VEKTOR

BY

INVESTPUPPY

OUR BETTER WAY · VEKTOR BY INVESTPUPPY

The Cost of Technical Debt

Why architecture is a business decision, not a development preference.

For DPMs, family office principals, and prospective partners

InvestPuppy

September 2026

Most software projects fail not because the team lacked talent, but because relentless pressure to ship features forces architecture and quality into a "later" that never comes. This paper explains what technical debt costs in business terms, why the interest compounds faster than most organisations expect, and what a considered alternative looks like in practice. Written by the InvestPuppy founding team from experience building and operating financial technology platforms.

This paper is written for business and technical leaders, not just engineers. Concepts are framed in business impact terms — speed, cost, risk, and competitive position — because that is where the consequences land.

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.

THE CORE ARGUMENT

Every software system accumulates a form of debt. When a team cuts corners to meet a deadline — choosing the quick solution over the right one — they borrow against the future. Like financial debt, this is not inherently wrong. Sometimes borrowing is the right call. The problem is when the interest is not acknowledged, not tracked, and not repaid.

At low levels, technical debt is invisible. Features ship on time. The system works. The shortcuts are forgotten. Then the system grows. The shortcuts compound. What once took a developer a day now takes a week — not because the feature is more complex, but because the codebase has become a maze of interdependencies that nobody fully understands.

> "The system becomes so fragile and slow to change that adding even simple features takes disproportionate effort."

Eventually the cost to fix exceeds the cost to rebuild. But rebuilding takes months or years. The business has lost its window to scale. Competitors have caught up. The engineering team that built it has burned out and moved on. This is not a hypothetical — it is the standard trajectory for systems built under sustained feature pressure without architectural discipline.

THE DELIVERY TRAP

Why speed kills architecture — and why "later" never comes.

The pressure to ship features is real and legitimate. Investors want progress. Customers want new capabilities. Competitors are moving. The business case for cutting a corner today in exchange for shipping next week is often compelling in the moment.

The problem is structural. Once a team ships under pressure using shortcuts, the next deadline is just as tight — tighter, because now they are building on top of the shortcuts. The "fix it properly later" conversation happens again, with the same outcome. The debt accumulates silently in the background while the roadmap expands in the foreground.

This is the delivery trap: the very act of moving fast degrades the system's ability to move fast in the future. A team that ships twelve features in the first six months may find that the next six months produce four — not because the team got slower, but because the system became harder to change.

Stages of technical debt accumulation:

Early	Fast delivery. Features shipping on schedule. Shortcuts accumulating. Architecture decisions deferred. Test coverage thin. Each new feature makes the next slightly harder.
Middle	Delivery slowing. Estimates getting longer. More bugs than expected. Shortcuts compounding. Developers spending significant time understanding existing code before they can change it. Bug fixes creating new bugs.
Late	Progress nearly stopped. Key engineers leaving. The system has become a liability. Adding a feature requires touching code that is too risky to modify. The team is maintaining, not building.
Crisis	Competitor has caught up. Cost to fix now exceeds cost to rebuild. Rebuilding takes a year. The business has no good options.

The trap is self-reinforcing: pressure creates shortcuts, shortcuts create complexity, complexity creates slowness, slowness creates more pressure. Breaking the cycle requires a conscious decision — not just at the engineering level, but at the business level.

THE COMPOUNDING COST

Technical debt as a business risk, not just a development problem.

Technical debt is routinely framed as a code quality issue — something engineers care about and stakeholders tolerate. This framing is wrong, and it is why debt is rarely addressed until it becomes a crisis. The real costs of technical debt are commercial, not technical.

Slower time to market:

Features that should take weeks take months. The business loses its ability to respond to market opportunities faster than competitors. Developers must understand and work around existing complexity before adding new functionality. The ratio of understanding-to-building time increases as debt accumulates.

Security vulnerabilities:

Systems built quickly without security architecture are harder to secure retroactively. Patching a shortcut often introduces new exposures. Security is an architectural property. It cannot be added after the fact without understanding the full system.

Scaling failures:

Systems built without load or growth assumptions break under success. The moment a product gains traction, the infrastructure becomes the bottleneck that limits it. Shortcuts in database design, service boundaries, and infrastructure provisioning are invisible under low load. They become structural at scale.

Engineering attrition:

The best engineers leave systems they cannot work in productively. Senior engineers are drawn to systems where their decisions matter. In a high-debt system, most decisions are constrained by legacy choices. The work becomes maintenance, not engineering.

