

You hired a coder. You needed an owner.

Coding agents have made software cheap to write, and enterprises are hiring developers to write it. The decision looks like staffing. It is risk allocation — and the role worth building in-house is not the one being hired.

SUMMARY OF THE ARGUMENT

A new pattern has taken hold in enterprise IT. Rather than commission software, organisations hire full-stack developers fluent in coding agents and build their own systems. The reasoning is sound as far as it goes: the cost of writing code has collapsed, so the cheapest way to get software appears to be to employ someone who can now write a great deal of it.

This paper argues that the reasoning stops one step short. Writing code was rarely where enterprise projects failed. Most failed at requirements — deciding what the operation actually needs — and at adoption — getting people to work differently once the system exists. The technical failures that remain are failures of engineering, not of typing. A developer with a coding agent makes the typing cheap and leaves requirements, adoption and engineering exactly as hard as they were. Meanwhile the enterprise has quietly taken on the delivery risk a partner would otherwise carry: talent, schedule, outcome, quality, architecture, and the knowledge that walks out when the developer does.

Seen this way, the build decision is not a staffing question but a risk-allocation question, and it belongs with the CFO as much as the CIO. The paper sets out the risks an enterprise accepts when it hires to build, explains why the timing of cost against value matters more than the headline cost, and describes what to look for in a delivery model that carries the controllable ones instead — including how to tell a software factory that equips your developers from one that delivers to your business.

It closes with the role that is worth building in-house: not the coder but the owner — a forward-deployed person who owns the requirement on one side of the build and the adoption on the other, and who is accountable for the outcome. If an enterprise wants capability that compounds, that is the capability to build.

IN THIS PAPER

- 1 The trend is rational
- 2 Writing code was rarely the hard part
- 3 What an enterprise takes on when it hires to build
- 4 A risk decision, not a staffing decision
- 5 Not all software factories are the same thing
- 6 Starting from built, not from blank
- 7 Governing the running system, not only the build
- 8 Ownership without the risk
- 9 Build the owner
- 10 Where building your own engineering is right
- 11 Six questions before you approve the hire

1 The trend is rational

Through 2026 a quiet shift has run through enterprise technology budgets. Systems that would once have gone out to tender — a lending platform, a case-management tool, an order-to-invoice workflow — are increasingly built in-house by a small team of developers working with coding agents. Job descriptions that asked for a stack now ask for fluency with the agents that write in it. Vendors of those agents have raised capital at valuations that assume the pattern will become the norm, and the phrase *software factory*, once a piece of delivery jargon, has become a category on its own.

It would be easy to dismiss this as fashion. It is not. The cost of producing working code has fallen sharply, and in the delivery work this series draws on, a capable developer with a good agent now produces in days what used to take weeks. An enterprise that notices this and concludes it should stop paying vendors to write code is drawing a reasonable inference from a real change.

This paper accepts the premise. Code is cheap. The question it asks is whether writing code was ever the expensive part — and if it was not, what the enterprise is actually buying when it hires to build.

Nor is this an argument that enterprises should never build software themselves. Section 10 sets out where building a substantial engineering capability is the right decision. The argument is about one specific mistake: treating cheap code production as if it were cheap software delivery.

2 Writing code was rarely the hard part

Enterprise projects do fail for technical reasons. Systems buckle under load, migrations corrupt data, integrations break in production, security gaps are found by the wrong people, and codebases become so tangled that they cannot be changed safely. Nobody who has run delivery for long would claim otherwise.

But look at what those failures have in common. None of them is a failure of *writing* code. They are failures of engineering: architecture chosen without the load in mind, integration assumptions nobody tested, verification that confirmed the happy path, design decisions that made the second year harder than the first. And in the experience of most people who have run enterprise delivery, the more common and more costly failures sit elsewhere. Systems that faithfully implement a requirement that was never what the business needed. Systems that work and are not used the way they were designed to be used, because the process around them never changed.

The research most often cited on enterprise AI points the same way. It finds most pilots producing no measurable financial impact, and identifies the redesign of the workflow, not the choice of technology, as the factor most associated with returns.

Coding agents have made the writing cheap. They have done nothing for the requirement, nothing for adoption, and nothing for the engineering judgement that decides whether a system survives contact with production. In practice they can make all three harder, for three reasons.

The bottleneck moves rather than disappears

When code took months, the organisation's capacity to specify, review, test and govern was roughly matched to the rate at which code arrived. When code takes days, that balance breaks. A single developer can now produce more than the organisation can meaningfully review, and the volume of working, plausible software outruns anyone's ability to say whether it is the right software. Output rises; finished, trusted systems do not.

