not optional infrastructure. It is the difference between catching a problem in an hour and catching it in a week.

Execution layer. Strategies communicate intent through a standardised API; the execution layer translates that intent into orders, handles broker connectivity and manages the operational complexity of running strategies at scale. This is a platform component, something the team should not have to rebuild for every strategy. It is discussed in detail in the Live Trading section.

The platform must be built with researchers, not just for them. A platform designed only by engineers, however capable, risks solving the wrong problems. Researchers know what data they actually need, how they think about portfolio construction, what a useful performance report looks like and what slows their work down. The gap between a data API that an engineer thinks researchers want and one that researchers actually use is often wide. That knowledge needs to be in the room when the platform is being built.

In practice, the platform and the first strategies are often built in parallel. The team cannot wait for a complete platform before starting research, and waiting would be the wrong instinct anyway: building infrastructure in isolation, without real strategies pushing against it, tends to produce the wrong infrastructure. The feedback loop between strategy development and platform development is valuable and should not be broken.

That said, a minimal foundation should exist before the first strategies are implemented. At minimum this means a working data API, a basic portfolio construction library and a consistent project structure enforced by Rhiza, a tool that keeps every strategy repository aligned with a common template. Without these in place, the first strategies will each invent their own solutions, and unpicking that fragmentation later is costly. A small shared foundation built early pays back many

times over.

Keeping the Platform Consistent

Solving fragmentation at the strategy level (shared tooling, common data access, consistent analytics) does not solve it at the infrastructure level. A team that builds excellent shared libraries but allows each strategy repository to manage its own CI pipeline, its own linting configuration and its own Python version has solved half the problem. The scaffolding that enforces standards is as important as the standards themselves. A linting rule that is disabled in one repo, a CI check that has been bypassed in another, a security fix that landed in the template but was never propagated: each of these is a small crack. Across twenty repositories they become a maintenance burden that nobody fully owns and nobody can easily quantify.

The standard response is to generate scaffolding once, at project creation, and leave it to each team to keep up. This works until the platform evolves, and the platform always evolves. The moment the canonical template changes, every existing repository is behind. Nobody notices until the divergence is severe enough to cause a problem, and by then it is spread across dozens of repos in dozens of different ways.

Rhiza was built on a different premise: scaffolding is not a one-time generation but a continuous synchronisation. When the central template changes (a new CI workflow, an updated linting config, a change to the containerisation setup), Rhiza opens a pull request in each downstream repo with a clear diff of what changed. Owners review, adapt where needed and merge. Nothing is forced and nothing is missed. The scaffolding stays current the same way the code does: through a visible, reviewable change process rather than manual propagation or silent drift.

Backtesting

A backtest is only useful if it is honest. The history of quantitative finance is littered with strategies that looked compelling on paper and disappointed in production, and the gap is rarely explained by bad ideas. It is almost always explained by a backtest that was, in some subtle way, too optimistic.

The most common culprit is look-ahead bias: the strategy had access to information during the backtest that it could not have had at the time. This can happen through data that has been revised after the fact, through target variables that leak future information into the features, or simply through a timestamp that is off by one bar. The platform enforces strict point-in-time data semantics. Every data query is anchored to a historical timestamp, and the data API makes it structurally difficult to request information that would not have been available at that point.

Transaction costs are the second place where backtests mislead. A strategy that ignores market impact, bid-ask spreads and borrow costs can look highly profitable while being economically meaningless at any realistic scale. The platform models transaction costs explicitly, and the cost model is calibrated against real execution data where available. A researcher should be able to see what a strategy's net performance looks like under conservative, realistic and optimistic cost assumptions before it is taken seriously.

Overfitting is harder to guard against because it is partly a discipline problem rather than a tooling problem. The platform supports walk-forward analysis and out-of-sample evaluation as standard, making it easy to separate the in-sample period used for development from the out-of-sample period used for evaluation. But no tool prevents a researcher from repeatedly tweaking a strategy until the out-of-sample period looks good too. The culture of the team, and the scrutiny applied during strategy review, matters as much as the infrastructure.

A backtest that passes these tests is not a guarantee. Markets change, and a strategy that worked for ten years may stop working. The backtesting framework provides evidence, not certainty. The team should treat strong backtest results with interest and some scepticism in equal measure.

