SUPPLY CHAIN RESEARCH · EDITORIAL FEATURE

Build versus Buy in the Age of AI

Why the Easy Button Is a Mirage

Artificial intelligence has quietly changed the economics of building software. It has not turned building into an easy button. For supply chain and technology leaders, the old discipline, buy the commodity and build only what truly sets you apart, matters more now, not less.

Independent, vendor-neutral research for supply chain and technology leaders

Supply Chain Research | supplychainresearch.com | 2026

Contents

A decision framework for the AI era

01 Executive summary

For as long as enterprises have run on software, they have faced the same question: build it, or buy it. The arrival of capable artificial intelligence coding tools has convinced many leaders that the question has been settled in favor of building. If an assistant can write code in seconds, the reasoning goes, then the cost and risk of building custom systems must have collapsed, and the balance must have tipped away from expensive packaged software toward bespoke tools tailored to exactly what the business needs. This guide argues that the reasoning is half right and dangerously incomplete. AI has materially lowered the cost of writing code. It has not lowered the cost of owning software, and owning software, not writing it, is where the money, the risk, and the regret have always lived.

This is written for supply chain and technology leaders weighing that decision, and for the teams who will live with the consequences. It is deliberately even-handed. There are real cases for building and real cases for buying, and the right answer depends on whether a capability sets you apart, how mature the vendor market is, and whether you have the talent to own a system for its entire life. But we say honestly that the current evidence does not support the easy-button narrative. The most rigorous studies of AI-assisted development show gains on narrow tasks and, in some settings, losses on complex ones. Independent research shows AI-generated code arriving with more duplication, more churn, and more security defects. And none of it touches the parts of software that were always the hard parts: understanding the domain, designing the architecture, integrating with the systems you already run, getting the data right, and maintaining the whole thing for years. The pages that follow lay out the classic framework, the base rates for building, the true lifetime cost, what AI has and has not changed, how the buy side is strengthening, and a practical way to decide.

60 to 90%
of a custom system's lifetime cost is maintenance, not the initial build

56% less value
the average large IT build delivers against what was predicted

19% slower
how much a 2025 trial found experienced developers were with AI on real tasks

What the evidence says

- **AI lowered the cost of code, not the cost of ownership.** Writing code was never the expensive part. Maintaining, securing, integrating, and evolving software is, and AI does not remove that burden.
- **The productivity evidence cuts both ways.** A controlled trial found a 56% speed-up on a bounded task, while a 2025 trial found experienced developers 19% slower on real work in mature codebases.
- **AI-generated code shows measurable quality and security costs.** Independent analysis finds rising duplication and churn, and studies find a large share of AI-written code contains vulnerabilities.

- **The hard parts remain human.** Requirements, architecture, integration, data quality, testing, compliance, and change management are most of the work, and AI accelerates only the coding slice.
- **For supply chain, the vendor market is deep and getting deeper.** Established suites are pouring research into embedded and agentic AI, which strengthens the case to buy the core and build only the difference.

02 The question has not changed, but the temptation has

Something has shifted in how organizations talk about building software, and it is worth naming precisely. A year of rapid progress in AI coding assistants, from inline completion tools to agents that can scaffold an entire application from a prompt, has produced a genuine and understandable excitement. Demonstrations that once took a skilled engineer a week now appear on screen in minutes. Leaders who have spent years frustrated by the compromises of packaged software, the features they do not need, the workflows that do not quite fit, the vendor roadmaps they cannot control, look at these demonstrations and draw a natural conclusion: if building is now this fast, perhaps we should build.

The temptation is real, and it is not foolish. The cost of producing a working prototype has fallen sharply. What the temptation misses is that a prototype is not a product, and producing code is not the same as owning software. The distinction sounds pedantic until you have lived through the difference. A demonstration handles the expected case. A production system handles the thousands of unexpected cases: the malformed record, the network timeout, the concurrent update, the edge condition that appears only at scale, the security probe, the regulatory audit, the integration that breaks when a partner changes a field. Getting from the demonstration to the production system was always the majority of the work, and it is precisely the majority that AI helps with least.

So the question has not changed. Build versus buy is still a question about strategic differentiation, total cost of ownership, time to value, control, and risk. What has changed is the temptation, and the temptation is running ahead of the evidence. The purpose of this guide is to slow that down, not to dismiss AI, which is a real and useful tool, but to put it in proportion. The leaders who navigate this well will be the ones who separate the genuine change, cheaper code, from the imagined change, cheaper software ownership, and who apply the same discipline that has always separated good technology decisions from expensive ones.

A note on what this guide is not

This is not an argument against building, and it is not an argument that AI coding tools are without value. Both positions would be wrong. There are capabilities an organization should build because they are the source of its advantage, and AI can make building those capabilities meaningfully faster. This is, rather, an argument for proportion: for understanding exactly what AI changes, exactly what it does not, and how to weigh both in a decision that will shape an organization's cost base and agility for a decade. The most dangerous outcome is not building or buying. It is deciding to build on the belief that AI has removed the hard parts, discovering two years later that it has not, and being left with a system nobody fully understands and cannot afford to maintain.

The forces converging on the build temptation

If the underlying question is unchanged, it is worth being precise about why the temptation to build has grown so sharply. Three forces have converged, and each is powerful on its own. Understanding them is the first step to resisting the ones that lead in the wrong direction, because a temptation you can name is a temptation you can weigh.

The first force is the visible collapse in the cost of a first draft. A capable coding assistant can turn a paragraph of description into a running prototype in the time it once took to schedule the kickoff meeting. Leaders see this happen on a screen and reason, understandably, that if the first ninety percent appeared in an afternoon, the last ten percent cannot be far behind. The impression is vivid, immediate, and almost entirely about the cheapest part of the work. It says nothing about integration, data quality, security, or the decade of maintenance that follows, and yet it is the impression that most often drives the decision.

The second force is pressure from the top of the organization to have an artificial intelligence story. Boards ask what the company is doing with AI, and building something visible feels like a more convincing answer than buying a capability that a competitor could buy too. The instinct is understandable, but it confuses activity with advantage. A custom system built to satisfy a board's curiosity is an expensive way to generate a talking point, and it commits the organization to owning that system long after the presentation is forgotten.

The third force is accumulated frustration with packaged software. Every leader has lived through an implementation that ran long, a renewal that ran high, or a roadmap that never delivered the feature they were promised. Against that memory, the idea of building exactly what you want, unconstrained by a vendor's priorities, is genuinely appealing. The frustration is real and often justified. What it obscures is that the frustrations of ownership, the upgrades that break, the integrations that drift, the single engineer who understands the system and then resigns, are simply moved in-house rather than removed.

None of these forces is irrational, and none should be dismissed. But each of them measures the wrong thing. The first measures the cost of the first draft rather than the cost of the finished, operated, and maintained system. The second measures optics rather than advantage. The third measures the pain of a bad vendor rather than the pain of becoming your own vendor. A disciplined decision acknowledges all three feelings and then sets them aside in favor of the two questions that actually determine the outcome: does this capability differentiate us, and can we own it for its entire life.

03 The classic test: core versus context

Before any discussion of cost or tooling, the build-versus-buy decision turns on a single strategic question, and the clearest framing for it is now two decades old. In his 2005 book on managing innovation, the strategist Geoffrey Moore drew a distinction between what he called core and context. Core is any activity that creates durable, differentiating advantage, the reason a customer chooses you over a competitor, the thing rivals would need years to replicate. Context is everything else: work that is often necessary, frequently mission-critical, and sometimes urgent, but that does not differentiate. Payroll is context for almost every company. So is email, and so, for most enterprises, is the software that runs a warehouse or routes a truck.

Moore's prescription follows directly from the distinction. An organization should pour its scarce resources, its best people, its capital, its attention, into core, and should ruthlessly minimize, standardize, automate, or outsource context. The reason is not that context does not matter. It matters enormously, and doing it badly can sink a company. The reason is that doing context brilliantly earns no premium, because customers do not choose you for it. Effort spent building a merely adequate version of something you could have bought is effort not spent on the thing that actually sets you apart. As one practitioner puts it, building capability that does not differentiate you is motion without progress.

The differentiation test

The practical way to apply the framework is a short series of questions about any capability you are considering building. Would a customer pay a premium purely because of this capability? If you executed it flawlessly, would competitors need years to catch up? Does the story you tell investors or your board about why you win depend on this area? If the honest answer to these is no, the capability is context, and context is a buy candidate almost by definition. If the answer is yes, you may have found something worth building, and the rest of this guide is about how to build it, or a thin version of it, without the usual regret.

The trap that catches organizations is the belief that their context is core. Almost every company believes its processes are unique, and almost every company is mostly wrong. A distribution operation may run its warehouse in a way it considers distinctive, but the underlying capabilities, receiving, putaway, picking, packing, shipping, cycle counting, are industry-standard, and dozens of mature vendors implement them at a depth a single company would struggle to match. The genuine differentiation, if it exists, is usually a thin layer on top: a particular optimization, a specific customer-facing service, a proprietary piece of logic. That thin layer may be worth building. The commodity beneath it almost never is. Distinguishing the two is the first and most important act of judgment in the entire decision.

Capability	Usually core or context	Default posture
Warehouse execution (WMS)	Context for most	Buy
Transportation management (TMS)	Context for most	Buy
Demand and supply planning	Mostly context	Buy, configure

Capability	Usually core or context	Default posture
A proprietary routing or pricing algorithm	Can be core	Build a thin layer
A unique customer-facing service	Often core	Build
Integration between bought systems	Necessary glue	Build, often AI-assisted

The table is a starting point, not a verdict. What is context for one company can be core for another, and the point of the framework is to force the question rather than to answer it in advance. A logistics provider whose entire value proposition is a superior routing capability may rightly treat routing as core and build it. A retailer for whom routing is simply a cost to be minimized should buy it. The discipline is to make the classification objectively, resisting the natural pull to believe that everything the organization touches is special.

How the classification goes wrong

The core and context test is simple to state and surprisingly hard to apply, because the ways it fails are subtle and self-flattering. In practice, organizations misclassify capabilities in a handful of predictable patterns, and recognizing them in advance is most of the defense against them.

The first failure is confusing difficulty with differentiation. A capability can be genuinely hard to build and still be pure context. Tax calculation is difficult, deeply regulated, and utterly undifferentiating, which is precisely why almost no company builds its own. Difficulty is a reason to respect a problem, not a reason to own it. The question is never whether something is hard, but whether doing it better than a vendor would cause a single customer to choose you.

The second failure is confusing history with strategy. The phrase we have always built this here is a description of the past, not an argument about the future. Systems that were reasonably built in-house a decade ago, when the vendor market was thin, are frequently kept in-house long after mature alternatives have appeared, simply because they exist and someone is proud of them. Sunk cost masquerades as differentiation. A clean classification asks what you would do if you were starting today with no existing system to defend.

The third failure is letting the loudest internal customer define what is core. The operations leader who wants a bespoke tool, the executive with a favorite feature, the team that finds every packaged option beneath them, each can push an organization toward building something that serves an internal preference rather than an external advantage. Internal enthusiasm is not the same as market differentiation, and a capability that only its sponsor considers special is context wearing a disguise.

A disciplined classification therefore separates three questions that are easily blurred together. Is the capability necessary? Almost everything is. Is it difficult? Often. Does it differentiate us in the eyes of a customer who could choose a competitor instead? Rarely. Only the third question determines whether something is core, and only capabilities that survive it earn the cost and permanence of a build. Everything else, however necessary and however difficult, belongs in the column marked buy.

How core and context differ by industry

Because the classification is strategic rather than technical, the same capability lands in different columns depending on the business it serves, and it is worth walking through several industries to see the boundary move. In each, the vast majority of software is context to be bought; what changes is the identity of the thin core worth building.

In manufacturing, the enterprise resource planning system, the general ledger, the human-resources platform, and the standard modules of quality and maintenance management are context, supplied deeply by mature vendors and undifferentiating however essential. The core, where it exists, is usually a process capability: a proprietary production technique encoded in software, a specific scheduling or yield-optimization method a competitor cannot buy off the shelf, or the integration between shop-floor systems that reflects a particular way of running the plant. Build the method; buy the ledger. The failures in Section 13 are disproportionately manufacturers and distributors who inverted that rule and customized the ledger instead.

In retail and e-commerce, point-of-sale, merchandising, warehouse execution, and payments are context, and the history of retail technology failures is largely a history of companies treating one of them as special. The core is typically the customer-facing experience, the way shoppers discover, personalize, and move through a purchase, and sometimes a proprietary merchandising or pricing method. A retailer should buy the machinery of commerce and build only the storefront and the logic that makes its assortment or its prices distinctively its own.

In healthcare, the electronic health record, billing, and scheduling are heavily regulated context that essentially every provider buys, because the compliance and interoperability burden is enormous and shared cheaply across a vendor's base, exactly the compliance dividend of Section 11. The core, for a care provider, is rarely software at all; for a health-technology company, by contrast, the core may be a clinical algorithm or a device's embedded intelligence, which is differentiating and often, under the frameworks of Section 10, precisely the high-risk system that makes its builder a regulated provider. Here the build decision and the regulatory decision are the same decision, and must be made together.

In financial services, the core banking or trading platform, the payments rails, and regulatory reporting are context bought from specialists, because the cost of building and maintaining them to regulatory standard is prohibitive and best shared across an industry. The core is the proprietary model, the pricing engine, the risk model, the fraud or credit algorithm, that constitutes the firm's actual edge. Institutions that have tried to build their own core platforms have mostly repeated the failures of Section 13; those that buy the platform and build the model tend to prosper.

In logistics, the exception is most visible, which is why this guide has drawn so heavily on the sector. Warehouse and transportation management are context for most shippers and retailers, who should buy them without hesitation. But for a third-party logistics provider or a carrier whose entire value proposition is a superior routing, consolidation, or yield method, that specific capability is genuinely core, the worked example in Section 14 made exactly this case, and belongs on the build side, layered thinly on a bought operational core. Same capability; opposite classification; different business.

For a software or technology company, the boundary shifts furthest, because the product is software and its core is, by definition, built. Yet even here the discipline holds in the other direction: the differentiating product is built, and the enormous commodity remainder, the billing, the customer-relationship management, the analytics warehouse, the human-resources platform, is bought, precisely so that scarce engineering effort concentrates on the product rather than on reproducing infrastructure the market already supplies. The most capable builders in the world are also disciplined buyers; that is not a coincidence but the same judgment applied consistently.

Across all of them the shape is identical even as the contents change. A large commodity substrate is bought; a thin, business-specific core is built; and the hard work is not the buying or the building but the honest classification that assigns each capability to the right column. An organization that learns to make that classification well in its own industry has learned the most transferable skill in this entire guide, because the columns will keep shifting, with technology and with strategy, and the discipline of asking the differentiation question anew is what keeps the answer current.

