VEKTOR

BY

INVESTPUPPY

OUR BETTER WAY · VEKTOR BY INVESTPUPPY

THE CONFIGURABILITY GAP

Why the two camps in wealth management technology cannot converge — and what Vektor built instead.

InvestPuppy

June 2026

This paper is written for portfolio managers, operations leads, and technology decision-makers at boutique and institutional wealth management firms. It does not assume a background in software engineering. Where technical concepts appear, they are explained in terms of their commercial consequences — because that is where the consequences land.

OUR BETTER WAY · THE SERIES

The Unvarnished series documented what we witnessed in the rooms where financial software implementations fail — thirteen papers, published under the InvestPuppy name, naming the failure modes clearly because naming them seemed more useful than another optimistic vendor promise.

Our Better Way is the other half of that account. Each paper describes what we built in response to what we witnessed: the architectural decisions, the engineering discipline, and the operational choices that define how InvestPuppy approaches the problems the Unvarnished series named. Not theory. A record of how we applied what we learned.

This is Paper 03. Paper 01 (BW-01) covers the cost of technical debt and the structured architecture that prevents it. Paper 02 (BW-02) covers the Vektor : House institutional implementation pathway. This paper covers the deeper architectural question that underlies both: the configurability problem that every serious financial software platform eventually has to answer.

---

1. The Problem with the Demo

Every financial software evaluation begins the same way. The platform does everything. You watch it screen a universe, optimise a portfolio, allocate capital, generate an order file. You ask whether it handles your market. Yes. Your fee structure. Yes. Multiple clients with different configurations. Yes.

The demo is a single-client, single-market, single-configuration environment. It was built to answer yes. UNV-13 named this pattern precisely: the demo closes contracts. The platform that emerges after client three is a different thing from the one that was demonstrated.

> *“The client does not have the vocabulary to ask whether what they are watching is architecture or aspiration.” — UNV-13: Built for client demo — rebuilt by client 3*

The problem is not dishonesty. It is the normal consequence of building for one client first. With one firm, one market, one set of requirements, hardcoding everything is reasonable. Lot sizes in a constant. Fee percentages in a configuration file. Market rules in a dictionary. It works — until the second client needs

something slightly different from the first.

By client three, the codebase carries three clients' worth of if-statements, special cases, and accumulated exceptions. The change that should take an hour takes a week — not because it is complex, but because no one is certain what they will break. The demo never had a client three. Client three was never in the room.

---

2. Two Camps, One Gap

The wealth management technology landscape is split into two camps. Each has a fundamental strength. Each has a corresponding weakness. Neither has solved both at once.

Camp 1 — Legacy PMS: deep configurability, outdated architecture

Temenos, Avaloq, ERI Bancaire, Charles River, SimCorp. These platforms have been running at private banks and asset managers for twenty, thirty years. The reason they survive is not inertia. It is parameterisation depth.

New exchange? Add a row to the parameter table. New fee structure? Configure it — no code change, no deployment, no developer. New settlement rule per market? Data-driven. Operations handles it. Per-client business rules? An override hierarchy resolves system default → product rule → client override — in sequence, automatically.

A Temenos installation at a Swiss private bank might carry 50,000 or more parameters. Each one represents a business requirement that was met without a code change — without a developer, a test cycle, or a deployment. That is why these systems run for decades without fundamental rewrites. The system bends to business needs. The business does not bend to the system.

The weakness is structural. The architecture is from another era. Scaling means buying a bigger machine, not adding more instances. A parameter change may require a nightly restart. Deployments are quarterly events. The technology is proprietary — vendor-specific languages, vendor-specific databases, vendor-specific tooling. You are not just using a platform. You are inheriting a set of dependencies you cannot escape.

Camp 2 — Digital-native: modern architecture, shallow configurability

Thought Machine Vault, Mambu, 10x Banking. Born cloud. Microservices with independent deployment and scaling. API-first. True multi-tenancy. Automated testing. Deploy multiple times per day. Open standards throughout.

Everything that legacy PMS cannot do, these platforms do by default. The engineering discipline is real. The infrastructure is genuinely modern. They are, by every architectural measure, better-built systems.

The weakness is the configuration ceiling. Smart contracts or configuration files cover eighty percent of cases. Then you are writing custom code. Adding a new exchange often means updating code, deploying, testing. Business rule changes require developer involvement — the rules live in code, not in data. Per-client overrides are either unsupported or require custom code paths per client.

> *When you hit the configuration ceiling on someone else's platform, you are writing bespoke code on top of someone else's infrastructure. That is not a vendor relationship. That is a trap.*

The gap between what operations can configure and what requires a developer is large. And it grows with every client. By client three, each client's requirements have accumulated in the codebase as custom logic. The architecture is modern. The configurability is not.