Requirement errors now arrive at machine speed

A developer fluent in an agent is not, by virtue of that fluency, fluent in lending regulation, clinical workflows, or a charity's fund flows. When the intent behind a system is poorly distilled, the agent builds the wrong thing quickly, completely, and convincingly. The prototype looks finished. The misunderstanding inside it surfaces months later, usually in front of a user or an auditor, and by then it is load-bearing.

Plausible code hides engineering debt

An agent produces code that runs, reads cleanly and passes the tests it wrote for itself. That is precisely what makes its weaknesses hard to see. Load behaviour, failure handling, data integrity under concurrent use and the consistency of one module with the next are not visible in a demonstration. They surface later, under real use, when they are expensive to fix. The faster the code arrives, the more of this debt can accumulate before anyone looks.

The problem, in other words, is not that AI lets software be built with too little human involvement. It is that the humans are involved at the wrong points: heavily in the middle, where the machine is now strong, and too lightly at the two ends, where judgement decides the outcome. The role that owns those two ends, the requirement and the adoption, is the subject of Section 9 and the thread that runs through the rest of this paper.

Coding agents made writing code cheap. They did not make requirements, adoption or engineering any easier, and those are where projects fail.

3 What an enterprise takes on when it hires to build

The decision to hire developers is usually framed as a cost comparison: salaries and tooling against a vendor's quote. That framing misses most of what the enterprise is acquiring. When it hires to build, it does not only acquire capacity. It acquires every risk that a delivery partner would otherwise have carried.

Risk	What it looks like in practice
Talent	Developers fluent in coding agents are scarce and mobile. The enterprise must find them, pay a premium for them, and keep them in a market that is bidding for the same people.

Risk	What it looks like in practice
Delivery	Salaries run whether or not anything ships. A project that slips by two quarters costs two quarters of a team, and nobody else absorbs it.
Outcome	If the system ships and does not move the number it was built to move, the loss is entirely the enterprise's. Nobody has agreed to fix it without being asked.
Key person	The knowledge of why the system works as it does lives in prompts, commits and one person's memory. When that person leaves, the enterprise owns a large codebase that no one fully understands.
Quality and regulation	Agents are good at the happy path and weak on audit trails, entitlements, residency and behaviour under failure. Those gaps surface late — at security review or audit — when they are most expensive to close.
Architecture	Each project rebuilds authentication, roles, audit and reporting slightly differently. After three projects the enterprise has three stacks and three security postures, and no shared foundation.
Verification	Tests written by the same agent that wrote the code confirm that the code does what it does. They do not confirm that it does what the business needed.

None of these risks is new. Every one of them existed when enterprises hired traditional development teams, and every one of them is the reason the industry built delivery firms, fixed-price contracts and service levels in the first place. What is new is that the falling cost of code has made them easy to overlook. A developer who costs less than a vendor looks like a saving, and the risks arrive later, one at a time, on a different budget line.

What about a full engineering organisation?

A sophisticated enterprise will answer that it is not hiring a developer. It is building a governed engineering function, with architects, QA, security, platform engineering, product owners and engineering leadership. That is a serious answer, and it does address several of the risks in the table.

But it is worth being clear about what it is: building a software factory of one's own. The cost of that organisation, the years it takes to become effective, and every risk it fails to remove still sit with the enterprise, and the factory it builds serves a single customer. Where software is the business, that is the right investment, as Section 10 argues. Where software runs the business, the question is whether owning a factory is a better use of capital than owning the problem and the outcome.

4 A risk decision, not a staffing decision

If the build decision is really about who carries these risks, it belongs on a different desk. A CIO can compare developers with vendors. A CFO compares risk-adjusted returns, and the comparison looks different from there.

The timing of cost against value

Return on a system depends less on what it costs to build than on when its value begins and how certain that value is. An in-house build starts its costs on the first day of hiring. Value begins at some later date that nobody can promise, if the project lands at all. For the whole of the build, the cost curve runs ahead of the value curve, and the gap between them is the enterprise's exposure.

A delivery model in which payment is triggered by go-live reverses that order. Cost sits behind value rather than in front of it. If the system also starts from components that already exist, value arrives sooner. And if the delivering party has committed to correcting the outcome when it slips, the delivery risk it can control moves to the party contractually responsible for it.

Not every risk moves, and no honest delivery model claims otherwise. Process ownership, adoption, data quality, executive sponsorship and regulatory accountability stay with the enterprise, because only the enterprise can manage them. What moves is the controllable delivery risk: schedule, build quality, architecture, verification, and the correction of the outcome within the agreed scope. The point is not that risk disappears. It is that each risk sits with the party able to control it.