04 The base rate: what happens when companies build

A decision to build is a bet, and it is worth knowing the odds before placing it. Decades of research into large software and IT projects paint a consistent and sobering picture, one that has barely improved even as tools and methods have advanced. The single most cited study is a 2012 analysis by McKinsey in cooperation with the University of Oxford, which examined more than 5,400 IT projects with initial budgets above fifteen million dollars. On average, the study found, these projects ran 45 percent over budget and 7 percent over time, while delivering 56 percent less value than had been predicted. Software projects in particular carried the highest risk of cost overruns. Most striking, roughly 17 percent of large projects became what the authors called black swans: overruns of 200 to 400 percent that could threaten the survival of the organization undertaking them.

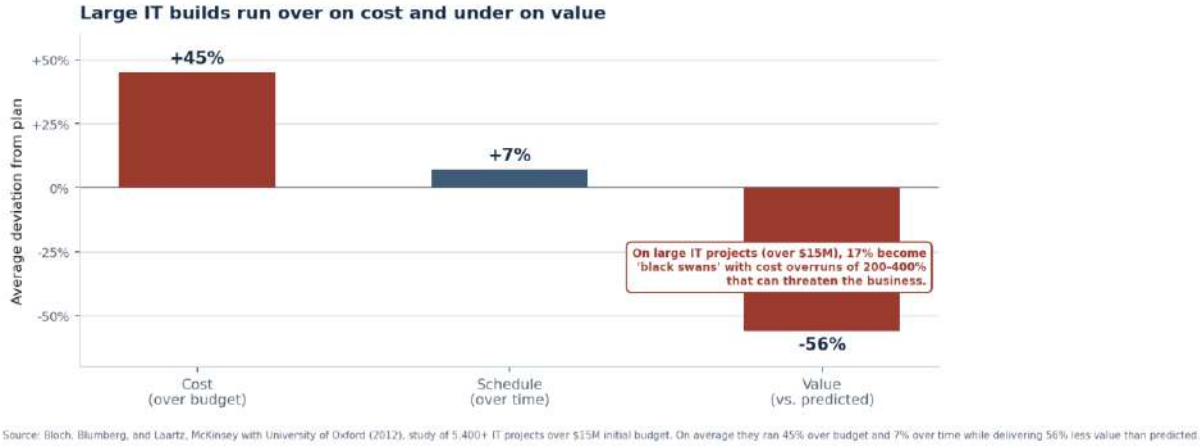

Figure 1. The average large IT project overruns on cost and time while delivering far less value than promised, and a meaningful minority overrun catastrophically.

The pattern is not confined to one study. For three decades the Standish Group has tracked software project outcomes in its CHAOS research, sorting projects into successful, challenged, meaning late, over budget, or short on scope, and failed, meaning cancelled or never used. The precise numbers move year to year, but the shape is remarkably stable: only around a third of projects succeed outright, roughly half are challenged, and the remainder fail. In other words, about two out of three software projects end in partial or total disappointment, a figure that has scarcely improved across the entire history of the research.

Figure 2. Standish CHAOS outcomes. Only about a third of software projects fully succeed; most are late, over budget, or under scope, and roughly a fifth fail outright.

An honest word about the data

These figures deserve a caveat, because intellectual honesty requires it. The Standish CHAOS data is a proprietary survey whose underlying data is not published, and its definitions have been criticized by academic researchers who argue they overstate failure rates. The McKinsey and Oxford study is a decade old and focused on the largest projects. Neither should be read as a precise law of nature. What they establish, taken together and alongside the lived experience of anyone who has run enterprise software programs, is directional and robust: building large custom systems is hard, the failure rate is high, and the tendency to overrun on cost and underdeliver on value is persistent. A leader betting on a build should assume a meaningful probability of landing in the disappointing majority, and should ask what, specifically, will make this project the exception.

Why builds go wrong

The reasons builds fail are well understood and have little to do with the availability of coding tools. Requirements are misunderstood or change midstream. The complexity of integrating with existing systems is underestimated. Data quality problems surface late. Testing is compressed to hit a deadline. The people who understood the system leave. Scope expands quietly until the project collapses under its own weight. None of these failure modes is a code-writing problem, which is the central reason to be skeptical that a faster way to write code will move the base rate. The failures happen in the space around the code, in understanding, design, integration, data, testing, and organizational change, and that space is exactly where the next two sections show AI helps least.

The anatomy of a runaway build

Base rates describe what happens on average, but they are easier to trust when you can see the mechanism that produces them. Runaway builds are rarely the result of a single catastrophic decision. They are the accumulation of small, reasonable choices that compound, and the pattern repeats across industries with enough regularity to be worth naming in advance.

It usually begins with a genuine and underestimated problem. The initial scope is drawn around the visible requirement, the part everyone can see and agree on, and the estimate is built on that scope. What the estimate omits is the surrounding work that only becomes visible once construction starts: the edge cases in the data, the exception that finance insists on, the report that a regulator requires, the third system that turns out to hold the authoritative record. Each of these is discovered rather than planned, and each arrives with its own small overrun.

The second stage is the quiet expansion of scope. Because a custom system can do anything, everyone who hears about it asks for one more thing, and because saying yes is easier than saying no, the system grows. Every addition seems affordable in isolation. Collectively they reshape the project into something far larger than what was funded, and the timeline stretches to accommodate work that was never in the original case. This is why large technology projects overrun so consistently that the pattern has a name;

the research on major IT programs finds that they run over budget far more often than not, and that a meaningful share become true outliers that consume many times their intended cost.

The third stage is the erosion of the team that holds the knowledge. Long builds outlast the tenure of the people who start them. The engineer who understood the original design leaves, the context that lived only in their head leaves with them, and their successors move more slowly because they are reconstructing intent as much as writing code. Every departure raises the cost of the next change and lowers the confidence of the people making it. What began as a project becomes an obligation that the organization services indefinitely.

The final stage is the point of no return, where the system is too embedded to abandon and too fragile to trust. The business now depends on something that only a shrinking group understands, that resists change, and that quietly consumes the engineering capacity that was meant to be spent on advantage. The tragedy is that each individual decision along the way was defensible. It is only in aggregate, and only against the base rate, that the pattern reveals itself as the ordinary way that builds go wrong rather than an unlucky exception.

What would make a build the exception

If the base rate is discouraging, the useful response is not fatalism but specificity: to ask what, concretely, would place a given build among the minority that succeed. The projects that beat the odds tend to share a recognizable set of traits, and their absence is as diagnostic as their presence.

- **A narrow, stable scope.** The successful builds are small and sharply bounded, and they resist the scope creep that sinks the rest. If the scope is broad or still moving, that is the base rate reasserting itself.
- **A genuine differentiator.** The build produces something the organization actually competes on, not a commodity it could have bought, which is what justifies the effort and sustains the commitment when the work gets hard.
- **A named, funded owner.** There is a person or team accountable for the system's whole life, not a project sponsor who disbands at launch.
- **Clean interfaces to a bought core.** The build is a thin layer on top of purchased infrastructure, not a monolith that must do everything itself.
- **Disciplined implementation.** The project phases its rollout, tests against real data, and respects the disciplines of Section 16 rather than compressing them to hit a date.

A build that can honestly claim all five has a real chance of being the exception. A build that can claim none of them is simply the base rate wearing optimism, and the most useful thing a leader can do with the base rate is not to fear it but to ask, clause by clause, which of these traits the proposal in front of them actually has.

05 The iceberg: the true lifetime cost of building

The most consequential misunderstanding in build-versus-buy is about cost, and it is a misunderstanding of shape. Organizations tend to picture the cost of building as the cost of the build: the months of engineering to get the first version working. That initial development is real, but it is the visible tip of a much larger mass. Across decades of software engineering research, the consistent finding is that maintenance, everything that happens after the first version ships, consumes the majority of a system's lifetime cost, and often the large majority. Estimates vary by methodology and context, but the range that recurs is 60 to 90 percent of total cost of ownership. For complex enterprise software running on-premises, the figure sits toward the top of that range.

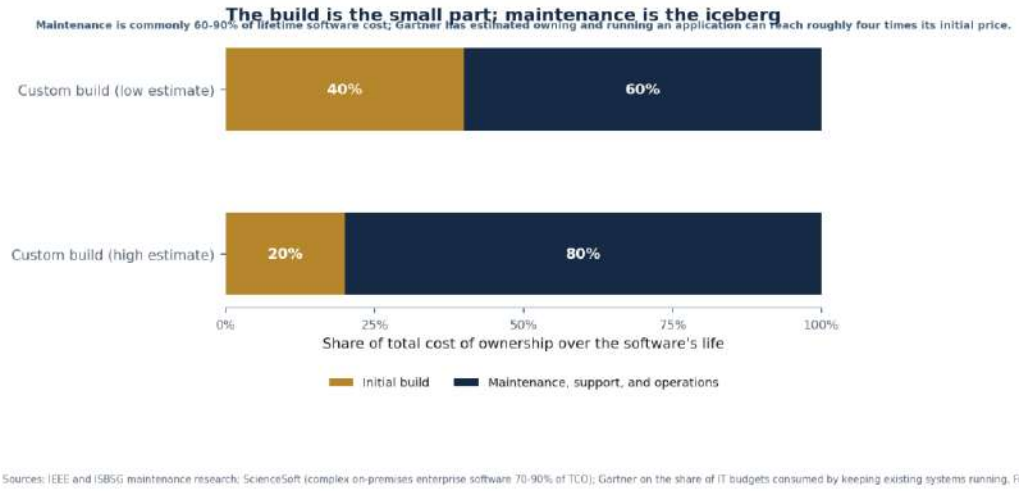

Figure 3. The initial build is the small, visible part. Maintenance, support, and operations dominate the lifetime cost of custom software.

Put in the plainest terms, if building the first version of a system costs a certain amount, owning it over its life will typically cost several times more. Gartner has estimated that owning and running an application over its lifetime can reach roughly four times its initial price, and that organizations commonly spend the majority of their IT budgets simply keeping existing systems running rather than building anything new. The initial build, in other words, is a down payment on a long and growing obligation. This is true of all software, bought or built, but with a crucial difference: when you buy, the vendor carries most of that obligation, spreading it across every customer. When you build, you carry all of it alone.

The hidden mass beneath the surface

What exactly makes up the submerged cost? It is the accumulation of everything a working system requires once real users, real data, and real adversaries encounter it:

- **Integration and compatibility.** Every system you connect to is a system that can change, and each change can break your integration. The connective tissue between systems is a permanent maintenance cost, and in enterprise environments it is frequently the largest one.

- **Security and compliance.** New vulnerabilities appear constantly, dependencies must be patched, and regulatory requirements evolve. Securing a system is not a one-time task but a continuous obligation that never ends while the system runs.
- **Bug fixes and edge cases.** Real use surfaces conditions no one anticipated, and each must be diagnosed and fixed, often under pressure, often years after the original author has moved on.
- **Infrastructure and operations.** Running software means monitoring it, scaling it, recovering it when it fails, and paying for the compute and storage beneath it, indefinitely.
- **Documentation and knowledge.** Systems must be understood by people who did not build them, and the erosion of that understanding over time, as staff turn over, is a slow but serious cost.
- **Technical debt.** Shortcuts taken to ship faster accumulate as debt that must eventually be repaid, in the form of slower future changes and higher defect rates, with interest.
- **Concentration and key-person risk.** Custom systems tend to depend on a small number of people who truly understand them. When those people leave, the organization is exposed, and the cost of that exposure is easy to ignore until it is realized.

The opportunity cost that dwarfs the rest

Beneath even the maintenance iceberg lies a deeper cost, and it is strategic rather than financial. Every capable engineer assigned to build and then maintain a commodity capability is an engineer not working on the thing that actually differentiates the business. This is the opportunity cost, and for organizations whose advantage depends on technology, it is often the cost that matters most. A team that spends its time keeping a home-built warehouse system running is a team that is not improving the proprietary capability that sets the company apart. Buying the commodity is not only frequently cheaper in direct terms. It frees the scarcest resource an organization has, the attention of its best builders, for the work that only they can do and that only the company can claim as its own.

Putting numbers on the hidden mass

The claim that maintenance dominates the lifetime cost of software is easy to assert and easy to wave away. It is more useful to see where the mass actually sits, because the categories beneath the waterline are not exotic. They are the ordinary, recurring obligations that every operated system carries, and each one is a line the initial build estimate almost never includes.

The largest category is adaptive maintenance: keeping the system working as the world around it changes. Operating systems and libraries are deprecated, browsers and devices evolve, the systems you integrate with publish new versions and retire old ones, and tax rules, carrier formats, and compliance requirements shift every year. None of this adds a single feature. All of it is mandatory, and it never stops. A system that is merely kept current, with no new capability at all, still consumes real engineering time every quarter simply to stand still.

The second category is corrective maintenance, the diagnosis and repair of defects that surface only in production, under real data and real load. These are the failures that no test anticipated, discovered at inconvenient times, and they carry a hidden multiplier: the cost of the incident itself, the cost of the fix,

and the cost of the confidence lost while the system was unreliable. Corrective work is unpredictable by nature, which makes it impossible to schedule around and expensive to staff for.

The third category is perfective maintenance, the changes the business asks for after launch because requirements are never truly frozen. A new customer segment, a new channel, a new pricing model, a new report, each is a modification to a system that must keep running while it is modified. This is the category that leaders imagine when they picture owning software, and it is only a fraction of the total. The adaptive and corrective work beneath it is the part that surprises.

Around all three sits the operational overhead that rarely appears in any estimate: monitoring and alerting, hosting and infrastructure, backups and disaster recovery, access control and audits, on-call rotations and the documentation that keeps them possible. This is the machinery of running a system as opposed to writing one, and it is a fixed cost of ownership that persists for as long as the system lives. When practitioners say that the build is the small, visible tip of the iceberg, this is the mass they mean, and it is why a system that was cheap to write can be ruinously expensive to keep.

The layers of the iceberg, one by one

It helps to name the hidden layers individually, because a cost you can name is a cost you can budget, and the surprise of a build is rarely one large omission but the accumulation of several predictable ones.

- **Maintenance.** The largest and most certain layer, at roughly fifteen to twenty-five percent of the build cost every year for the life of the system; over a typical lifespan it outweighs the build that produced it, and it does not taper.
- **Security.** A permanent, continuous tax, patching, disclosures, and certifications, that a vendor spreads across its customer base and a builder carries alone, made heavier by the evidence in Section 9 that generated code raises the security burden.
- **Integration.** The connective tissue between systems, routinely underestimated, sometimes larger than the system itself, and re-incurred every time any connected system changes.
- **Infrastructure and operations.** Hosting, monitoring, backups, scaling, and on-call response, invisible in an estimate and unavoidable in a life, included in a bought service's price and self-provided for a build.
- **Talent and knowledge.** The standing engineering capacity a build permanently claims, plus the risk that the people who understand it leave and take the design intent with them.
- **Technical debt.** The layer that compounds, as every deadline shortcut accrues interest paid in slower, riskier future change, faster now that code can be generated faster than it can be understood.
- **Opportunity cost.** Beneath them all, the differentiating work not done because engineers are maintaining a commodity, the layer that appears on no invoice and dwarfs the rest.