Opportunity cost:

Every hour spent managing the consequences of debt is an hour not spent acquiring customers, iterating on product-market fit, or building the next capability. For a small team, this is the most acute cost. A two-person team spending 40% of their time managing debt effectively has a one-person engineering capacity.

> "For a small team startup, you cannot afford to rebuild. Every hour spent untangling shortcuts is an hour not spent acquiring customers."

DESIGN-FIRST IS NOT SLOWER

Front-loading the investment rather than paying compound interest.

The most common objection to architectural discipline is speed: designing properly takes time that early-stage teams do not have. This objection confuses short-term delivery speed with long-term velocity. They are different things, and optimising for the first at the cost of the second is one of the most expensive mistakes a technology company can make.

Architectural investment works the same way as financial investment. The weeks spent establishing clear service boundaries, proper infrastructure patterns, and automated pipelines at the outset are paid back many times over in the months that follow — in faster feature delivery, safer deployments, smaller incident blast radius, and a codebase that new engineers can understand and contribute to quickly.

Service boundaries:

Shortcut approach: Build one system. Split later if needed. Structured approach: Define domain boundaries before writing code. Each domain owns its data, its logic, and its API from day one.

Infrastructure:

Shortcut approach: Provision manually. Automate later. Structured approach: Infrastructure as code from the first environment. Every resource is defined, versioned, and reproducible.

Security:

Shortcut approach: Open access initially. Restrict when there is something to protect. Structured approach: Network security, WAF, and access controls are foundational layers — built before any service is exposed.

Monitoring:

Shortcut approach: Add alerting when something breaks. Structured approach: Observability built in from the first deployment. Failures are visible before they affect users.

Testing:

Shortcut approach: Write tests for the important bits. Cover more later. Structured approach: Automated test coverage is a release criterion, not an aspiration. The pipeline enforces it on every change.

The upfront cost of a structured approach is measured in weeks, not months. The avoided cost — in development velocity, security incidents, scaling failures, and engineering attrition — is measured in years.

BUILT TO AVOID DEBT FROM THE START

The InvestPuppy architecture as a case study.

InvestPuppy applied the structured approach from the first line of infrastructure. The Vektor platform is built on a layered, modular architecture with clear domain separation — not because the team had unlimited time, but because a two-person team cannot afford the alternative. The architecture was designed so that rebuilding is never necessary.

The architectural principles described here have been tested in production institutional environments, not only in a startup context. The founding team led a production implementation of a generic OpenWealth API layer on a Tier 1 portfolio management system, deployed across Europe, the Middle East, and Asia Pacific. That credential informed every architectural decision in Vektor.

One item is worth naming directly: the RBAC maker/checker workflow — required for institutional governance where an analyst runs optimisations and a CIO approves before any order is submitted — is Priority 1 on the pre-live engineering roadmap. The architecture is designed to accommodate it; the implementation is in progress.

THE LAYERED INFRASTRUCTURE MODEL

Layer 1 — Network: VPC, subnets, routing, VPN, traffic flow controls. Every other layer depends on network topology. Changing it after services are deployed is expensive and high-risk. Done once, done right.

Layer 2 — Security: WAF, IAM policies, security groups, encryption at rest and in transit. Security built on top of services is always incomplete. Security built into the network layer protects everything above it automatically.

Layer 3 — Platform: Container orchestration, load balancers, shared observability and monitoring infrastructure. Standardising the platform once means every service benefits from the same operational

practices without duplicating them.

Layer 4 — Services: Domain-specific services: market data, portfolio management, trade execution, research and signal generation. Services change most frequently. Isolating them means a change to the trading service cannot affect the market data service.

DOMAIN-DRIVEN SERVICE DECOMPOSITION

Market Data: Instrument data, price history, exchange rates. Read-only foundation. Can be updated or replaced with a different provider without touching any other service.

Portfolio: Client portfolios, positions, cash management, strategy assignment. Portfolio logic can evolve without affecting the trading or research layers.

Trade: Order generation, signal application, execution, fill confirmation, order audit trail. The highest-risk service. Isolation means a bug in trade execution cannot corrupt portfolio or market data.

Research: Signal generation, ML validation, strategy backtesting, efficient frontier computation. Can run intensive computation without affecting the operational latency of the trading service.

> "A two-person team can operate what would otherwise require a dedicated platform engineering group — because the architecture does the work that headcount would otherwise do."

MANAGING THE INEVITABLE

When debt accumulates anyway — visibility and a payback plan.