Live Trading

The traditional view of going live treats it as a deployment event: a moment of special procedures, checklists and risk. This architecture treats it as a promotion between environments. That distinction matters more than it might appear. When going live is a special event, it is also a moment of maximum uncertainty: the code has never run in this environment, with these configurations, against this broker. When it is a promotion, it is a moment of minimum uncertainty: the strategy has already run in an environment that was deliberately made indistinguishable from production.

The mechanism is the separation of code from configuration. The container (the fixed, versioned artifact that packages the code and its entire runtime environment) does not change between research, backtesting, paper trading and live. What changes is the configuration file it reads: where to find data, which parameters to use, which execution venue to connect to, what risk limits to respect. The container has no knowledge of which environment it is in. It reads its configuration and runs. Moving a strategy to live is a one-line change. If something breaks, it would have broken in paper trading. If it does not break in paper trading, the team has genuine evidence, not just hope, that it will not break in live trading.

This separation also makes incidents recoverable. Every configuration file is versioned. Every deployment is a known container image combined with a known configuration state. If something

goes wrong, the team can reconstruct exactly what was running and with what parameters. Rolling back is a configuration change, not an emergency deployment under pressure.

Each strategy is implemented as a service with a standardised API, a deliberate choice with consequences beyond convenience. A strategy that knows about brokers, about scaling, about order routing is a strategy that is hard to test in isolation, hard to replace components of, and hard to reason about when something goes wrong at 3am. The standardised API is a contract: the strategy satisfies it by expressing intent; the infrastructure honours it by handling execution. Each side can be tested, replaced and reasoned about independently. The strategy does not become safer by knowing more about the world it operates in. It becomes safer by knowing less.

Prime broker connectivity. The strategy never communicates with the outside world directly. It expresses intent through its API (buy this instrument, in this quantity, with this urgency) and the platform's execution layer handles everything else: translation into FIX or a proprietary protocol, order routing, fill reconciliation, position and margin queries with the prime broker. This boundary is among the most consequential in the system. Errors here are not silent; they are immediate and financial. Keeping the strategy clear of that complexity is not just good architecture: it is what makes the system safe to operate.

Switching brokers or adding a new venue is a configuration change to the execution layer. The strategy is unaffected. In backtesting and paper trading the same interface is present, backed by a simulated fill engine rather than a live connection. The strategy code is identical across all environments. The broker, like everything else, is a detail the platform absorbs so the strategy does not have to.

Risk Management

The standardised strategy API is what makes professional risk management composable. Because every strategy expresses intent through the same interface, the execution layer can intercept, inspect and act on every order without the strategy knowing or caring. Pre-trade risk checks (position limits, notional limits, concentration limits, maximum order sizes) sit as middleware between the strategy's intent and the market. The strategy does not implement them; it does not need to know they exist. They are enforced at the boundary, configured independently of the strategy code and versioned alongside everything else. Tightening a limit during a volatile period is a configuration change. It does not require touching the strategy, redeploying a container or restarting a process.

The same composability extends to external tools. Because the strategy API is standardised, any system that speaks the same interface can attach to the execution layer: independent risk engines,

prime broker risk controls, OMS and PMS systems, accounting and bookkeeping infrastructure. Professional risk management at scale requires components that a trading team should not be building from scratch: position reconciliation, P&L attribution, regulatory reporting, margin calculations. The architecture makes it possible to plug these in without modifying the strategies that generate the orders. The strategy is a signal source. What happens to the signal after it leaves the strategy is the platform's responsibility.

This has a direct consequence for how risk is monitored in production. Because all order flow passes through a single execution layer, aggregate exposure across strategies is observable in one place. Gross and net exposure, drawdown against defined thresholds, execution quality and fill rates: these are properties of the execution layer, not of individual strategies. A strategy that begins behaving inconsistently with its historical profile surfaces at the layer that processes its orders, before the inconsistency becomes a loss.

The kill switch is a first-class platform concept precisely because of this architecture. Stopping a strategy cleanly (cancelling open orders, unwinding positions in an orderly way, returning the system to a known state) is possible because the execution layer has full visibility of what the strategy is doing and full authority to stop it. This is not an emergency procedure. It is something the team tests regularly, and it works because the architecture was designed with it in mind from the start.