Summed, these layers are why the true cost of a build so reliably exceeds the estimate, and why the estimate is not merely imprecise but biased downward: it measures the tip and ignores the mass. Price the layers, and the surprise disappears.

06 A worked example: what a custom build really costs

Abstractions about lifetime cost become persuasive only when they are made concrete, so consider a stylized example. A regional distributor needs a warehouse execution capability. Its engineering team, encouraged by how quickly modern tools let them build, proposes to construct one in-house rather than license a commercial product. The team estimates the build at around nine hundred thousand dollars, and compares that favorably to a vendor whose software, they note, will cost well over a million dollars across the first few years. On that comparison, building looks cheaper, and the decision seems obvious. The comparison is also wrong, because it measures the wrong thing: the cost of the initial build against the multi-year cost of buying. The honest comparison puts the full lifetime cost of each option side by side, and when it does, the picture inverts.

The figures that follow are illustrative and directional, chosen to show the shape of the problem rather than to serve as a benchmark. Real costs vary widely by scope, organization, and vendor, and the point of the exercise is the pattern, not the precise numbers. What the pattern reveals is that the visible build cost is a fraction of the total, and that the parts left out of the naive comparison, maintenance, integration, security, and operations, are what make building the more expensive path over time.

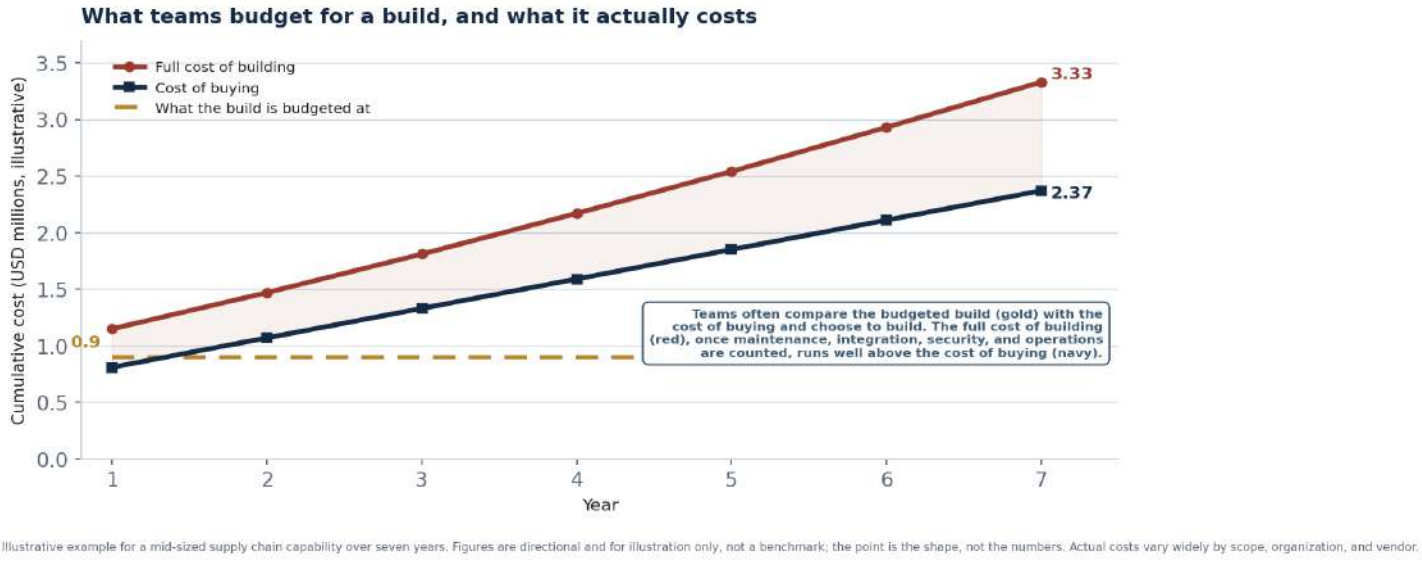

Figure 4. An illustrative seven-year comparison. Teams anchor on the budgeted build, but the full cost of building, once maintenance and operations are counted, runs well above the cost of buying.

The two columns, fully totaled

Set out over a seven-year horizon, the reasonable life of such a system, the two options compare as follows. The build begins with the nine hundred thousand dollars of initial development, but that is only the first line. Integrating the new system with the distributor's existing enterprise resource planning, carriers, and devices adds a substantial one-time cost. Then, every year for seven years, the system must be maintained, at a rate that industry experience puts at roughly a fifth of the build cost annually, and it must be secured, operated, and hosted. Each of those is a recurring line that the initial estimate omitted

entirely. The buy option, by contrast, carries an annual subscription, a one-time implementation and integration cost, a configuration and adoption cost, and a modest annual administration cost, and little else, because the vendor absorbs maintenance, security patching, and platform evolution across its whole customer base.

Cost line (illustrative, 7-year)	Build	Buy
Initial development	$900K	n/a
Implementation and integration (one-time)	$250K	$400K
Configuration and adoption (one-time)	included	$150K
License or subscription (7 years)	n/a	$1,540K
Maintenance (7 years, ~20% of build/year)	$1,320K	in subscription
Security and compliance (7 years)	$420K	in subscription
Infrastructure and operations (7 years)	$560K	in subscription
Internal administration (7 years)	in maintenance	$280K
Approximate seven-year total	**~$3.45M**	**~$2.37M**

On these illustrative numbers, the option that looked like a nine-hundred-thousand-dollar build is a three-and-a-half-million-dollar commitment over its life, roughly half again as expensive as buying. And the table still flatters the build in two ways. It uses the team's own estimate as though it were reliable, when the base rates from Section 4 suggest a meaningful chance of a substantial overrun. And it ignores opportunity cost: the engineers building and then maintaining this commodity capability are not building the distributor's actual source of advantage. Add a risk-adjusted contingency for overrun and a realistic charge for the diverted talent, and the gap widens further.

What the example is, and is not

This is not a claim that building always costs more, nor a precise model anyone should apply to their own situation. It is a demonstration of a reasoning error, the error of comparing a build's upfront cost to a purchase's multi-year cost, which flatters building and recurs in real decisions. The corrective is simple to state and harder to practice: when weighing a build, total its cost the way you would total a purchase, across the whole life of the system, with every recurring line included, and only then compare. The tools that write code faster do not change any of the lines in this table except, at most, a portion of the first one. The maintenance, the integration, the security, and the operations, the lines that make building expensive, are untouched, which is the theme the next sections develop.

Sensitivity: how the math shifts

A single worked example is a snapshot, and its value lies less in the exact figures than in the levers that move them. The honest way to use it is to ask which assumptions, if they changed, would change the conclusion, because that is where a real decision should focus its scrutiny. Three levers dominate the outcome, and each can be estimated before a line of code is written.

The first lever is the fully loaded cost of the team and the length of the commitment. Build estimates almost always quote salaries, but the cost that matters is fully loaded: benefits, recruiting, management overhead, tooling, and the fraction of senior time spent on hiring and review rather than delivery. More importantly, the commitment does not end at launch. A system that requires two engineers to build often requires one to maintain in perpetuity, and that perpetual half-team is a standing cost that continues for as long as the system runs. Multiply a conservative annual maintenance figure by a realistic system lifetime and the maintenance column routinely dwarfs the build.

The second lever is the maturity and pricing power of the vendor alternative. The case to buy is strongest where the market is deep, competitive, and commoditized, because that competition holds prices down and pushes capability up without any effort on your part. It is weakest where a single vendor dominates a narrow niche and prices accordingly, or where no credible packaged option exists at all. Before accepting a build estimate, the discipline is to price the genuine alternative, including implementation and several years of subscription, so that the comparison is lifetime against lifetime rather than build against nothing.

The third lever is the probability and cost of failure, which build estimates almost always assume away. The base rates are unambiguous: a substantial share of custom builds are late, over budget, or delivered with far less value than promised, and a smaller but real share fail outright. A defensible comparison weights the build column by that probability. Even a modest expected-failure adjustment, applied to a project's full cost, shifts the arithmetic sharply, because the buy option carries a fraction of that risk. The vendor has already absorbed the cost of getting it wrong across hundreds of customers, and you inherit the survivor.

When all three levers point the same way, a deep vendor market, a long expected life, and an ordinary risk profile, the example is not close, and no reasonable change to the inputs rescues the build. When they point in different directions, a thin market, a short horizon, a genuinely differentiating capability, the case for building strengthens, and the example is doing exactly what it should: not delivering a verdict, but showing you which few facts your decision actually turns on.

Three scenarios: small, mid-market, and enterprise

The single worked example above is a mid-sized case. It is worth seeing the same arithmetic at three different scales, because the levers that decide it behave in instructive ways as the system grows. The figures below are, like the earlier table, illustrative and directional rather than benchmarks, chosen to show the shape of the relationship as scope increases.

Scenario (illustrative, 7-year, all-in)	Naive build estimate	True cost to build	Cost to buy	Lower lifetime cost
Small regional distributor — warehouse execution	$300K	~$1.6M	~$0.9M	**Buy**

Scenario (illustrative, 7-year, all-in)	Naive build estimate	True cost to build	Cost to buy	Lower lifetime cost
Mid-market third-party logistics — TMS and WMS	$900K	~$3.5M	~$2.4M	**Buy**
Global retailer — end-to-end planning suite	$6M	~$22M	~$14M	**Buy**
A genuinely differentiating edge capability	$500K	~$1.4M	no real fit	**Build**

An illustrative all-in comparison across three scales, plus the exception. The true cost to build runs well above both the naive estimate and the cost to buy at every scale; only where no product fits and the capability differentiates does building win.

Three things stand out when the comparison is run at different scales. The first is that the ratio is stubborn. Across the small, mid-market, and enterprise cases the true cost to build lands somewhere between roughly one and a half and two times the naive estimate once maintenance, integration, security, and operations are counted, and it runs comfortably above the cost to buy. The reason is structural: the lines the naive estimate omits scale with the system, so making the build bigger does not make the omission smaller. It makes it larger in absolute terms.

The second is that the absolute stakes grow faster than the percentages suggest. A fifty-percent overrun on a three-hundred-thousand-dollar build is an annoyance; the same proportional overrun on a six-million-dollar enterprise program is millions of dollars and, on the base rates in Section 4, sits inside the range where roughly one project in six becomes a black swan that threatens far more than the IT budget. Scale does not merely multiply the cost of building. It multiplies the cost of being wrong.

The third is the exception in the last row, and it is the important one. When no packaged product genuinely fits, and the capability is a real source of competitive advantage rather than a commodity, the buy column has no entry to compare against, and building becomes not only defensible but correct. That is precisely the edge the third path reserves for building, and it is a narrow target. For every capability that belongs in that last row, most organizations will find a dozen that belong in the first three. And in none of the rows does AI change the arithmetic in the builder's favor, because it touches, at most, the naive-estimate column and leaves every recurring line that makes building expensive exactly where it was.

Reading a build estimate: what to add back

Because the naive estimate is structurally biased downward, a useful discipline is to take any build estimate you are handed and mechanically add back the lines it almost certainly omits. The exercise turns an optimistic number into a defensible one, and it can be done on the back of an envelope.

Start with the build figure as quoted, the cost to design and write the software to a working state. Then add the first correction, maintenance, at roughly fifteen to twenty-five percent of the build cost per year for the expected life of the system; over a typical horizon this alone often exceeds the original build. Add

the second, integration with the systems the new one must connect to, which is routinely underestimated and sometimes rivals the build itself. Add the third, security and compliance, as a continuing annual cost rather than a one-time feature, and higher now than a few years ago because of the regulatory obligations described in Section 10.

Add the fourth, infrastructure and operations, the hosting, monitoring, backups, and on-call response a bought service would have folded into its price. Add the fifth, the standing engineering capacity the system will claim every year, valued at what those engineers would otherwise have produced. And add the sixth and largest, the opportunity cost: the differentiating work not done because scarce engineers are maintaining a commodity. Only the first of these, the build figure itself, is the one AI plausibly reduces, and it is the smallest of the six over the life of the system.

Run this addition and the pattern of Section 6's scenarios reappears on your own numbers: the true cost lands well above both the naive estimate and the cost to buy, and it does so not because of any single omission but because the estimate measures the one line that is cheap to quote and ignores the several that are expensive to live with. An estimate that has had these lines added back is not pessimistic; it is merely complete, and a decision made on the complete number is the only kind worth defending.

07 What AI actually changed on the build side

To weigh AI fairly, it is necessary to give it full credit for what it truly does, before turning to what it does not. AI coding assistants are useful, and the evidence for their usefulness on the right kind of task is real. The most-cited controlled study, conducted by researchers with GitHub in 2023, asked ninety-five professional developers to build an HTTP server in JavaScript, with half using an AI assistant and half not. The group with the assistant completed the task about 56 percent faster, seventy-one minutes against a hundred and sixty-one. The gains were largest for less experienced developers. It is a striking result, and it is a fair representation of AI at its best: a well-bounded, greenfield task in a common language, exactly the conditions under which the tools shine.

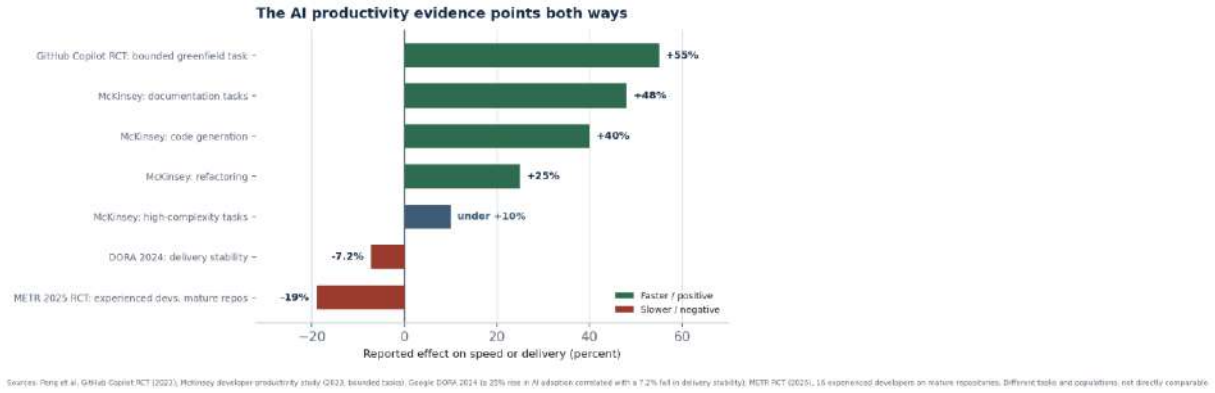

Figure 5. The productivity evidence is truly mixed. AI speeds bounded, greenfield tasks substantially, but gains shrink on complex work and reverse in some rigorous trials of experienced developers.