No system is completely free of technical debt. Some debt is intentional and entirely acceptable — choosing a simpler implementation to ship faster when the risk is understood and the payback is planned. The key distinction is not between systems with debt and systems without it. It is between organisations that can see their debt and those that cannot.

Invisible debt is the dangerous kind. When shortcuts are not acknowledged, they are not tracked, not prioritised, and not repaid. They accumulate until they cause an incident, a scaling failure, or a security breach — at which point the cost of addressing them is orders of magnitude higher than it would have been if they had been managed with intention.

Practices for managing debt visibly:

Track debt as first-class backlog items: Technical debt items have issue tracker tickets alongside feature requests. They are visible, prioritised, and assigned — not accumulating in a list that nobody reviews. Debt that is visible can be traded off explicitly against feature work.

Dedicated refactoring capacity — the 20% rule: Reserve a consistent proportion of engineering capacity — typically 15–20% — for debt repayment and system improvement. Not negotiable with the roadmap. Refactoring capacity that is not protected will be consumed by features in every planning cycle.

Automated quality gates: Code linting, static analysis, test coverage thresholds, and security scanning run automatically on every pull request. Changes that fail cannot be merged. Quality gates prevent new debt from being introduced.

Architecture reviews at defined intervals: Periodic structured review of the system against current load, security requirements, and business needs. Scheduled and documented, not ad-hoc. Architecture reviews surface debt that has accumulated below the threshold of individual tickets.

Make debt visible to stakeholders: Report debt metrics — time spent on unplanned work, deployment frequency, change failure rate — alongside feature velocity in business language. Stakeholders who cannot see debt cannot make informed decisions about when to address it.

CONCLUSION

Technical debt is not an engineering problem that happens to have business consequences. It is a business problem that happens to manifest in code. The decisions that create it — to ship faster now, to fix it later, to add one more feature before the architecture is ready — are made in boardrooms and planning meetings as much as in development sprints.

The organisations that manage it well are not the ones with the most talented engineers or the most rigorous code reviews. They are the ones where the business understands what it is trading when it chooses speed over architecture — and makes that trade consciously, with a plan to repay it.

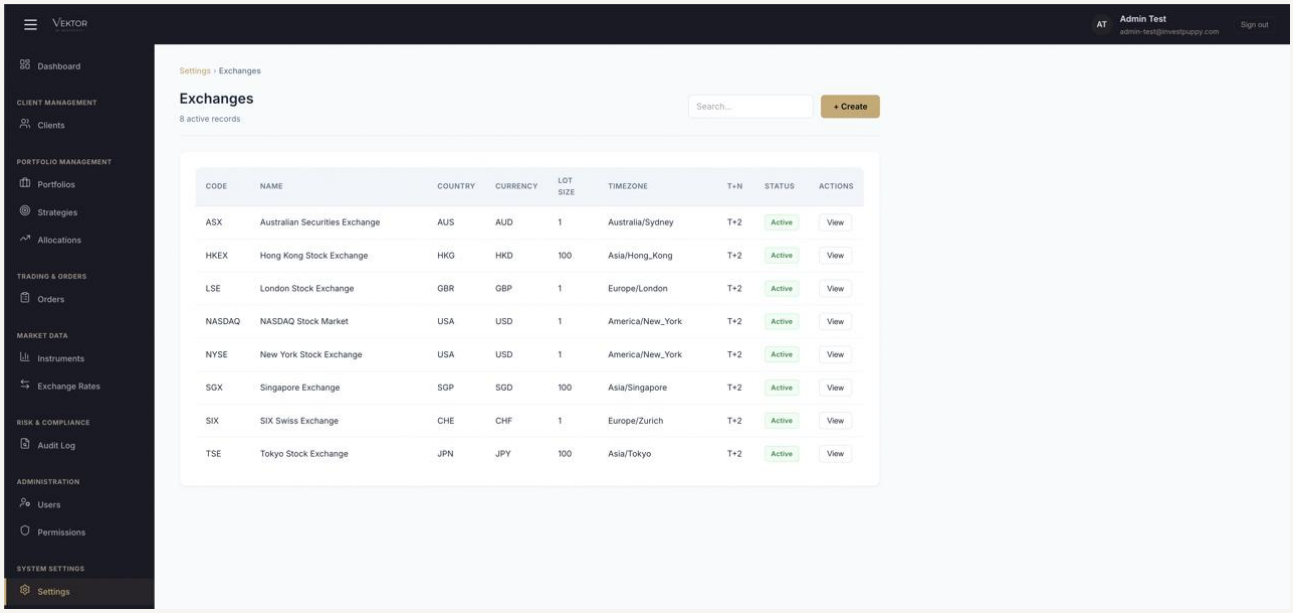
For a startup operating with a small team and a compressed timeline, the calculus is clear: architectural investment made early costs weeks. Architectural debt left unaddressed costs the business its window to scale. The question is not whether to invest in architecture. The question is whether to pay now or pay more later.