The gap

No platform in the market currently occupies both positions simultaneously. Legacy systems cannot retrofit microservices without rebuilding from scratch — Temenos has been attempting this for years; the result remains fundamentally T24 underneath a new wrapper. Digital-native systems cannot retrofit deep parameterisation without introducing the domain model complexity that made legacy systems what they are.

The gap exists because each camp is starting from the wrong end. Legacy PMS would need to rebuild their architecture to get modern infrastructure. Digital-native platforms would need to rethink their domain model to get deep configurability. Each path leads back to the other camp's weakness. The table below maps the gap across the dimensions that matter commercially.

Dimension	Legacy PMS (Temenos, Avaloq)	Digital-native (Thought Machine, Mambu)	Vektor
Architecture	Monolith	Microservices	Microservices
Cloud-native	Retrofitted — not born here	Born cloud	Born cloud
Configuration depth	Deep — 50,000+ parameters	Shallow — configuration ceiling	Deep — four-layer hierarchy
Add a new exchange	Minutes — parameter table	Days — code change, deploy	Minutes — API call, no deploy
Add a client override	Minutes — parameter	Weeks — custom code per client	Minutes — API call (C3+)
Audit and compliance	Proprietary audit trail	Application logs	Architectural enforcement — MAS FEAT-aligned, 5yr retention
Vendor lock-in	Extreme — proprietary language	Moderate — platform dependency	None — open source, AWS-portable
Ops can change without a developer	Yes — parameter-driven	No — requires developer	Yes — data-driven from day one
Target market	Tier 1–3 banks ($5M+ licence)	Tier 1–2 banks, challengers	Boutique DPMs, EAMs, family offices

These are platforms built for Tier 1 and Tier 2 banks. Vektor is not competing for that market. What legacy PMS does for a Swiss private bank — configuration without code, operations without developers, durability without rewrites — the Vektor parameter architecture does for your boutique practice. At your price point. On modern infrastructure.

---

3. Why Fintechs Skip This

The answer is simple, and it applies to every platform that started small and grew into the client three problem.

Parameterisation is not a feature. It is infrastructure. It does not appear in a demo. It does not close a contract. No investor ever funded a company because it had a well-designed parameter hierarchy. No client ever signed because the fee structure was data-driven rather than hardcoded.

With one client, you do not need it. Hardcode everything. It works. The codebase is clean and simple. Delivery is fast.

With two clients, the differences are small enough to manage with a few conditional branches. Still manageable. Still fast.

By client three, the conditional branches have branches. The exceptions have exceptions. The developer who originally wrote the client-one code has moved on. The developer who wrote the client-two modifications does not fully remember the client-one assumptions they built on top of. The change that should take a day takes two weeks — not because the requirement is complex, but because the codebase has become a system that nobody completely understands.

> *Retrofitting a parameter service requires touching every service in the system. By the time the pain arrives, the cost of fixing it exceeds the cost of living with it. This is the architecture trap. It assembles itself from rational decisions made one at a time.*

The engineers who built legacy PMS systems understand this viscerally. They have seen what fifty thousand parameters produce: a system that bends to business needs for twenty years without a rewrite. They also carry the cost. The legacy architecture they built it on is the reason those systems cannot become what they would build today.

Vektor was designed by people who have been in both rooms. The founding team built and operated systematic investment processes within licensed portfolio management environments — the same environments that run on the legacy systems described in this paper. They have seen the override hierarchy working at scale. They have also seen the configuration ceiling appear in digital-native replacements. Vektor was built with that experience in every architectural decision. The parameter layer was the first infrastructure built. Because we have seen what happens when it is the last.

---

4. What Vektor Built Instead

The Vektor parameter architecture takes the domain knowledge from legacy PMS — the override hierarchy, the progressive complexity layers, the principle that business rules live in data, not in code — and implements it on modern infrastructure: microservices, containers, cloud-native services, API-driven configuration, independent scaling, automated testing.

The result is a four-layer parameter hierarchy. Each layer can be configured by operations — without a developer, without a deployment, without a code change. The hierarchy is progressive: Layer 1 was built in the first development cycle. Layers 2, 3, and 4 extend the same architecture as multi-client complexity arrives.

Layer	What it governs	Legacy PMS equivalent	Ops can change it?
1 Market Reference	Exchanges, currencies, data providers, lot sizes	SYSTEM_PARAMS, EXCHANGE_MASTER, CURRENCY_MASTER	Yes — from day one. Adding a new exchange is an API call.
2 Business Rules	Confidence thresholds, approval limits, rebalancing triggers	LOCAL_PARAMS, BUSINESS_RULES	Yes — from C2 delivery.
3 Client Overrides	Per-client fee structures, risk limits, mandate parameters	CUSTOMER_PARAMS — override hierarchy	Yes — from C3 delivery. No client requires custom code.
4 Product Config	Product-level entitlements and feature flags	PRODUCT_PARAMS, ENTITLEMENTS	Yes — from C3+ delivery.