A widely-read 2023 McKinsey study of around forty developers found a similar pattern with an important qualification. On bounded tasks, the tools cut the time to complete documentation by 45 to 50 percent, to write new code by 35 to 45 percent, and to refactor by 20 to 30 percent. But the same study found the gains fell to under 10 percent on tasks of high complexity, and that developers unfamiliar with the relevant framework saw little benefit. The study also noted, candidly, that the assistant sometimes produced incorrect recommendations and even introduced errors, so that the time saved depended on the developer actively reviewing and correcting the output rather than accepting it.

Where AI truly helps

The shape of the benefit is consistent across the credible research. AI is strongest at generating boilerplate, the repetitive scaffolding that every project needs and no one enjoys writing. It is good at code completion, at drafting tests, at producing first-pass documentation, at explaining unfamiliar code, and at helping a developer work in a language or framework they do not know well. It accelerates prototyping, letting an idea become a clickable artifact in a fraction of the usual time. For a well-specified, self-contained problem, it can produce a working first draft remarkably fast. These are real capabilities, and an organization that ignores them is leaving value on the table.

The gap between the demonstration and the product

But notice what unites the tasks where AI excels: they are bounded, they are common, and they are the beginning of the work rather than the end. The moment the task becomes producing not a draft but a production system, one that must be correct, secure, integrated, performant, and maintainable, the contribution of the tool thins, and the contribution of human judgment grows. This is the gap between the demonstration and the product, and it is the crux of the entire build-versus-buy question in the AI era. A prototype is perhaps a fifth of a production application. The other four fifths, the error handling, the edge cases, the security hardening, the integration, the performance work, the monitoring, the long tail of correctness, is the part AI helps with least and the part that determines whether a system is a genuine asset or an expensive liability. The next section examines why that gap has proven so stubborn.

What the acceleration is worth, and what it is not

It is important to be generous and specific about what artificial intelligence genuinely changed, because overstating the skeptical case is as much an error as believing the easy-button story. AI did not make software cheaper to own, but it did make parts of the work faster, and knowing exactly which parts is what separates a useful tool from a dangerous assumption.

The clearest gains fall on bounded, well-specified, and low-context tasks. Writing a function whose behavior is fully described, translating code from one language to another, generating tests against a clear contract, drafting boilerplate, scaffolding a new component, and explaining an unfamiliar piece of code are all work that assistants now do quickly and often well. These are real savings, and for an individual developer on the right task they can be dramatic. A controlled study of a bounded implementation task famously found a large speed-up, and that result is credible precisely because the task was bounded.

The difficulty is that this kind of work, the pure production of code against a clear specification, is a minority of the total effort in building and running a system. The larger share is spent deciding what to build, understanding the domain, negotiating requirements with people who disagree, designing an architecture that will survive change, integrating with systems that were not designed to cooperate, getting the data right, and then maintaining the result for years. AI accelerates the coding slice and leaves the surrounding work largely untouched, which is why an eighty percent reduction in the cost of writing code can produce a far smaller reduction in the cost of delivering a working, owned system.

There is also a distinction between the acceleration a skilled engineer gets and the illusion of acceleration a novice gets. In expert hands, an assistant is a fast, tireless pair programmer whose output the expert can judge and correct. In inexpert hands, the same tool produces plausible code that its user cannot fully evaluate, which feels like speed and is in fact the quiet accumulation of risk. The value of the acceleration therefore depends on the judgment of the person receiving it, and judgment is exactly the thing the tool does not supply.

The reasonable conclusion is that AI has genuinely compressed one segment of the work, that the compression is largest where a task is small and well-defined, and that it shrinks as the work grows more ambiguous, more integrated, and more consequential. That is a meaningful improvement worth

adopting deliberately. It is not the same as making it cheap to build and own a system, and treating the first as if it were the second is the specific mistake this guide exists to prevent.

A taxonomy of the tools, and where each helps

It helps to distinguish the categories of AI development tools, because they differ sharply in how much they help and how much risk they carry, and lumping them together as “AI coding” obscures the distinction that matters for a build decision.

The most mature category is inline autocomplete, the assistant that finishes the line or the block you have begun. Its gains are real and its risk low, because the developer stays in control, reviewing each suggestion in the context of code they are actively writing. This is AI at its best on the build side: an accelerator of typing and recall, not a replacement for judgment.

Conversational assistants, which answer questions and generate larger fragments on request, help most with bounded, well-specified tasks, writing a function to a clear specification, explaining unfamiliar code, drafting a test, and help least with the open-ended work of design and integration where the specification is itself the hard part. Their risk rises with the size of what they generate, because a large block accepted without full understanding is a large block no one truly owns.

Autonomous agents, which plan and execute multi-step tasks with limited supervision, are the newest and the most oversold category. They are genuinely useful for well-bounded, repetitive work with clear success criteria, and genuinely dangerous when pointed at consequential, under-specified work, because they can produce a great deal of plausible output quickly, and the effort of verifying it can exceed the effort the agent saved. The batch-size problem of Section 9 is an agent problem above all.

Two categories deserve special mention because they push in the safer direction: test generation and code review. Tools that help write tests or flag issues in a change add scrutiny rather than volume, and scrutiny is exactly what generated code most needs. An organization using AI heavily on the build side should lean into these categories deliberately, because they shore up the absorptive capacity that determines whether the other categories help or harm.

The practical lesson is that “we use AI to build” is not one decision but several, and the safest posture uses the low-risk categories liberally, the high-risk categories carefully and only at the bounded edge, and the scrutiny-adding categories as a deliberate counterweight to the volume the others produce. Pointed at the commodity core, none of them changes the case against building it. Pointed at the differentiating edge, with the guardrails of Appendix B, they are where AI on the build side genuinely earns its place.

08 Vibe coding: the seduction of the working demo

No phenomenon captures both the promise and the peril of AI-assisted building better than the practice that came to be called vibe coding. The term, popularized in early 2025, describes a way of working in which a person describes what they want in plain language, accepts what the AI produces, and iterates by feel, without closely reading or fully understanding the code that results. For the right purpose, it is a genuine breakthrough. A person with an idea and little or no programming background can now produce something that runs, and can do so in an afternoon. That is a real and, in its place, valuable capability, and it deserves to be recognized as such before its limits are examined.

The limits, however, are precisely where the build-versus-buy decision turns, because vibe coding is a prototyping technique that is easily mistaken for a production method. The distinction matters enormously. A prototype exists to validate an idea, to answer the question of whether something is worth doing. It can be discarded the moment that question is answered, and its flaws cost nothing because nothing depends on it. A production system is the opposite. It is depended upon, and everything that vibe coding skips, the reading, the understanding, the hardening, the testing, is exactly what dependence requires. Vibe coding is excellent for the first and dangerous for the second, and the danger is amplified because the two can look identical on screen.

Why the demo is so convincing

The seduction is that a vibe-coded application works, visibly, in the demonstration. It has a polished interface, it responds to input, and it does the thing it was asked to do. To a leader watching, it appears finished, and the natural conclusion is that the hard work is done and only minor cleanup remains. This impression is the trap. The demonstration exercises the expected path, the case the builder had in mind, and on that path the application performs. What the demonstration does not exercise is everything else: the malformed input, the simultaneous users, the failure of a dependency, the security probe, the volume of real data, the integration with the systems the business already runs. The gap between what the demonstration shows and what production demands is not a matter of cleanup. It is the majority of the work, and it is invisible precisely because a demonstration is designed not to reveal it.

Independent testing bears this out with unusual consistency. Reviewers who take the leading AI coding tools and attempt to carry a realistic project all the way to completion, rather than stopping at a snippet, report that the tools accelerate the early scaffolding impressively and then stall at the same juncture: the integration of the parts, the handling of the unexpected, and the final push to something robust enough to depend on. The tools do not announce their limit. They produce something that looks complete and is not, which is a more insidious failure than an obvious one, because it invites a team to declare the work finished at the moment the truly hard part begins.

The debt that vibe coding hides

A further problem lurks beneath the surface of a vibe-coded system, and it is a maintenance problem. Because the code was accepted without being closely read or understood, no one on the team may fully comprehend how it works. This is tolerable, even irrelevant, for a throwaway prototype. It is a serious

liability for a system the business relies on, because the day it breaks, and every system eventually breaks, someone must diagnose and fix code that no human ever truly understood, that may be duplicated and inconsistent in the ways the earlier evidence described, and that may harbor the security weaknesses that studies find in a large share of AI-generated code. Vibe coding does not remove the maintenance burden that Section 5 described. It defers it, and worse, it hands the eventual maintainer a system without the understanding that maintenance requires. The speed felt at the outset is borrowed against a cost paid later, by whoever must keep the thing alive.

Where vibe coding belongs

None of this is an argument against vibe coding in its proper place. That place is real and worth naming: rapid prototypes to test an idea, throwaway tools that will not be depended upon, personal or internal utilities where the stakes of failure are low, and exploratory work meant to inform a decision rather than to run a business. In those contexts the speed is a gift and the absence of hardening does not matter, because nothing important rests on the result. The error is not using vibe coding. The error is promoting a vibe-coded prototype into a production system without doing the eighty percent of work the demonstration hid, and doing so on the mistaken belief that because it looked finished, it was. The discipline is to hold the line between the two: to use AI freely to explore and to prototype, and to insist that anything the business will depend on cross the same bar of understanding, testing, security, and maintainability that it always had to, whether a human or a machine wrote the first draft.

Where vibe coding fits, and where it does not

- **Fits: prototypes and throwaways.** Validating an idea, building a tool no one will depend on, or exploring a design. The speed is a real gift and the missing hardening does not matter.
- **Fits: low-stakes internal utilities.** Small helpers where failure is cheap and the system can be rewritten or abandoned without consequence.
- **Does not fit: anything the business depends on.** Systems that must be correct, secure, integrated, and maintained cannot skip the reading, testing, and hardening that vibe coding omits.
- **The trap: the demonstration that looks done.** A vibe-coded prototype exercises the expected path and appears finished; promoting it to production without the hidden eighty percent of work is the characteristic mistake.

A field guide to where it breaks

The working demo is persuasive because it succeeds at exactly the things that are easy and stays silent about the things that are hard. It is worth walking through the specific places where a vibe-coded prototype tends to fail when it is asked to become a product, because the failures are consistent enough to serve as a checklist for anyone tempted to ship the demo.

It breaks first at the edges of the data. A demonstration runs on clean, well-formed, cooperative inputs, because that is what the person building it feeds it. Production runs on the real world: missing fields,

duplicate records, inconsistent formats, values that violate assumptions the code never knew it was making. The happy path was the whole demo; the unhappy paths are most of the actual work, and they are invisible until real data arrives.

It breaks next at scale and concurrency. Code that works for one user and a small dataset often behaves very differently when many users hit it at once and the data grows by orders of magnitude. Queries that were instant become slow, assumptions about ordering and timing that held by luck begin to fail intermittently, and the resource usage that was trivial in the demo becomes a cost and a bottleneck. None of this is visible in the moment the demo impresses the room.

It breaks at the boundaries with other systems. A prototype typically stands alone or talks to a single friendly interface. A product must authenticate against the identity system, respect the permissions model, reconcile with the systems of record, survive the outages and rate limits and version changes of everything it depends on, and do so without losing or corrupting data when something upstream misbehaves. Integration is where much of the real engineering lives, and it is precisely what a standalone demo omits.

Finally, it breaks over time, because no one can safely change it. Code generated quickly and accepted without full understanding tends to lack the structure, tests, and documentation that make future change safe. The first modification reveals that no one is quite sure how it works, the second introduces a regression, and the team slows to a crawl or freezes the system in place. The demo bought a fast start and mortgaged the entire future of the code, which is the debt the next section of any honest accounting has to record.

The handoff no one budgets for

The most expensive moment in the life of a vibe-coded system is the handoff, when the prototype that one person conjured must become a product that a team can operate, extend, and trust. It is a moment the demonstration never shows and the estimate never includes.

A system generated quickly and understood by no one is, from the perspective of the people who must now own it, indistinguishable from a system inherited from a departed employee: it works, mostly, until it does not, and when it does not, no one can say why. The work of making it maintainable, understanding what it does, documenting it, testing it, hardening it, securing it, is the seventy percent the demonstration skipped, and it does not become cheaper because the first thirty percent was fast. Often it becomes more expensive, because reverse-engineering intent from generated code is harder than writing code whose intent you held from the start.

This is why vibe coding belongs at the prototype and never at the product, and why the boundary between the two must be guarded deliberately. A prototype's purpose is to be thrown away, to test an idea or show a stakeholder what something could look like, and its value lies entirely in the speed with which it can be produced and discarded. The failure mode is the prototype that is not discarded, the demonstration stakeholders liked so much that it was pushed into production, inheriting all the debt the speed concealed. The most disciplined use of vibe coding treats the working demo as an argument, not an asset: proof that an idea is worth doing properly, not the thing itself.

For a build decision this sharpens the earlier warning. The ease of producing a working demonstration lowers the apparent cost of starting a build precisely by hiding the handoff, and the handoff is where the real cost lives. A leader shown a compelling demonstration should ask not how quickly it was built but what it would cost to own, and should treat the gap between those two numbers as the true price of the decision.

09 Why AI is not an easy button

If AI accelerated only the easy part of building, that would be a manageable limitation. The deeper problem is that the difficulty does not disappear when the code is written faster. It moves. The hard work shifts from producing code to verifying it, from writing systems to understanding systems written partly by a machine, and from the first eighty percent to the last twenty. This section examines the specific, documented reasons that AI-assisted building has proven far harder to carry to production than the demonstrations suggest, drawing on the most rigorous evidence available and flagging where that evidence is early or contested.

The seventy percent problem

The most useful way to describe the limitation comes from the engineer and writer Addy Osmani, who named it the seventy percent problem. AI, he observed, can get a capable person, and even a non-programmer, most of the way to a working result with surprising speed. The first seventy percent arrives almost for free. The remaining thirty percent, the part that makes software actually work in the real world, behaves differently. Each attempt to fix a problem introduces another. Progress slows to a crawl precisely as the finish line comes into view, because the final portion is where the genuine difficulty of software has always lived: the edge cases, the integration points, the security hardening, the performance under load, the correctness guarantees. AI is a brilliant assistant for the first seventy percent and a much weaker one for the last thirty, and it is the last thirty that separates a demonstration from a system a business can depend on.