What a risk-adjusted comparison asks

- Who pays if the system ships late, and for how long?
- Who pays if it ships on time and does not move the number?
- Who holds the knowledge of how it works if the people who built it leave?
- Who finds the regulatory gaps, and when — before go-live or at audit?
- Who carries the cost of the second and third systems, if each one starts from nothing?

Paper Six in this series made the commercial version of this argument: an outcome can only be bought from someone whose delivery model makes it contractable. The point here is the mirror image. An enterprise that hires to build has not avoided paying for an outcome; it has agreed to pay for the effort and to carry the outcome itself.

Hiring to build does not avoid paying for an outcome. It means paying for the effort and carrying the outcome yourself.

5 Not all software factories are the same thing

The natural response to this argument is to look for a software factory rather than a developer, and there is no shortage of them. The term now covers several quite different things, and the differences matter more than the label.

What is offered	Who uses it	Who carries the delivery risk
Prompt-to-application builders	Business users and founders, mostly for first versions and prototypes	The enterprise

What is offered	Who uses it	Who carries the delivery risk
Coding agents and agent platforms for engineering teams	The enterprise's own developers, working in its codebases	The enterprise
Specification and lifecycle control planes	Product, engineering and business teams upstream of the code, with execution handed to the coding agent of their choice	Mostly the enterprise
Factories that deliver to the business	The delivering firm, working with the enterprise's owners	Shared, and in some models carried by the deliverer from go-live

The first three are tools. They are often excellent, and they make the enterprise's own people more productive. But they do not change who carries the risks in Section 3 — they reinforce the in-house model rather than replace it. Most software factories, in other words, are sold to your developers. The question for an enterprise that wants to move controllable delivery risk off its own books is whether a factory delivers to the business.

What separates a factory that delivers

Several properties distinguish a factory that can carry delivery risk from one that equips others to carry it. A buyer can test for each.

- **The business confirms something it can use before anything is built.** A clickable prototype that the people who will use the system can walk through is a far stronger confirmation than a signed requirements document, and it is the single step that does most to remove requirement risk.
- **The specification is a durable artifact.** Intent is captured in a form — requirements, use cases, stories, test cases — that survives the people who wrote it and can be read by the next person.
- **Verification is independent of the builder.** Testing runs in layers, automated and human, and is not written by the same agent that wrote the code.
- **The work starts from something already built.** Shared foundations and domain components mean a project does not begin from an empty repository.
- **The running system is governed, not only the process that built it.** This is the subject of Section 7.
- **The enterprise owns the result.** This is the subject of Section 8.

Taken together, these describe a chain rather than a tool: a specification distilled by people who understand the business; a prototype the users confirm; a governed build on tested foundations; verification independent of the builder; deployment and adoption owned jointly; source and specification owned by the enterprise; and a running system whose decisions stay deterministic and auditable. That is the model this series argues for. People own intent and outcomes. An AI-native factory owns execution, and the delivery risk that comes with it.

6 Starting from built, not from blank

The largest difference between delivery models is not in how fast they write code but in how much code they do not have to write. A factory that builds every system from a specification on a blank sheet is faster than a team doing the same by hand, but it still starts from zero each time and still resolves problems that have been solved before.

A delivery model built around reusable domain cores works differently. A lending system starts from a ledger, an amortisation engine, payment application and delinquency handling that already exist and have already been tested. A real-estate system starts from inventory, party and escrow models that are already in production elsewhere. The project's work is the part that is genuinely specific to this organisation, and the jurisdictional rules for its market are, wherever possible, already written as a pack rather than discovered during build.

This matters for risk as much as for speed. A component that is running in other organisations has had its edge cases found. A ledger that has reconciled real money is a different proposition from one generated last week. And when the second and third systems are built on the same foundation, the enterprise inherits one architecture, one security posture and one audit model rather than three.

A core is not a template

A domain core is not a template, a low-code component library or a vertical SaaS module, and the difference matters. Templates and low-code components trade flexibility for speed, and usually bind the enterprise to the platform that hosts them. A domain core is tested engineering — data models, calculations, rules and audit already proven in production — delivered as source code the enterprise owns, and specified and extended through the same specification-first process as the rest of the system. It shortens the build without narrowing what the system can become.