The override resolution follows the same logic as a Temenos parameter hierarchy — but delivered via a REST API with a performance cache instead of a proprietary transaction boundary. Client-specific requirements resolve automatically: client override → product configuration → business rule → market reference. The first non-null value in that chain is the effective parameter.

The practical consequence: adding a new exchange is an operator action taking under 60 seconds. Vektor currently has eight exchanges configured and active — SGX, LSE, NYSE, NASDAQ, HKEX, SIX, ASX, TSE. Each was configured without a developer, without a deployment, without a code change. Adding the next exchange — Euronext, Tadawul, Bursa Malaysia, or any listed equity market — follows the same path: exchange parameters, lot size rules, settlement conventions, and data provider mappings are all operator-managed. No developer. No deployment. No code change.

The same applies to currencies, data providers, confidence thresholds, and client-specific fee structures. None require engineering involvement. None require a deployment.

The design validation question

Every design decision in the Vektor platform is evaluated against a single question: can operations change this without a developer?

If the answer is no, and the thing in question changes more than once a year, it should be data-driven. Lot sizes change when exchanges change their board lot rules. Confidence thresholds change when market conditions change. Client fee structures change when mandates change. None of these should require engineering involvement. None of them do.

> *"The best parameter hierarchies are invisible to the people using them. They just know they can configure the system — without raising a ticket, without waiting for a deployment, without explaining to a developer why a business rule changed."*

---

5. The Audit Trail as Architecture

WP-06 (Audit Trail and Compliance Architecture) describes what Vektor's audit trail captures: every action, timestamped, PM-attributed, across every stage from signal selection to order generation. WP-08 (AI and Machine Learning Philosophy) describes how the AI governance framework was developed with reference to the four principles of the MAS FEAT framework, and is consistent with the AI governance principles published by the FCA/PRA (UK), FINMA (Switzerland), and SFC/HKMA (Hong Kong). This section addresses the underlying design question: how is the audit trail enforced, not just specified?

The distinction matters. An audit trail that is a feature — something added to the system to capture what happens — can be misconfigured, partially implemented, or bypassed by a developer who forgets to add a log entry. An audit trail that is architectural — enforced at the infrastructure level, applied to every system action before service code runs — cannot be bypassed without changing the architecture. A developer cannot bypass it by forgetting. The logging happens regardless of whether the developer thinks about it. There is no path through the system that does not pass through the audit log.

This extends to the parameter service. Every change to a system parameter — every threshold adjustment, every exchange addition, every client override — is captured with the user identity, the timestamp, and the full state at the time of the change, not just a flag that something changed. This has been demonstrated in production: on 13 June 2026, the addition of the Australian Securities Exchange (ASX) and the reactivation of HKEX were each captured in the compliance audit log with complete event_data payloads — exchange code, lot size, currency, timezone, settlement days, and active status — timestamped and attributable. Not because the developer who wrote the parameter service remembered to add logging. Because the logging is applied at the level below the service code, automatically, on every write.

For a portfolio manager operating under MAS, FCA, FINMA, or DFSA oversight, this matters in a specific way: the compliance documentation is not produced by asking the system what happened after the fact. It is produced by the system as a structural output of normal operation. The audit trail exists before anyone asks for it.

---

6. What This Means for You

If you are a boutique DPM, EAM, or independent wealth manager:

The configurability architecture is what makes the “any exchange, any currency, any mandate” claim credible rather than aspirational. When you ask whether Vektor supports your market, the answer is not “yes, we’ve configured it.” The answer is “yes, and your operations team can reconfigure it without involving us.”

Eight exchanges are configured and active today: SGX, LSE, NYSE, NASDAQ, HKEX, SIX, ASX, TSE. Adding yours — Euronext, Tadawul, Bursa Malaysia, or any listed equity exchange — takes under 60 seconds and requires no developer involvement. Not a project. Not a deployment. Not a developer conversation.