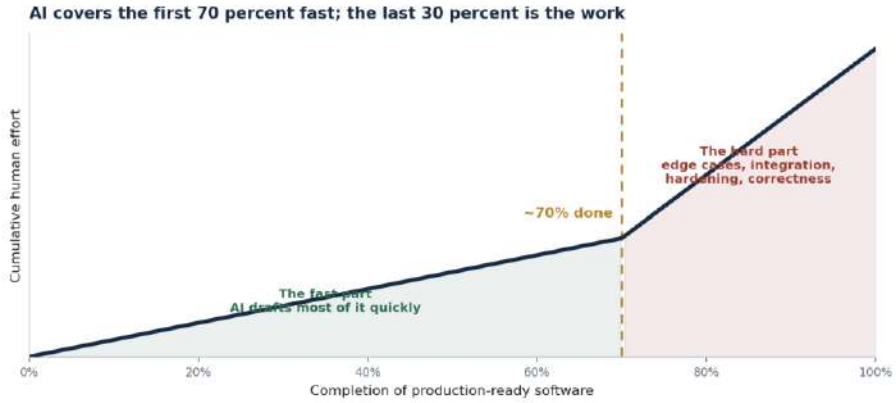

Figure 6. The characteristic shape of AI-assisted building. The initial draft comes quickly, then human effort concentrates in the final, hardest portion of the work.

Independent testing of AI coding tools has repeatedly reproduced this shape. Reviewers who set the tools a complete, realistic project, rather than an isolated snippet, find that they accelerate the early scaffolding impressively and then stall at the same point: the integration of the pieces, the handling of the unexpected, the final push to something robust. The tools do not fail visibly. They produce something that looks finished and is not, which is a more dangerous failure than an obvious one, because it invites a team to declare victory at seventy percent and discover the remaining work only when it is deployed.

When the measurement is rigorous, the gains can vanish

The productivity numbers that fuel the easy-button narrative come largely from bounded, greenfield tasks. What happens when researchers measure experienced developers doing real work on codebases they know well? In 2025 the research organization METR ran what is, to date, the most rigorous controlled trial of the question. It recruited sixteen experienced open-source developers, gave them two hundred and forty-six genuine tasks drawn from the large, mature repositories they normally maintain, and randomly assigned each task to be done with or without AI assistance. Before starting, the developers expected AI to speed them up by about 24 percent. After finishing, they believed it had sped them up by about 20 percent. In fact, measured against the clock, they were 19 percent slower when using the AI.

The gap between perception and reality is the finding that should give every leader pause. The developers could not tell that the tool was slowing them down. They felt faster while being slower, because the experience of prompting, waiting, reviewing, and correcting feels productive even when it consumes more time than simply doing the work. This does not mean AI slows everyone in every setting, and the study's authors are careful about its limits: it used the models available in early 2025, it involved expert developers on code they knew intimately, and the sample was small. Newer tools and different tasks may produce different results. But it punctures the assumption that measured productivity gains are universal, and it demonstrates that developer enthusiasm is not reliable evidence of developer speed. When a team reports that AI makes it faster, that report may be true, or it may be the same illusion the METR developers experienced.

Quality erodes: duplication, churn, and the decline of refactoring

Speed is only half of the question. The other half is what the code is like to live with afterward, and here the independent evidence is troubling. The software analytics firm GitClear examined roughly two hundred and eleven million changed lines of code across the years spanning the rise of AI assistants, from 2020 through 2024. It found that the proportion of copy-pasted lines rose from about 8 percent in 2021 to more than 12 percent in 2024, and that for the first time in the data's history, copy-pasted code exceeded code that had been moved or refactored. Over the same period, the share of lines that represented refactoring, the disciplined restructuring that keeps a codebase healthy, fell sharply, from around a quarter of changes to under a tenth. The frequency of duplicated code blocks rose roughly eight-fold.

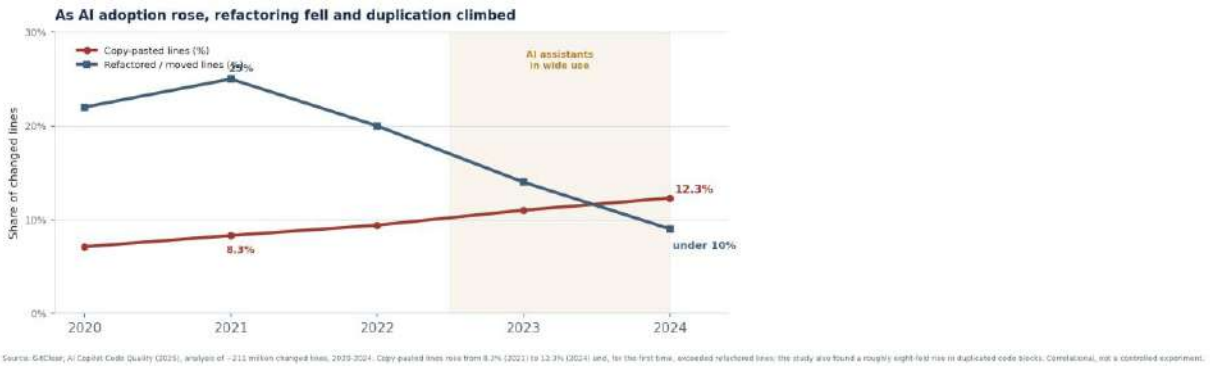

Figure 7. GitClear's analysis of 211 million changed lines. As AI assistants came into wide use, copy-pasted code rose and refactoring fell, both signs of eroding maintainability.

The interpretation GitClear offered is memorable: AI-generated code resembles an itinerant contributor, one who passes through, adds what is needed for the moment, and moves on without regard for the coherence of the whole. Duplication and low refactoring are not cosmetic concerns. They are the leading indicators of code that becomes progressively harder to change, because the same logic exists in many places and must be updated in all of them, and because the structure decays rather than improving over time. This is technical debt accumulating faster, and it lands squarely on whoever must maintain the system, which in a build scenario is the organization itself, forever. It is worth noting that GitClear's analysis is correlational rather than a controlled experiment, so it establishes association rather than proof of cause. But the direction is consistent with the broader evidence, and the mechanism is plausible: a tool that makes it effortless to generate more code, and offers no equivalent pressure to consolidate it, will tend to produce more sprawl.

The systemic view reinforces the point. Google's DORA research, one of the most respected long-running studies of software delivery, reported in 2024 that a 25 percent increase in AI adoption was associated with a 7.2 percent decrease in delivery stability and a small decrease in throughput, even as individual developers reported feeling more productive. In the same research, a large share of developers said they had little or no trust in AI-generated code, a striking admission from the people using it daily. The message from the systemic data is not that AI is useless. It is that individual speed and organizational health are not the same thing, and that gains felt at the keyboard can coincide with losses measured across the delivery pipeline.

Why the same tool helps one team and harms another

The systemic penalty has a mechanism, and understanding it explains why the same assistant can lift one team and sink another. The DORA researchers traced much of the stability decline not to the code itself but to batch size. AI makes it easy to produce more change at once, and larger changesets have always been the enemy of safe delivery: they are harder to review, harder to test, and harder to unwind when something breaks. A team that pairs a fast code generator with small, frequently integrated changes can capture the speed without the instability. A team that lets the generator inflate the size of every change inherits the instability along with the speed.

The other determinant is what the organization already has in place to absorb the new code. In the 2024 research, roughly nine in ten organizations reported at least one platform-engineering practice, an internal platform that standardizes how software is built, tested, and shipped, and the quality of that platform strongly shaped whether AI helped or hurt. Where the platform was strong, its automated tests, security gates, and review practices caught the defects a generator introduces before they reached production; where the platform was weak, or existed only on paper, the same generator simply produced more unreviewed code faster. The tool did not decide the outcome. The surrounding system did. This is why the easy-button framing is so misleading: the value of AI assistance is gated by the maturity of the engineering organization around it, which is precisely the capacity a company that has been buying rather than building is least likely to have already accumulated.

A final, subtler effect explains why the reclaimed time so often fails to appear on the bottom line. The DORA researchers called it the vacuum hypothesis: when AI completes a valuable task faster, the time it frees does not reliably flow to other valuable work. It is absorbed by the ambient load of meetings, coordination, and administrative friction that fills any gap. The developer feels the acceleration at the keyboard, and the acceleration is real, but it evaporates before it reaches the delivery metrics, which is one more reason the felt benefit and the measured benefit so stubbornly disagree.

Security is a structural problem, not a temporary one

Of all the hidden costs of AI-assisted building, security is the one most likely to cause serious harm, and the evidence here is unusually clear. In a controlled study published in 2023, researchers at Stanford found that developers who used an AI assistant wrote code that was measurably less secure than developers who did not, and, worse, were more likely to believe their code was secure. The combination is the most dangerous possible: less competence paired with more confidence. A developer who knows they are uncertain will seek review. A developer who wrongly believes the AI has handled security will not.

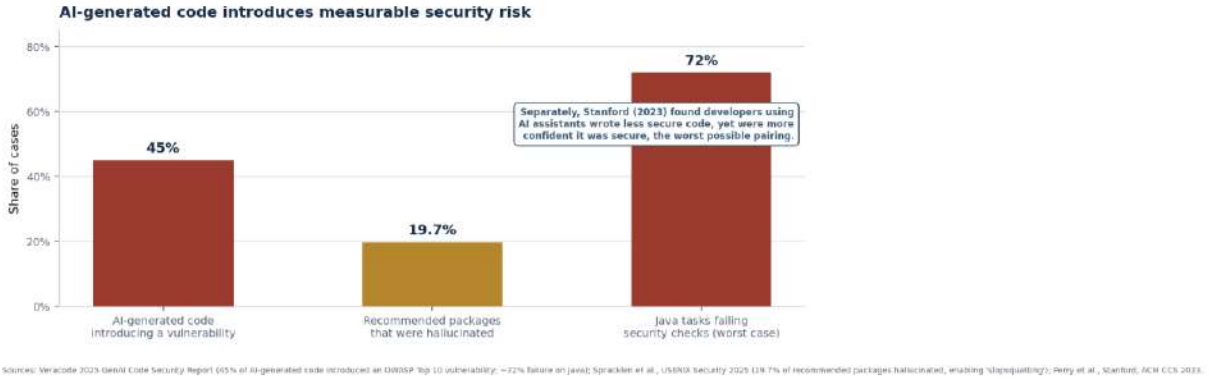

Figure 8. Independent security findings. A large share of AI-generated code contains vulnerabilities, a fifth of recommended packages do not exist, and AI users write less secure code while feeling more confident.

The scale of the issue has since been quantified. In 2025 the application-security firm Veracode tested more than a hundred large language models across dozens of coding tasks and found that about 45 percent of the AI-generated code introduced a vulnerability from the industry-standard list of common weaknesses. In some languages the rate was far higher, with Java-related tasks failing security checks in roughly seventy percent of cases. The most sobering detail was not the headline number but the trend: newer and larger models did not produce more secure code. The pass rate had stayed close to the same level for two years, which suggests the problem is structural rather than a temporary limitation that the next model generation will resolve. Producing functional code and producing secure code are different skills, and progress on the first has not brought progress on the second.

A new attack surface: hallucinated dependencies

AI-assisted building has also created an entirely new category of supply-chain risk, one that did not exist before and that illustrates how the technology can introduce novel failure modes even as it removes old friction. Large language models sometimes recommend software packages that do not exist, confidently

inventing a plausible-sounding library name. A landmark 2025 study that generated over half a million code samples across sixteen models found that nearly 20 percent of the packages the models recommended were hallucinations, amounting to hundreds of thousands of unique fictitious package names. Open-source models hallucinated at more than four times the rate of commercial ones, and, critically, the same fake names tended to recur across repeated prompts, making them predictable.

Predictability is what turns a curiosity into a threat. Security researchers named the resulting attack slopsquatting: an adversary observes which package names the models reliably hallucinate, registers those names in the public repositories, and fills them with malicious code. A developer who trusts an AI recommendation and installs the suggested package then imports the attacker's payload directly into the system being built. The defense is straightforward in principle, verify that every dependency actually exists and is legitimate before using it, but it is one more verification burden that AI shifts onto the human, and one more way in which building with AI is not the frictionless activity it appears. The broader pattern is the same across quality, security, and dependencies: AI produces output quickly, and the work of confirming that the output is correct, safe, and real falls to people, which is exactly the work that does not scale with a faster code generator.

~45%
of AI-generated code introduced a security vulnerability in Veracode's 2025 testing

~20%
of packages recommended by models in one large study did not exist

Stuck
security pass rates have not improved across two model generations

The compounding maintenance tax of generated code

Most discussions of AI-assisted development measure a single moment: how fast a task was completed, how much code was produced, whether a bug was introduced. The more important question is what happens to that code over the years it must be maintained, because software is not written once and left alone. It is read, changed, and debugged far more often than it is created, and generated code interacts with that reality in ways that compound.

The first compounding effect is volume. Assistants make it easy to produce more code, and more code is not a benefit; it is a liability that happens to be necessary. Every line must be read, understood, tested, secured, and maintained. When the marginal cost of generating a line falls to nearly zero while the lifetime cost of owning it does not, the natural result is systems that are larger than they need to be, and size is the single best predictor of the cost and risk of change.

The second effect is duplication and churn. Independent analysis of large volumes of code has found that as assistant use has spread, the share of code that is duplicated rather than reused has risen, and the share of code rewritten or reverted shortly after being written has grown. Duplication means a single fix must be made in many places, and high churn means effort is being spent producing code that does not survive. Both are signatures of code written faster than it is understood, and both raise the maintenance burden long after the initial speed-up is forgotten.

The third effect is the quiet decline of the practices that keep a codebase healthy. Refactoring, the disciplined restructuring of existing code to keep it comprehensible, tends to fall when the incentive is to generate the next thing rather than to improve the last thing. A codebase that stops being refactored does not fail suddenly; it degrades, becoming gradually harder to change until the cost of every modification is high and the confidence behind every change is low. This is the ordinary way that generated code turns a fast start into a slow decade.

The conclusion is not that assistants should be avoided, but that their output must be governed by the same discipline as any other code, and arguably more. The speed is real, and so is the tax, and the tax is paid later and by different people than the ones who enjoyed the speed. An organization that adopts these tools without also strengthening its review, its testing, and its refactoring practices is not lowering the cost of ownership. It is deferring and enlarging it.

The adoption gap: usage rises while the evidence hardens

The strongest recent evidence is not that AI fails but that its benefits have proven far harder to realize than adoption figures alone would suggest. By 2025, developer surveys found that roughly five in six professionals were using or planning to use AI coding tools, up sharply from the year before, and about half were reaching for them every day. Adoption, in other words, is close to universal. Yet in that same population only about one in six reported that the tools made them significantly more productive, while more than four in ten reported little or no effect at all. The tools are everywhere; the measured payoff is not.

Sentiment has moved in the opposite direction from adoption. Favorable opinion of these tools, which stood above seventy percent in the first flush of enthusiasm, had fallen toward sixty percent by 2025, and for the first time a larger share of developers said they distrusted the accuracy of the output than said they trusted it. This is the ordinary signature of a technology passing from novelty into daily use. The demonstrations still impress, but the people who depend on the tools have learned where they break, and their expectations have adjusted to the reality of reviewing and repairing what the machine produced.