The same logic applies to what the enterprise already has

Most organisations are not starting fresh. They have legacy systems they want to replace, and increasingly they have last year's internally built AI prototypes that worked in a demonstration and stalled before production. Both are brownfield problems, and both have the same first step: re-derive the specification from what exists. Once the system has been respecified, there are three honest routes. Rebuild it on an existing domain core where one fits. Rebuild it custom where none does. Or extend it in place, where replacement cannot be justified. A delivery model that offers only one of those routes will tend to recommend it.

The fastest code to deliver is the code you do not have to write, because it already runs somewhere and its edge cases have already been found.

7 Governing the running system, not only the build

Much of the current conversation about governed software development concerns the build process: traceability from requirement to code, a record of who decided what, documentation that

does not drift. Those are real improvements and every serious factory now offers some version of them. They answer the question *how was this built?*

A regulator, an auditor or a board asks a different question: *how does this behave?* Which rule did the system apply to this case, on what date, and who signed the outcome? What happens when the language model is unavailable, or wrong? Those are properties of the running system, and no amount of traceability in the build process supplies them.

Earlier papers in this series described the architecture that answers them: a deterministic layer that computes, applies rules and decides, and a language model that drafts and explains but never has the final word; every workflow touched by a model provided with a fallback so that the system operates fully with the language layer down; and an immutable record of every decision, with the rule and the role behind it. A system built by coding agents without that discipline will typically put the model in the decision path, because that is the easiest thing for an agent to build. It will look finished in the demonstration and fail the first serious review.

The reason for this order is a principle of governance, not a preference of architecture.

- **The deterministic layer is the authority.** Rules, entitlements and calculations must give the same answer every time, so they are written in code that can be tested.
- **The language model is the interface.** It reads, extracts, drafts and explains, which is work where probabilistic output is useful and can be checked.
- **The audit trail is the evidence.** Every decision records the rule, the date and the role behind it, so it can be defended later.
- **The fallback is continuity.** Because no decision depends on the model, the system keeps operating when the model is unavailable or wrong.

For an enterprise deciding how to build, the practical test is simple. Ask whoever is building — your developer or a factory — what the system does when the model is switched off. If the answer is that it stops, the model is making decisions, and the risk in Section 3 under *quality and regulation* has not been addressed.

8 Ownership without the risk

The strongest argument for building in-house is not cost. It is control. Enterprises that have spent a decade on subscription software and vendor roadmaps want to own their systems, and they remember what outsourcing used to mean: a black box, a dependency, and knowledge that left when the contract ended.

That instinct is right, and it has historically forced a trade-off. Outsource, and you reduce delivery risk but accept lock-in. Build in-house, and you keep control but carry every risk yourself. The trade-off is not a law of nature. It is a property of particular commercial and technical models, and it can be designed out.

The old outsourcing fear	What removes it
A black box nobody can inspect	The enterprise owns the source code, under a perpetual licence, from delivery.
A shared platform the enterprise cannot leave	A dedicated, single-tenant instance, hosted by the deliverer or by the enterprise for residency.
Knowledge that leaves with the team	A specification, use cases and test cases that remain as the record of intent.
Paying for effort whether or not it works	Payment triggered at go-live, with the outcome judged over the period that follows and corrected without argument if it slips.
A vendor roadmap the enterprise must follow	Changes delivered as the enterprise's own releases on its own codebase.

When those conditions hold, the enterprise gets the control it wanted from building in-house and sheds the controllable delivery and outcome risk it did not want. That is the combination the build decision should be looking for, and it is the combination neither the developer route nor a pure tool can offer.

What makes an outcome commitment credible

One condition separates an outcome commitment from a slogan. An outcome can only be owned when what it rests on is defined together before the build: the specification, the outcome measure and its baseline, the dependencies on other systems, and the responsibilities that stay with the enterprise, such as adoption and the quality of its data. Within that agreed scope the correction should be unconditional — no argument about why the number slipped, only the work to restore it. Without the scope, a guarantee is either empty or impossible to honour.

9 Build the owner

None of this means an enterprise should build no capability of its own. It means it should be precise about which capability is worth building. The answer follows from Section 2. If projects fail at the requirement and at adoption, those are the competencies that compound, and they are the ones a delivery partner cannot supply on the enterprise's behalf.

One role, both ends of the build

The person who owns the requirement and the person who owns adoption are, in the best organisations, the same person. This series has called that role the forward-deployed engineer: someone who sits with the business, distils what the operation actually needs into a specification, confirms it with the people who will use the system, and then stays through deployment until the new way of working has replaced the old one and the outcome number has moved. It is a judgement role more than a coding role. It requires enough technical depth to know what a system can and cannot do, and enough operational depth to know what the business needs it to do.