THE COST OF SHORTCUTS vs THE RETURN ON STRUCTURED ARCHITECTURE

Legacy approach	The Vektor approach
Features that take weeks begin taking months	→ Delivery velocity sustained as the system grows
Security debt that cannot be patched without full redesign	→ Security built into the foundation, not retrofitted onto it
Scaling failures at the moment of commercial traction	→ Infrastructure that scales with the business, not against it
Engineering attrition as the work becomes unmaintainable	→ A codebase that engineers can understand and contribute to quickly
Rebuilding cost that exceeds the original development budget	→ A two-person team operating at the scale of a platform engineering group

> *The upfront cost was weeks, not months. The compounding return is measured in years.*

ABOUT INVESTPUPPY



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 01 of the Our Better Way series — InvestPuppy's account of what was built in response to the failure modes documented in the Unvarnished series. Paper 02 (BW-02) covers the Vektor : House institutional implementation pathway. Paper 03 (BW-03) covers the configurability architecture. Paper 04 (BW-04) covers machine learning governance and the AI boundary decision. All thirteen Unvarnished papers are available ungated at investpuppy.com/unvarnished.

VEKTOR
INVESTPUPPY

The Full Document Suite

Organised by access tier — not a catalogue, a map.
Full reading-order guidance in the Due Diligence Navigation Guide.

UNGATED — 29 DOCUMENTS

VEKTOR COMMERCIAL

– Vektor At a Glance

– Vektor Brand Story

– Vektor Platform Brochure

– Due Diligence Navigation Guide

– What Vektor Is Not

– Why Not the Incumbents?

– Why Vektor

– Reference Output — SGX Equity Mandate

VEKTOR RESEARCH PAPERS

– WP-09: Risk and Limitation Disclosure

– WP-10: Evaluating Performance

OUR BETTER WAY

● **BW-01: Technical Debt (you are here)**

– BW-02: Vektor House Implementation

– BW-03: The Configurability Gap

– BW-04: AI as Instrument

UNVARNISHED

– UNV-01: Why Implementations Fail

– UNV-02: Nobody Owns It

– UNV-03: Complexity as Cover

– UNV-04: Plan Last

– UNV-05: The Change Request Trap

– UNV-06: Invisible Stakeholders

– UNV-07: The Hammer, the Tool, and the Methodology

– UNV-08: Life After Live

– UNV-09: The U in UAT

– UNV-10: Nine Women

– UNV-11: The AI Revolution — Automating Bad Practice at Scale

– UNV-12: Model Bank — the perfect solution for banks that don't exist

– UNV-13: The parameter that broke the camel's back

WHITE PAPERS

– WP-12: Rebalancing: Drift Thresholds and Disclosure

– WP-13: Portfolio Valuation Methodology

investpuppy.com/documents

REGISTRATION REQUIRED — 13 DOCUMENTS

VEKTOR COMMERCIAL

– Vektor FAQ

– One Platform, Three Delivery Models

– Reference Output — SGX · NASDAQ · LSE · 21 May 2026

VEKTOR RESEARCH PAPERS

– WP-Index · Research Series Index

– WP-01: Quantitative Portfolio Construction

– WP-02: Signal Optimisation

– WP-03: Capital Allocation

– WP-04: Indicator Selection

– WP-05: Multi-Currency

– WP-06: Audit and Compliance

– WP-07: Technical Architecture

– WP-08: AI and ML Philosophy

WHITE PAPERS

– WP-11: Signal Robustness Architecture — Why walk-forward validation was considered and discarded

How to register: investpuppy.com/research

NDA REQUIRED — 8 DOCUMENTS

VEKTOR COMMERCIAL

– First 90 Days

– Mutual Non-Disclosure Agreement

– Proof Partners Programme

– Workflow Integration Guide

– Vektor Operational Walkthrough — Client to Portfolio Valuation

– Proof Partners Agreement

INVESTPUPPY

– Alloy Partners Programme

OTHER

– Vektor OpenWealth Integration — Commercial Overview

Available once a Proof Partners conversation begins. Start at investpuppy.com/proof-partners