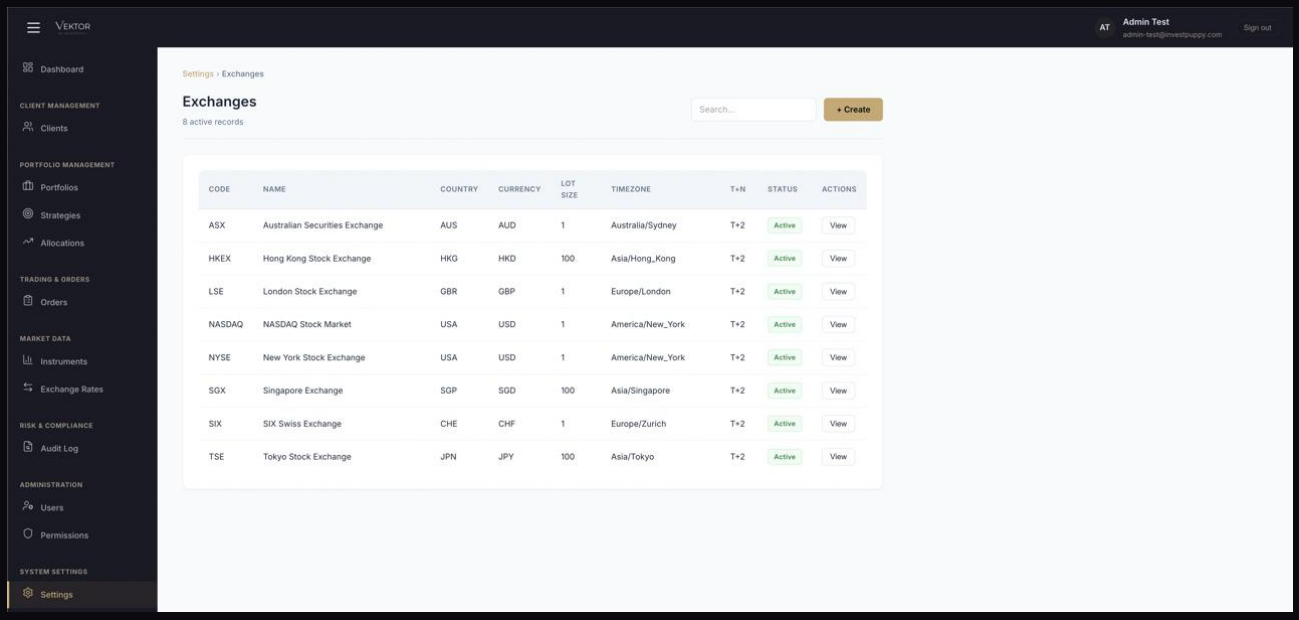
If you are evaluating the Alloy Partners institutional path:

The distinction between configuration and customisation is the gap UNV-13 named. It is the gap between a parameter that operations can change and a code change that requires a developer, a test cycle, and a deployment. Vektor was built to be the former, entirely, before the first client signed.

> *We documented what client three looks like in UNV-13. This paper describes what we built so that client three is not a crisis. The parameter layer was the first infrastructure we built. Because we have seen what happens when it is the last.*

> *LEGACY APPROACH → THE VEKTOR APPROACH*

Legacy approach	The Vektor approach
The demo shows a single-client, single-market environment. Client three was never in the room.	→ The parameter architecture was built before client one. Client three requires an API call, not a rewrite.
Business rules live in code. Changing a fee threshold requires a developer, a test cycle, and a deployment.	→ Business rules live in data. An operations team member changes a threshold via the admin portal. No developer. No deployment.
Adding a new exchange requires a developer to update code and deploy. Days or weeks.	→ Adding a new exchange is an API call to the parameter service. Minutes. No deployment. No developer involvement.
Per-client requirements accumulate as if-statements and custom code paths. By client three, the codebase carries three clients' worth of exceptions.	→ Per-client requirements resolve through the override hierarchy. Client-specific configuration is a parameter. The codebase does not change.
The audit trail is a feature — added by developers who remembered to log. A developer who forgets creates a gap.	→ The audit trail is architecture — enforced at the infrastructure level, below the service code. There is no path through the system that does not pass through the audit log.
Vendor lock-in is structural. Proprietary language. Proprietary database. The cost of leaving exceeds the cost of staying.	→ No lock-in. Open source stack. AWS-portable. The cost of leaving is a configuration change, not a rewrite.



Exchange registry — eight exchanges configured and active. Adding the next exchange is an operator action under 60 seconds. No developer. No deployment. No code change.

InvestPuppy Pte Ltd is a Singapore-based financial technology company building Vektor — a systematic listed equity portfolio management platform for boutique discretionary portfolio managers, independent wealth managers, and family

offices. The platform gives wealth management professionals institutional-grade systematic portfolio construction at a price point accessible to boutique practices.

This is Paper 03 of the Our Better Way series — InvestPuppy's account of what was built in response to the failure modes documented in the Unvarnished series. All thirteen Unvarnished papers are available ungated at investpuppy.com/unvarnished.