The controlled evidence tells the same story in a sharper form. When researchers measured experienced developers working on their own mature codebases, rather than on isolated puzzles, they found that the developers were slower with AI than without it, even as those same developers believed themselves to be faster. The measured penalty and the perceived benefit pointed in opposite directions, which is precisely the trap. The visible part of the work, the first draft, accelerates, while the invisible part, the prompting and the waiting and the reviewing and the fixing, quietly absorbs the gains. The frontier keeps moving, and later measurements of newer tools will read differently, but the gap between felt and measured productivity is the durable lesson.

For a build-versus-buy decision, the implication is not that the tools are worthless but that their productivity cannot be assumed. A plan that justifies a build on the promise of a two- or threefold speed-up is resting on the vendor demonstrations rather than on the controlled measurements, and the two disagree. The disciplined planner treats AI acceleration as a real but modest and unevenly

distributed effect, largest on greenfield scaffolding and smallest on the mature, high-stakes systems where most enterprise work actually lives.

10 The hard parts AI does not solve

Step back from the code itself and a larger truth comes into view. Writing code was never the whole of building software, and in most enterprise projects it was not even the majority. Building a system that a business can rely on involves understanding what is actually needed, designing how the pieces fit, connecting to the systems already in place, getting the data into usable shape, proving the result is correct, securing it, and then operating and evolving it for years. Writing the code is one slice of that work. AI accelerates the slice, and leaves the rest largely untouched.

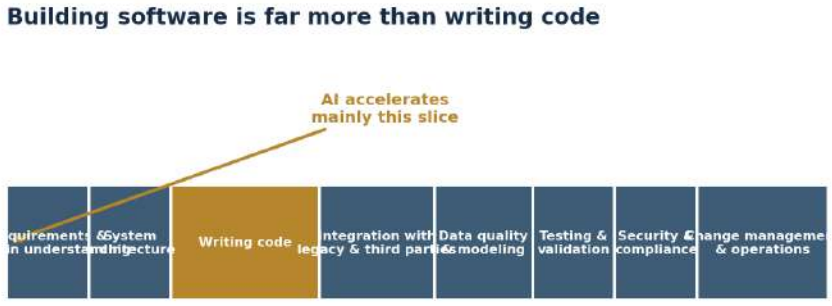

Figure 9. Building software is far more than writing code. AI mostly speeds the coding slice, while the larger share of the work, from requirements to operations, remains stubbornly human.

Consider each of the parts AI does not solve, because they are the parts that determine whether a build succeeds:

- **Requirements and domain understanding.** The hardest question in most projects is not how to build the thing but what to build, and that requires deep understanding of the business, the users, and the domain. A supply chain system embodies countless decisions about how a particular operation actually works, and no assistant can extract that understanding from an organization that has not done the thinking itself.
- **System architecture.** How a system is structured determines whether it can scale, evolve, and endure, and architectural decisions are exactly the ones with consequences too large and too context-dependent to delegate to a tool that sees only the immediate prompt.
- **Integration with existing systems.** Enterprise software does not stand alone. It must connect to the systems already running the business, each with its own quirks, and this integration is where enterprise projects most often bog down. The complexity lives in the specific, undocumented behavior of real systems, which is precisely what a general model does not know.
- **Data quality and modeling.** Software is only as good as the data beneath it, and enterprise data is notoriously messy, inconsistent, and incomplete. Modeling it correctly and cleaning it is painstaking, domain-specific work that AI can assist at the margins but cannot own.

- **Testing and validation.** Proving that a system is correct, especially across the edge cases that matter, is a discipline in its own right. AI can draft tests, but deciding what must be proven, and confirming that it has been, remains a human responsibility that grows more important, not less, when code is generated quickly.
- **Security and compliance.** As the previous section showed, securing a system is a continuous, expert activity, and regulatory compliance adds requirements that shift over time and carry legal weight. Neither is a coding task.
- **Change management and operations.** Software succeeds only if people adopt it and it keeps running. The organizational work of change, and the operational work of keeping a live system healthy, are among the largest costs of any system and lie entirely outside what a coding assistant addresses.

The talent paradox

A common hope is that AI will let organizations build serious software with less senior talent, since the assistant handles the difficult coding. The evidence points the other way. To use AI well on a real system, an organization needs experienced engineers who can architect the whole, judge whether the generated code is correct and secure, catch the subtle errors, and own the result. If anything, AI raises the premium on senior judgment, because it produces a larger volume of code that must be reviewed and a new class of plausible-looking mistakes that only an expert will notice. Meanwhile, there is a real concern that heavy reliance on AI may slow the development of junior engineers, who learn less when the assistant does the work, and may erode the pipeline of senior talent the approach itself depends on. The people who can truly own a complex system remain scarce, and AI does not remove the need for them. It sharpens it.

The forever commitment

All of these threads converge on a single, inescapable fact. A decision to build is not a decision to write a system once. It is a decision to understand, secure, integrate, operate, and evolve that system for as long as the business depends on it, which is usually far longer than anyone expects at the outset. AI does not remove this commitment. In one respect it can worsen it, because code generated faster than it can be understood becomes a system that must be maintained by people who did not fully author it and may not fully comprehend it. The maintenance iceberg described earlier does not melt in the warmth of a productive coding session. It remains, and it is carried alone by whoever chose to build. This is the reality that the easy-button narrative omits, and it is the reason the decision to build deserves the same sober scrutiny it always did, applied now with clear eyes about what AI has and has not changed.

Domain knowledge and the memory that leaves

The hardest parts of building software have never been about code, and they are precisely the parts an assistant cannot supply. Chief among them is domain knowledge: the accumulated understanding of how the business actually works, why the exceptions exist, and what the rules mean in practice rather than on paper. This knowledge is rarely written down, and where it is written down it is usually out of

date. It lives in the heads of the people who have run the operation for years, and it is the raw material from which good software is made.

An assistant can write a function to apply a rule, but it cannot tell you what the rule should be. It does not know that this customer is invoiced differently because of a settlement reached years ago, that this warehouse ships on Saturdays only during peak, that this product code is reused across two incompatible meanings for historical reasons, or that the number the report shows must reconcile to the figure a regulator expects rather than the one the system would naturally produce. These are the details that determine whether software is correct, and they come from the domain, not the model.

This is why the talent question is not only about whether you can hire engineers who can code. It is about whether you can pair them with people who understand the domain deeply enough to specify what correct means, and whether that understanding can be captured before the people who hold it move on. Software built without that pairing is fast, confident, and wrong in ways that surface expensively and late, because it faithfully implements a misunderstanding.

The memory problem compounds the talent problem. A system encodes a great deal of judgment that is never made explicit, and when the people who made those judgments leave, the reasoning leaves with them while the code remains. Their successors inherit behavior they cannot explain and are afraid to change, which is how organizations end up maintaining systems that no one fully understands and no one dares to touch. A vendor absorbs this risk across a large customer base and a stable product team; an in-house build concentrates it in a handful of individuals whose departure is a matter of when, not if.

None of this is an argument against building where building is warranted. It is an argument for being honest that the cost of a build is not paid mainly in engineering hours. It is paid in the scarce and fragile resource of institutional understanding, which must be extracted, encoded, and preserved for as long as the system lives. AI has made the code cheaper and has done nothing to make that understanding easier to capture or safer to keep. If anything, by making it easier to produce systems quickly, it raises the risk of building things whose logic no one ever fully held in the first place.

The regulatory surface area a build makes you own

There is one more hard part that AI does nothing to solve and, in a subtle way, makes more dangerous. A few years ago the decision to build or buy a software capability was mostly an economic and operational one. For AI systems it is now also a regulatory one, because law has begun to attach obligations directly to whoever is deemed to have built a system rather than merely used it. The clearest and most developed example is the European Union's Artificial Intelligence Act, formally Regulation 2024/1689, but the logic it encodes is spreading, and it maps onto the build-versus-buy question almost too neatly to ignore.

The Act draws a hard line between two roles. A provider is a party that develops an AI system, or has one developed, and places it on the market or puts it into service under its own name. A deployer is a party that merely uses such a system under its own authority. The two roles carry very different burdens. The provider bears the heavy obligations: a risk-management system, data governance, technical documentation, event logging, human-oversight design, testing for accuracy and robustness, a

conformity assessment, CE marking, registration in an EU database, and post-market monitoring for the life of the system. The deployer's duties, set out in Article 26, are real but far lighter: use the system in line with the provider's instructions, assign competent people to oversee it, monitor its operation, retain its logs for at least six months, inform the workers and affected individuals it touches, and carry out a fundamental-rights impact assessment where one is required.

For build versus buy the mapping is direct. A company that buys a high-risk AI system is, in the ordinary case, a deployer, and inherits the lighter set of duties, while the vendor, as provider, must carry the documentation, the conformity assessment, the CE marking, and the ongoing monitoring across its entire customer base. A company that builds the same capability becomes the provider, and inherits the full weight alone. This is the same shared-research-and-development logic that governs the economics, expressed now in law: the cost of compliance, like the cost of engineering, is amortized across a vendor's customers when you buy and concentrated on a single balance sheet when you build.

The line that turns a buyer into a builder

The detail that matters most for anyone drawn to the “buy the core and customize it heavily” path is where the line between the two roles actually sits, because it is easier to cross than most teams assume. Under Article 25 of the same regulation, a deployer that substantially modifies a high-risk system, puts its own name or trademark on it, or uses it for a purpose the original provider did not intend becomes a provider itself, and inherits every provider obligation. Fine-tuning a vendor's model on your own data, rebranding a white-label system, or repurposing a tool beyond its intended use can each be enough to flip you across that line. It is the regulatory echo of the customization trap that sank Lidl: bend a bought system far enough and you have, for legal purposes as much as practical ones, built it, and taken on the liability that follows.

The obligations are arriving on a staged schedule rather than all at once. The Act entered into force in August 2024; its outright bans and its AI-literacy duties applied from February 2025; its rules for general-purpose models from August 2025; and the bulk of the high-risk obligations from August 2026, with a 2025 and 2026 simplification package deferring certain high-risk categories, including biometrics, critical infrastructure, education, and employment, to late 2027, and systems embedded in already-regulated products to 2028. The penalties are calibrated to be felt: up to thirty-five million euros or seven percent of worldwide annual turnover for the most serious violations, and up to fifteen million or three percent for breaches of the high-risk obligations. And a point procurement teams miss at their peril: a vendor being “AI-Act compliant” does not make you compliant, any more than using a certified supplier makes you certified. The vendor supplies the components and the documentation; as a deployer you still own your own oversight, monitoring, incident handling, and evidence.

The European regime is the most fully formed, but it is not alone. Sectoral rules in financial services and healthcare, data-residency and privacy law, and a growing body of national AI regulation all attach graduated duties to whoever is treated as having built or materially altered a system. None of this is a coding task, and none of it is made easier by a tool that writes code quickly. If anything, the ease of generating a bespoke system raises the odds of quietly becoming a provider, of building something that carries obligations the organization never budgeted for and is least equipped to discharge. Compliance,

like the other hard parts, is a reason the decision to build must be made with open eyes, and increasingly with counsel in the room.

The forever commitment, made concrete

It is worth making the forever commitment concrete, because leaders routinely underestimate it by imagining a build as a project with an end rather than a system with a life.

Consider what owning even a modest internal system actually entails, year after year, once it is built. Its dependencies, the libraries and frameworks it stands on, release new versions and retire old ones, and keeping current is not optional, because the versions it depends on eventually stop receiving security fixes. The platforms it integrates with change their interfaces, and each change demands work. The regulations it touches evolve. The original developers move on, and their knowledge must be replaced or the system becomes a black box. None of this is failure; it is the ordinary weather of software, and it never stops.

This is why the honest unit of a build decision is not the cost to reach launch but the commitment to fund a team, indefinitely, to keep the system alive. A capability that genuinely differentiates you is worth that commitment; a commodity is not, which is the entire reason to buy it and let a vendor absorb the perpetual maintenance across its whole customer base. The question a leader should sit with before any build is not whether the organization can build the thing, but whether it is prepared to still be maintaining it in ten years, in competition with everything else that will demand those same engineers. If the honest answer is no, the decision is already made.

AI does nothing to shorten this commitment. It may help write the first version faster, but the first version is the beginning of the obligation, not the end of it, and a faster start to a forever commitment is not the bargain it appears to be.

11 How AI is strengthening the case to buy

There is a further turn in the argument that the easy-button narrative misses entirely. AI is not only a tool for building. It is also, and increasingly, a tool that commercial vendors are embedding in their products, and this is strengthening the case to buy rather than weakening it. Every major supply chain software provider is now investing heavily in AI, from forecasting and anomaly detection to conversational assistants and, most recently, autonomous agents that can take action within defined bounds. When an organization buys such a product, it inherits those AI capabilities in production-grade, maintained, continuously improving form, without carrying any of the burden of building or securing them.

The pace of this investment is worth appreciating, because it changes the arithmetic. A single company deciding to build its own AI-enabled planning or execution capability is competing against vendors pouring hundreds of millions of dollars, and years of specialized research, into exactly that problem, across a customer base that funds continuous improvement. The question the build-minded leader must answer is no longer merely whether building is feasible, but why the organization believes it can out-invest and out-maintain a vendor whose entire business is that capability. For a truly differentiating need, the answer may be sound. For a commodity capability that vendors are already advancing rapidly, rebuilding it in-house is a race against a far better-funded field.

What the vendors are shipping

The specifics illustrate the point. Blue Yonder has introduced a suite of domain-specific AI agents spanning network, warehouse, and logistics decisions, running a structured sense-analyze-decide-act loop over a unified data foundation, and reports customers using its capabilities to cut transportation costs and improve forecast accuracy and on-time delivery. Kinaxis has embedded AI across its concurrent-planning platform. o9 Solutions built its offering around a knowledge-graph approach that ties planning decisions to a connected model of the business. SAP, Oracle, and Manhattan Associates are each shipping embedded and increasingly agentic capabilities within their suites. Independent analysts regard the architectures underlying several of these platforms, concurrency, knowledge graphs, unified data foundations, as genuine and hard to replicate, precisely the kind of deep capability that a from-scratch build would struggle to match.

Vendor	AI direction (illustrative)	Buy-side implication
Blue Yonder	Domain-specific agents; sense-analyze-decide-act loop on unified data	Production agentic capability without the build
Kinaxis	AI embedded across concurrent planning	Hard-to-replicate concurrency architecture
o9 Solutions	Knowledge-graph model tying decisions to the business	Deep model a build would struggle to match
SAP / Oracle	Embedded and agentic AI across the suite	Broad capability funded across customers

Vendor	AI direction (illustrative)	Buy-side implication
Manhattan Associates	AI within warehouse and transportation execution	Maintained, continuously improved

Vendor capabilities and claims are drawn from public disclosures and should be verified directly, as the pace of change is rapid and marketing frequently runs ahead of what is in production. That caution matters, but it does not change the structural point: the direction of vendor investment is toward exactly the AI capabilities an organization might be tempted to build, and buying captures them without the ownership burden.