The Impact of AI

The arrival of capable AI tools has changed one thing more than any other: the cost of turning an idea into a running experiment. That cost was always the hidden tax on quantitative research. A researcher with a sharp intuition about market structure would spend days implementing the infrastructure to test it (data wrangling, signal construction, backtest scaffolding, performance reporting) before learning whether the intuition was worth anything. Most of the time it was not, but the cost of finding out was high enough that fewer ideas got tested than should have.

AI compresses that cycle. The effect is not that researchers work less; it is that the ratio of thinking to implementation shifts, and ideas that were previously not worth the cost of testing become worth testing. A platform built on shared tools amplifies this further: AI works best when the environment is clean and the interfaces are consistent. Fragmented infrastructure and AI are a poor combination: the model has no reliable context to work from. A clean, consistent platform makes AI more useful, not less relevant.

This creates a risk the document would be incomplete not to name. Faster iteration means more

experiments, which means more opportunities for overfitting. The discipline problems described in the Backtesting section (look-ahead bias, data snooping, the temptation to tweak until the out-of-sample period looks good) do not disappear when iteration is cheap. They get worse. A researcher who can run fifty backtests in the time it previously took to run five will, without discipline, find five times as many spurious results. The platform provides the infrastructure for honest evaluation; the team culture provides the discipline to use it honestly. AI raises the stakes for both.

In production, AI is most valuable where humans are least reliable: sustained attention to continuous streams of data. A live strategy generates thousands of data points a day. Monitoring for anomalies (fills that do not match expectations, signals that drift, execution quality that degrades) requires exactly the kind of pattern recognition that AI handles well and humans find tedious. Problems surface faster. The kill switch gets pulled sooner.

What AI does not do is generate insight. It can prototype a signal but cannot determine whether the signal is real or spurious. It can flag an anomaly but cannot decide whether the market has changed or the model is broken. It can write the code but cannot judge whether the strategy belongs in production. Those judgements belong to the team. AI removes friction. The work it exposes is still the work.

Conclusion

The problems described in this document are not technical failures. They are organisational ones. The handover model, the personal scripts, the reinvented wheels: none of these happened because teams lacked talent or ambition. They happened because the structures in place made sharing hard, translation inevitable and accountability diffuse. Better tooling alone does not fix that. The platform has to be accompanied by a different way of working.

What this document argues for is not a specific technology stack but a set of principles: shared environment over translation, quality at every stage over inspection at the end, common infrastructure over individual reinvention, accountability through shared ownership over alibi through separation. The specific tools (containers, Rhiza, a standardised execution API) are expressions of those principles, not the principles themselves. A team that internalises the principles will make good decisions about the tools. A team that adopts the tools without the principles will find ways to recreate the old problems inside the new infrastructure.

It is worth being honest about what this way of working asks of people. A researcher who has

built their own tools, their own scripts, their own way of doing things is being asked to give that up, or at least to hold it more loosely. An engineer who owns a pipeline is being asked to share that ownership with the team. This is not a small ask. The autonomy that comes with personal ownership is genuinely valuable; it is fast and unencumbered and answers to nobody. What it trades away is comparability, continuity and the ability to build on what came before. The platform makes that trade explicitly. Whether it is the right trade depends on what the team is trying to do. For a team serious about compounding knowledge and operating at scale, we think it is.

The edge in quantitative trading is scarce and hard to find. The platform exists to make sure that the search for it is not cluttered by problems that have already been solved.

The ideas described here are not untested. A small number of firms have built in roughly this way for some time: unified environments, shared infrastructure, researchers and engineers working as a single team rather than adjacent ones. Their ability to iterate quickly, compound knowledge and operate at scale is visible in their results. The gap between those firms and those still working under the handover model is not primarily a gap in talent or capital. It is a gap in how the work is organised.

The challenges this document points towards (transition paths for existing platforms, the organisational work of shared ownership, the right boundaries between research and production at different scales) are ones we are actively thinking about. Readers working on the same problems are welcome to reach out at thomas@jqr.ae.

Acknowledgements

Mohammed Abu Sharikh — developer and friend since our days together at Winton Capital. Many of the ideas in this document were first tested in conversation with him.

In memory of Alexander Belopolsky — our often heated debates made me a much stronger engineer than I would otherwise have been.

We are grateful to the early adopters who have put Rhiza to work in their production environments. Their feedback has shaped the tool in ways that no amount of internal testing could have.