An enterprise that wants to build in-house should build this. Its forward-deployed people own intake, distillation, adoption and the outcome. The delivering factory owns the build, the shared foundations, the governance of the running system and the commitment to correct the outcome. Each side carries the risk it can actually control.

How the two sides work together

- **Paired at intake.** The enterprise's forward-deployed people work alongside the deliverer's at intake and learn the distillation discipline through the work itself. It is taught by apprenticeship, not by course.
- **Co-signed specifications.** Where the deliverer is guaranteeing an outcome, it must co-sign the specification that outcome rests on. The gate can lighten as the enterprise's people mature; it should not disappear.
- **Stewardship of what is owned.** Because the enterprise owns the source, it needs technical stewardship — environments, integrations, the ability to read what it owns. That is an IT ownership role, not a build role.
- **More work, not less.** An enterprise with its own forward-deployed people removes its own intake bottleneck and can put more systems through delivery, not fewer.

Do not hire coders. Build an owner on each side of the build.

10 Where building your own engineering is right

The argument of this paper has limits, and it is stronger for stating them. For some organisations, building a substantial in-house engineering capability is plainly the right decision.

- **When software is the product.** A technology company, or a bank whose digital channel is its competitive position, should own its engineering. The argument here applies to enterprises where software runs the business but is not the business.
- **For small internal tools.** A team that needs a dashboard or a utility, with little regulatory exposure and a short life, should build it with the tools at hand. Commissioning it would be overkill.
- **For exploration.** Prototypes built to learn what is possible are cheap and valuable. The risk arises only when they are promoted to production without being respecified.

The line runs through systems of record, regulated workflows, and anything that must still behave correctly in three years. Those are where the risks in Section 3 compound, and where the build decision deserves a risk-adjusted comparison rather than a salary one.

11 Six questions before you approve the hire

The series ends each paper with something a buyer can use. These six questions apply whether the proposal on the table is to hire developers, to license a tool, or to engage a factory — including the kind this paper describes.

1. **Who owns the specification when the person who wrote it leaves?** The acceptable answer is an artifact the next person can read. *It is in the code* is not an answer.
2. **What will the people who use the system confirm before it is built?** The acceptable answer is something they can walk through. A signed document is the weakest version.
3. **Who tests the work, and did they write it?** The acceptable answer is layered verification that is independent of the builder.
4. **What does the system do when the language model is switched off?** The acceptable answer is that it keeps working, with the deterministic layer making every decision that matters.
5. **What happens to the cost if the system ships late, or ships and does not move the number?** The acceptable answer names who pays. If it is only the enterprise, the risk has not moved.
6. **What does the enterprise own at the end?** The acceptable answer is the source, the specification and the right to change both, with no dependency it cannot leave.

None of these questions is about who can write the software. AI has made that cheap. The question now is who will own what the software is supposed to accomplish, and who will carry the risk of getting it there.

Build that owner. Let the machine do the middle.

About this series

This is the eighth paper in a series on enterprise AI as an operating system rather than a tool. Paper One established where the value is. Paper Two set out the method for re-deriving a process from its outcome. Paper Three described delivery, and Paper Four turned the method into a strategy for a company of 50 to 500 people. Paper Five applied it to the accounting and advisory practice; Paper Six separated the commercial model from the delivery model that makes an outcome contractable. Paper Seven described the governed workforce that does the routine work once the strategy is written. This paper addresses the decision that sits underneath all of them: who builds, and who carries the risk.

This is a position paper. Its claims about where projects fail are drawn from long experience of enterprise delivery rather than from a survey, and the references below are context rather than proof. Section 5 describes categories of tool and delivery model generically and makes no claim about any particular vendor's product or terms. The fourth of those categories — people owning intent and outcomes, an AI-native factory owning execution and delivery risk — is the model Xamun builds to. Every paper is readable at xamun.ai/whitepapers.

Sources and references

- MIT NANDA, *The GenAI Divide: State of AI in Business* (2025) — on the share of enterprise pilots reporting no measurable P&L impact.
- McKinsey & Company, *The State of AI* (2025) — workflow redesign as the strongest correlate of EBIT impact.
- Public reporting through 2026 on investment in coding-agent platforms and in the “software factory” category, including the category's recognition among the defining trends at the 2026 AI Engineer World's Fair.
- Xamun Position Papers No. 1 to No. 7 (August–October 2026), xamun.ai/whitepapers.

No. 8 in the Xamun position-paper series. © 2026 Xamun Technologies. Version 1.0, October 2026.