The caveats on the buy side

Even-handedness requires acknowledging that buying is not free of its own AI-related risks. The first is credibility. Not every vendor claim of intelligent capability reflects a mature, in-production reality, and separating genuine capability from what the industry calls AI-washing requires the same scrutiny a buyer would apply to any other claim. Some capabilities marketed as autonomous remain in pilot. The second risk is lock-in and data governance. AI features are often tied to a vendor's platform and trained on or fed by the buyer's data, which raises legitimate questions about portability, transparency, and control that belong in any evaluation. The third is that buying still requires the organization to configure, integrate, and adopt the product well, which is real work, though far less than building and maintaining the equivalent. None of these caveats reverses the conclusion. They refine it: buy the commodity, but evaluate the vendor's AI with clear eyes, and negotiate for the data rights and portability that protect the organization over time.

The economics of shared research and development

The deepest reason the case to buy is strengthening has little to do with any individual feature and everything to do with how the cost of software is amortized. A vendor spreads the cost of building and improving a product across its entire customer base. You pay a fraction of a shared investment; a build asks you to fund the whole thing alone. Artificial intelligence has widened this gap rather than narrowing it, because it has raised the pace at which capable vendors can ship.

Consider what a serious platform vendor now invests in a single category. It funds a large product and engineering organization, a dedicated security function, a compliance and certification effort, a research team pursuing embedded and increasingly agentic AI, and a support and implementation network, all sustained by the combined spend of hundreds or thousands of customers. No single buyer of that software could rationally reproduce that investment for its own use, because the return depends entirely on spreading the cost across many buyers. This is the structural advantage of buying, and it existed long before AI.

What AI changed is the rate of improvement that this shared investment now buys. The same vendors are pouring research into assistants, copilots, and agents embedded directly in their products, and every customer inherits those capabilities as part of the subscription, tested and supported, without owning the underlying complexity. A capability that would take an in-house team a year to build, secure, and maintain arrives in a release note. The buyer who chose the platform gets the improvement for the

price they were already paying; the builder who chose to go it alone must now fund the equivalent work themselves, forever, just to keep pace.

There is a compounding dynamic here that favors the buyer over time. A vendor with more customers has more revenue to reinvest, more usage data to improve the product, and more incentive to keep advancing it, which attracts more customers still. The gap between what a mature vendor can offer and what an individual company can build for itself tends to widen rather than close, because the vendor's advantage is structural and self-reinforcing. Betting on a build is, in part, a bet that you can outrun that dynamic with a fraction of the resources, and the base rates suggest that is a bet most organizations lose.

The caveat, developed in the next section, is that shared investment cuts both ways: a vendor optimizes for the average of its market, not for you specifically, and the very standardization that makes buying efficient can leave a genuine point of differentiation unserved. That is the seam the third path is designed to exploit. But the starting presumption, for any capability that does not differentiate you, should be that a shared investment you rent will beat a solitary investment you own, and that AI has made this truer, not less true.

The agentic turn: what the vendors are now selling

The most consequential recent shift on the buy side is that vendors have stopped selling features and started selling autonomy. Through 2025 and into 2026 the frontier moved from the assistant that suggests a line of code to the agent that takes a whole task, reads the surrounding system, makes coordinated changes across it, runs the tests, and iterates without a human touching the keyboard in between. The same movement is visible in the products enterprises actually buy. Planning suites, transportation systems, and procurement platforms now ship not merely with dashboards and forecasts but with agents that execute the routine workflow the dashboard used to leave to a person.

The market is repricing accordingly. Gartner projects that supply chain management software carrying genuine agentic capability will grow from less than two billion dollars of spend in 2025 to roughly fifty-three billion by 2030, and that the share of enterprises using such features will climb from around five percent to sixty percent over the same span. Whatever the precise figures turn out to be, the direction is unambiguous. The capability a company might have contemplated building in 2023 is becoming a line item on a vendor's price list, and it is arriving with the vendor's research budget, security review, and update cadence already attached.

There is an important subtlety beneath the marketing, and it strengthens rather than weakens the case to buy. A working agent is not a model; it is a model wrapped in a harness, the accumulated scaffolding of tools, context management, testing, and guardrails that turns raw capability into something dependable. That harness is expensive to build and expensive to keep current as the underlying models change every few months. It is precisely the kind of shared, fast-depreciating investment that favors renting over owning, because a vendor amortizes it across thousands of customers while a lone builder pays for all of it and rebuilds much of it with each new model generation.

The caveat is that availability is running well ahead of deployment. The same analysts are equally clear that enterprise adoption will lag the general availability of these features, because the surrounding operating model, the data, the processes, and the people, does not change as quickly as the software. For the buyer this is reassuring rather than alarming. It means the pressure to build one's own agent in order to keep pace is largely illusory: the capability will be available to buy long before most organizations are ready to use even the bought version well, which is the opposite of a situation that rewards a from-scratch build.

The compliance dividend of buying

One more form of shared investment deserves its own mention, because it has grown sharply in importance and the easy-button narrative ignores it entirely. Beyond research and engineering, a vendor now amortizes the cost of compliance. As Section 10 described, regulation increasingly attaches its heaviest obligations, risk management, technical documentation, conformity assessment, and ongoing monitoring, to whoever builds or materially alters an AI system. A vendor that builds such a system bears those provider obligations once, across its whole customer base, and supplies its customers the documentation and conformity evidence they need to satisfy their own lighter duties as deployers.

For the buyer this is a dividend that compounds with the others. Buying a compliant product does not discharge your obligations, you remain a deployer with real duties, but it means the most expensive and specialized part of the burden, the provider's, is carried by the party best equipped to carry it and spread across everyone who buys. A builder inherits that burden alone, and inherits it precisely where it is heaviest, funding the risk assessments, the technical documentation, the conformity work, and the perpetual monitoring from a single balance sheet. As the regulatory surface area grows, which every indication suggests it will, this dividend widens, and it widens in the same direction as all the others: toward buying the commodity and reserving the build for the differentiator whose compliance burden you have chosen, with open eyes, to own.

12 The third path: buy the core, build the edge

The framing of build versus buy as a binary choice is itself part of the problem, because the best answer is usually neither pure building nor pure buying but a deliberate combination of the two. The modern consensus, and the posture this guide recommends for most organizations, can be stated in a single line: buy the commodity, build the differentiator. Acquire a mature, well-maintained platform for the capabilities that do not set you apart, and reserve your building effort for the thin layer that does. This is not a compromise between building and buying. It is a more precise application of the core-versus-context test, capability by capability, rather than system by system.

Architecturally, this hybrid has become far more achievable than it once was. The rise of composable design, sometimes described through the principles of microservices, application programming interfaces, cloud-native delivery, and headless front ends, means that a bought core no longer has to be a monolith that resists extension. Modern platforms increasingly expose their capabilities through well-defined interfaces, which allows an organization to buy a robust foundation and then build specific, differentiating extensions and workflows on top of it, connected through those interfaces. Buyers of composable architectures commonly cite faster time to market and lower total cost of ownership over a multi-year horizon, though the approach demands more architectural discipline and can be more than a smaller organization needs. The trade-off is real, but the direction is clear: the choice is no longer all-or-nothing between a rigid packaged suite and a system built from scratch.

Where AI actually belongs

This hybrid is also where AI-assisted development finds its best and safest home. The work of building thin differentiating layers on top of a bought platform, integration code that connects systems, custom workflows that encode a particular way of operating, extensions that add a specific capability, is exactly the kind of well-bounded work at which AI is strongest, and it is contained enough that AI's weaknesses are manageable. Using AI to accelerate the construction of a connector, or a custom report, or a workflow that sits atop a mature and maintained core, plays to the tool's strengths of drafting bounded components quickly, while the platform beneath carries the burdens of scale, security, and maintenance that AI handles poorly. The mistake is to point AI at the core, asking it to generate the foundational system that must be correct, secure, and maintained forever. The wiser move is to buy that core and point AI at the edge, where fast drafting is valuable and the cost of imperfection is contained.

Principles for the hybrid path

- **Classify capability by capability.** Apply the core-versus-context test to each capability, not to whole systems, so that you buy the commodity and build only the specific differentiator.
- **Buy a core that is built to extend.** Favor platforms that expose their capabilities through open interfaces, so a bought foundation can carry differentiating layers rather than blocking them.
- **Point AI at the edge, not the core.** Use AI to accelerate integrations, workflows, and extensions, where its speed helps and its weaknesses are contained, and let the vendor carry the core.

- **Keep the differentiating layer thin.** Build the least that delivers the advantage, because every line you build is a line you must secure, integrate, and maintain for as long as it runs.

Patterns for building at the edge

Buying the core and building the edge is easy to say and easy to get wrong, because the edge has a way of expanding until it has quietly become another core system to maintain. The discipline of the third path is to keep what you build thin, specific, and defensible, and there are a handful of recurring patterns that tend to stay on the right side of that line.

The first pattern is the layer of proprietary logic on top of a bought platform. The commodity beneath, the warehouse system, the transportation system, the planning engine, is purchased and configured, and the thin slice of genuine differentiation, a particular optimization, a specific scoring model, a piece of logic no vendor implements the way you need, is built and owned. The bought system does the heavy, undifferentiating work; the built layer does the small, differentiating work, and because it is small, it is affordable to own well.

The second pattern is integration and orchestration between bought systems. Almost no organization runs a single system, and the connective tissue between them, the flows that move data, reconcile records, and coordinate processes across platforms, is rarely something a vendor sells and is frequently where a company's real operating advantage lives. This is legitimate build territory, and it is also where AI assistance is genuinely useful, because much of the work is bounded, well-specified translation between known interfaces rather than open-ended invention.

The third pattern is the customer-facing experience built on top of standard operations. The way a customer sees, tracks, and interacts with your service can be a real differentiator even when everything behind it is commodity execution. Building a distinctive experience on a bought operational core lets you invest your scarce engineering effort where a customer can actually perceive it, while renting the machinery that the customer neither sees nor rewards you for owning.

What unites these patterns is a deliberate boundary. The build is defined narrowly, bounded by the interfaces of the systems it sits on, and justified by a differentiation that survives the core-versus-context test. The failure mode is boundary creep, where the edge steadily absorbs responsibilities that the platform could have carried, until the organization is once again maintaining a large custom system it never decided to build. The third path is not a license to build; it is a discipline for building only the difference, and for keeping the difference small enough to own for as long as it matters.

The same pattern across functions

This guide has drawn most of its examples from supply chain, but the third path is not a supply-chain idea; it is a general one, and it is worth seeing it hold across the functions where technology decisions are made. In each, the same line divides the commodity core that should be bought from the thin edge that may be worth building.

Function	Buy the commodity core	Build the thin edge, only if it differentiates
Sales and CRM	A mature platform for accounts, contacts, and pipeline	Proprietary lead-scoring, bespoke sales workflows, and the integrations that feed them
HR and payroll	A packaged HCM and payroll system, where regulation is heavy and undifferentiated	Little to nothing on the core; at most a workforce-analytics or talent layer that genuinely sets you apart
Finance and ERP	The general ledger and core ERP, adopted close to standard	Reporting, planning, and analytics layers and specific integrations, never the ledger itself
Data and analytics	A cloud warehouse or lakehouse for storage and compute	The proprietary models, transformations, and data products that are the actual advantage
Customer support	A packaged helpdesk and ticketing platform	The routing, automation, and knowledge logic that encode your particular service model
Marketing	A marketing-automation and content platform	The segmentation, attribution, and campaign logic unique to your go-to-market

The buy-the-core, build-the-edge pattern applied across functions. The commodity substrate is bought in every row; only the thin differentiating layer is ever a candidate to build.

The striking thing about surveying the functions side by side is how little the answer changes. In every one, the commodity substrate, the ledger, the CRM, the warehouse, the payroll engine, is something the market supplies well and competitively, and building it from scratch would mean paying to reproduce a solved problem. In every one, the differentiation, where it exists at all, lives in a thin layer of logic, models, or workflow that sits on top of that substrate. And in every one, that thin layer is exactly where AI-assisted development earns its keep, because the work is bounded and the cost of an imperfection is contained.

The exception is invariant as well. The case for building the core survives only where two conditions hold together: no packaged product genuinely fits the need, and the capability is itself a source of advantage rather than a cost of doing business. For a company whose product is software, the core of that product is the differentiator and belongs on the build side by definition. For nearly everyone else, in nearly every function, the core is context and the edge is core, and the discipline is to keep buying the former so you can afford to build the latter well.

A note on the third option: adopting open source

The build-versus-buy framing has a third term that deserves explicit mention, because it is neither building from scratch nor buying a commercial product: adopting open-source software. For many commodity capabilities a mature open-source project is the pragmatic answer, and it changes the economics in ways worth understanding.

Open source resembles buying in the ways that matter most: the software already exists, its development cost is shared across a community rather than borne alone, and adopting it means configuring and integrating rather than building. It resembles building in one important way: there is no vendor contractually obligated to support it, fix it, or carry its security and compliance burden, so the adopting organization inherits more of the ownership responsibility than it would with a commercial product. The result sits between the two poles, and where exactly it sits depends on the maturity of the project and the depth of the organization's own capacity to support it.

The disciplined way to place an open-source option is to run it through the same tests as any other. On the core-versus-context test it behaves like a buy: use it for commodity capabilities, not to reproduce your differentiator. On the total-cost test, its licence is free but its ownership is not, so the maintenance, integration, and security layers of Section 5 still apply, and for a thinly supported project they can apply more heavily than for a commercial product with a vendor behind it. Many organizations bridge the gap by buying commercial support or a managed version of an open-source project, which converts the ambiguous ownership back into a vendor relationship, the same shared-investment logic in a different wrapper.

Open source does not change this guide's thesis; it enriches it. It is often the best way to buy the commodity, especially where a standard is community-owned rather than vendor-owned, and it is almost never the right way to build the differentiator, because a capability visible to a public community is, by definition, not proprietary. Treated as a form of buying with a heavier ownership tail, it fits the framework cleanly, and it belongs in the evaluation of any commodity capability alongside the commercial alternatives.

13 The supply chain reality: a category built to buy

Everything argued so far applies to enterprise software broadly, but supply chain software is a special case, and the special case points firmly toward buying the core. There are four reasons, and together they make supply chain close to the least attractive category in which to build a foundational system from scratch, whatever the state of AI coding tools.

1. **The commercial market is deep and mature.** Warehouse management, transportation management, planning, order management, and procurement are served by well-capitalized, long-established suites that have absorbed decades of accumulated domain knowledge. Off-the-shelf products cover the overwhelming majority of standard requirements, which means a build starts not from a gap in the market but from a decision to re-create what already exists in mature form.
2. **The domain complexity is extreme.** Multi-echelon planning, constraint-based optimization, carrier and customs and port integrations, lot and batch tracking, cold-chain compliance, and multi-client billing each embody deep, specialized logic. This is the kind of complexity vendors have spent decades refining, and that a general-purpose coding assistant, however capable, does not possess.
3. **It is relentlessly integration-heavy.** Supply chain systems live or die by their connections, to enterprise resource planning, to carriers, to warehouses, to trading partners, and integration is precisely where enterprise projects most often stall. The recurring lament of supply chain technology is that the integration scoped as a small task becomes the largest part of the project, and this is the part AI helps with least.
4. **The failure history is a warning.** Supply chain and enterprise software carries a long record of costly failures, and while most involved packaged software rather than pure builds, the lesson is even more cautionary for building, because the failure drivers, heavy customization, poor data migration, and rushed timelines, are risks a from-scratch build multiplies rather than avoids.

A costly history worth studying

The record of large supply chain and enterprise software failures is instructive precisely because the causes recur. The examples below are drawn from contemporaneous reporting and published post-mortems, and their figures mix different measures, lost sales, write-downs, remediation costs, and abandoned-project spending, so they should be read as directional rather than precise. What matters is the pattern.

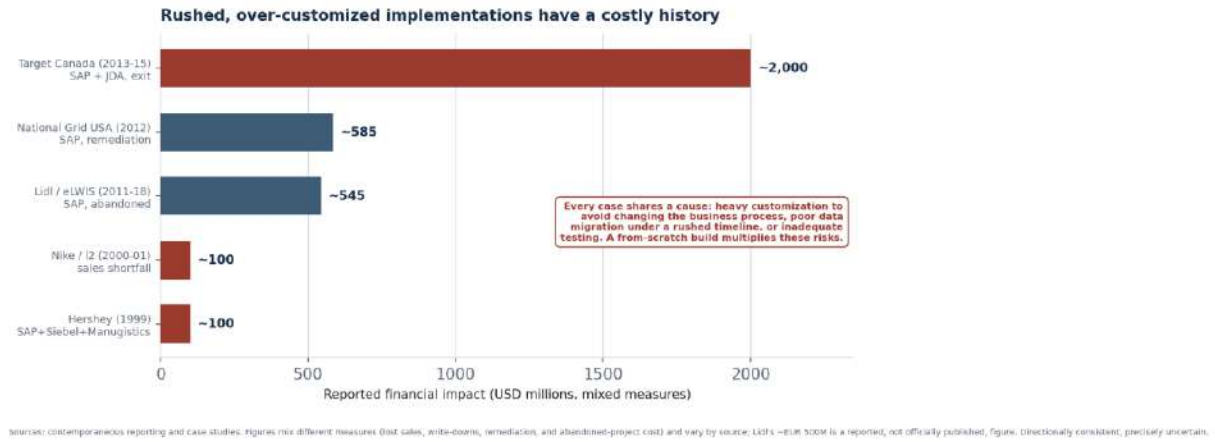

Figure 10. A selection of well-documented enterprise and supply chain software failures. In each, the driver was customization to avoid process change, poor data migration, or a rushed timeline.

Case	What happened	Recurring cause
Nike / i2 (2000-01)	A customized demand-planning rollout, rushed, tied to a large sales shortfall	Customization, rushed data
Hershey (1999)	A big-bang go-live before peak season left orders unfilled	Rushed timeline, scope
Target Canada (2013-15)	Flawed master data and a rigid schedule; shelves empty, then market exit	Data quality, timeline
Lidl / eLWIS (2011-18)	Heavy customization to preserve an existing process; abandoned	Customization over change
National Grid USA (2012)	A go-live just after a hurricane; large remediation cost	Timing, testing

The common thread across every case is not bad luck or bad technology. It is the decision to bend the software heavily to avoid changing the business process, to migrate data under a schedule that did not allow for its messiness, or to compress testing to hit a date. These are organizational and disciplinary failures, not coding failures, and they are exactly the failures a from-scratch build is more exposed to, not less, because a build carries all of that risk without the guardrails a mature product provides.

What this means for warehouse, transportation, and planning software

The general argument sharpens when applied to the specific systems supply chain leaders actually run, because each of the major categories illustrates the same lesson in its own way. Warehouse management systems encode decades of accumulated refinement in slotting, wave and task planning, labor management, and device integration, and the leading products cover the overwhelming majority of what any operation needs. Building one from scratch means re-creating that depth, and then maintaining it, alone. Transportation management systems embed carrier integrations, rating engines, optimization, and freight settlement that are painful to build and, harder still, painful to keep current as carriers, rates, and rules change constantly. Planning systems embed forecasting, inventory optimization, and constraint-based solving that represent a specialized science refined over many years.

In all three, the differentiating opportunity, where one exists at all, is a thin layer of proprietary logic sitting on top of the foundational system, not the foundational system itself.

There is a revealing data point in how the economics of these systems have evolved. Integration and ongoing services have become the largest cost and revenue segment surrounding warehouse and transportation software, which tells you plainly where the real and enduring work lives. It lives in connecting these systems to everything around them and keeping those connections healthy over time. A build does not escape that work. It inherits all of it, and adds the burden of constructing and owning the foundational system on top. This is why the recurring lament of supply chain technology, that the integration scoped as a small task becomes the largest part of the project, is a warning aimed squarely at anyone considering a build, since a build is integration-heavy by nature and carries no vendor to share the load.

Beyond the well-known failures already described, the record holds further cautionary cases in and around the supply chain, documented in contemporaneous reporting: a cosmetics company whose enterprise system rollout in 2018 disrupted order fulfillment and weighed on sales, a beverage manufacturer whose systems program ended in litigation, and others. In each, the driver was familiar, heavy customization, poor data readiness, or a compressed timeline, and in each, the lesson for a prospective builder is the same. If bending a mature product to fit an unchanged process can go this wrong, building the equivalent from scratch, without the product's accumulated guardrails, is a larger bet, not a safer one.

Where supply chain leaders have built well

None of this means building never succeeds in supply chain, and the successful cases are instructive because they mark the boundary precisely. Organizations have built well when they confined the build to a truly differentiating layer, a proprietary optimization, a distinctive customer-facing capability, or an orchestration that ties bought systems together in a way no product on the market offers, and placed that layer on top of bought foundations rather than in place of them. The largest logistics operators have built proprietary capability exactly where it is the source of their competitive advantage, and where they possess the rare combination of engineering depth and scale required to own it for the long run. The pattern in every successful build is the one the framework prescribes: buy the mature core, build the thin differentiator, and be rigorous about which is which. And the pattern in nearly every failed build is the inverse: a foundational system that should have been bought was built instead, or a packaged system was customized so heavily that it became a de facto build, carrying all of the risk of building and none of the protection of a supported product.

Four failures, in detail

Four of the cases named above reward a closer look, not for the grim spectacle but because the mechanism is more instructive than the numbers. In each, a capable organization with ample budget and a proven vendor still produced a costly failure, and in each the proximate cause sat in the space around the software, in data, timing, process, and the decision to customize, exactly the space a from-scratch build enlarges rather than removes. The figures that follow are drawn from contemporaneous

reporting and later published post-mortems, and because they mix lost sales, write-downs, and total program spend, they are best read as directional rather than precise.

Nike and i2: the hundred-million-dollar speed bump

In 2000 Nike went live with demand-planning software from i2 Technologies, the centerpiece of an ambitious program to shorten lead times and match production to demand. The rollout was a big bang across the company's United States and European operations, and it was fast: by later accounts Nike brought the system up in roughly a year, about half the time such programs typically take, and switched everything on at once rather than in phases. i2 carried a standing recommendation that customers limit customization of its software to something on the order of ten to fifteen percent. Nike, determined to preserve its own planning logic and to accommodate the shape of its existing data, customized well past that line.

The result was a system that behaved erratically under real volume. Forecasts came out wrong, orders were duplicated or dropped, and factories built the wrong shoes, too many of slow-selling models and too few of the ones stores were desperate for. To paper over the gaps Nike discounted heavily and air-freighted product at many times the usual cost. When the company disclosed the problem it warned of a quarterly earnings shortfall of roughly twenty-eight percent; its shares fell nearly twenty percent in a single day, and i2's fell further still. The lost sales were put at around one hundred million dollars, and the chief executive's rueful question on the earnings call, asking in effect whether this was what the company had to show for its investment, became the line everyone remembered. Analysts later placed the cost of the wider technology program far higher.

What makes Nike instructive is that the story did not end there. The company kept the broader platform, rebuilt its planning discipline, phased the remaining rollout instead of forcing it, and invested heavily in training, and within a few years the system was working as intended. The failure was not the software and not the ambition; it was the compression of the timeline and the customization undertaken to avoid changing the business. Those are precisely the risks a build carries in fuller measure, because a build is customization all the way down and has no vendor methodology to overrun in the first place.

Hershey: the Halloween the orders never shipped

Hershey's failure a year earlier is remembered for its timing. Through the late 1990s the company undertook a program of roughly one hundred and twelve million dollars to replace its legacy systems with three enterprise platforms at once, an SAP core, Manugistics for supply chain, and Siebel for customer management, and it compressed the schedule from a recommended forty-eight months to about thirty, in part to clear the effort before the Y2K deadline. To hit the date, testing was abbreviated, and the company chose to go live in July 1999, weeks before the orders for Halloween and the holiday season, the most concentrated and unforgiving stretch of its year.

When the systems came up, the integration between order processing and fulfillment buckled under the seasonal load. Candy sat in warehouses while orders went unshipped; Hershey was ultimately unable to fulfill something on the order of one hundred million dollars of product for retailers who had every

reason to expect it. Quarterly profit fell by about nineteen percent, and the stock dropped on the news. As with Nike, the platforms themselves were proven, and the company later ran them successfully after a calmer, phased reimplementation. The failure was one of scheduling and testing discipline, a big-bang cutover onto an unforgiving date, and it is exactly the discipline a build depends on even more heavily, because a build has no vendor-tested cutover path to follow and every integration is bespoke.

Target Canada: how bad data closed 133 stores

Target's expansion into Canada, launched in 2013, is the case that most directly implicates data. To stand up 124 leased locations, ultimately 133 stores, together with three distribution centers on a roughly two-year timeline, about half the three to five years such a rollout is usually given, the company deployed an SAP-based supply chain platform new to its Canadian team. Feeding that platform meant entering the master data for roughly seventy-five thousand products, and the entry was done at speed by inexperienced staff, often working from unreliable vendor-supplied information, with no validation to catch mistakes.

The errors were mundane individually and catastrophic in aggregate: dimensions recorded in the wrong units or the wrong order, prices mis-keyed, the occasional wrong currency, missing fields. Because replenishment and distribution keyed off that data, the consequences cascaded, shelves stood empty in stores while distribution centers overflowed with stock the system could not route. The company mounted a heroic cleanup effort and had begun to stabilize by late 2014, but the losses had already run into the billions and patience had run out. In January 2015 Target Canada filed for creditor protection; all 133 stores closed within months, and roughly seventeen thousand six hundred people lost their jobs. It was not the first Canadian grocer to founder on the same rock, a competitor had abandoned its own SAP effort more than a decade earlier, but it is the most complete illustration of a simple truth: a supply chain system is only as good as the data poured into it, and no amount of software sophistication rescues a rushed migration.

Lidl and eLWIS: half a billion euros, then back to the old system

Lidl's is the purest customization story of the four. Beginning in 2011 the German discount grocer set out to replace its aging in-house merchandise system with a standard SAP for Retail platform, a program it code-named eLWIS and staffed with something like a thousand employees and hundreds of outside consultants. It ran into a single, decisive mismatch. Lidl valued its inventory at the price it paid suppliers, while SAP's retail software was built around valuation at the price goods sell for. Rather than adopt the standard approach, Lidl chose to bend the software to its existing practice.

That one decision propagated. Each accommodation demanded another; the platform grew more customized, more fragile, and harder to maintain, and an effort originally budgeted at roughly two hundred million euros swelled toward five hundred million, near six hundred million dollars. The trajectory was disguised for years, SAP publicly honored Lidl as a flagship customer as late as 2017, but in July 2018 the company abandoned eLWIS and reverted to a modernized version of its old system, effectively writing off the investment. The lesson supply chain leaders draw from Lidl is the one this guide keeps returning to: when a mature product does not fit your process, the disciplined move is

almost always to change the process, because customizing the product past a certain point turns a buy into a de facto build, inheriting all of a build's risk while forfeiting the vendor's guardrails.

What a builder should take from four bought-software failures

It may seem paradoxical to marshal four failures of purchased software as an argument for buying, so the logic deserves to be explicit. Every one of these disasters befell an organization that chose to buy, and yet every one was caused by the very behaviors a build makes unavoidable: extensive customization to preserve an existing way of working, data migrated under a schedule that did not respect its messiness, and testing compressed to meet a date. Buying did not cause the failures; treating a bought system as raw material to be reshaped did. A build begins from that reshaped-material posture by definition. It is customization with no ceiling, integration with no vendor to share the load, and a cutover with no tested path, and it must supply, from the organization's own discipline alone, the guardrails these four companies discovered they could not do without.

This is the context in which the AI-era temptation should be read. The tools that now make it feel inexpensive to generate a working system do nothing to reduce the risks that actually sank these programs, which lived in process, data, integration, and timing rather than in the writing of code. If anything, by lowering the apparent cost of starting a build, they make it easier to walk into exactly the exposure that a mature product, and the discipline to adopt it as designed, exists to prevent.

Failures beyond retail: utilities, cosmetics, and the public sector

The four retail and consumer cases are the most famous, but the pattern is not confined to any one industry, and several more failures are worth naming because each isolates a different lesson.

National Grid, the utility that supplies gas and electricity across parts of the northeastern United States, went live with a major SAP platform on the fifth of November 2012, days after Superstorm Sandy had devastated its service territory. The program was already three years old and roughly thirty percent over its budget, and the decision to proceed on schedule rather than delay again was made under exactly the pressure that produces bad calls. The consequences were severe and specific: the payroll system miscalculated pay so that employees were overpaid, underpaid, or not paid at all, with some eight million dollars in overpayments never recovered; two months after go-live the company had a backlog of fifteen thousand vendor invoices it could not process; and the financial close that had taken four days stretched to forty-three, degrading the company's reporting so badly that it temporarily lost access to the short-term borrowing it relied on. Stabilization ran at roughly thirty million dollars a month, the cleanup took more than two years and cost on the order of five hundred and eighty-five million dollars, well above the original implementation, and the company's suit against its systems integrator was ultimately settled for seventy-five million, a fraction of the damage. The lesson National Grid isolates is timing and readiness: a go-live is a business event, not an IT one, and forcing it onto an unforgiving date is its own decision with its own consequences.

Revlon's failure, in 2018, isolates the danger of stacking a system migration on top of a merger. Having acquired Elizabeth Arden two years earlier, the cosmetics company chose to consolidate onto a new SAP platform, a system with which it had little hands-on experience, and brought it live without confirming

that its core business processes would actually work. The rollout disrupted its North Carolina manufacturing facility badly enough that the company could not ship roughly sixty-four million dollars of orders, and the disclosure sent its stock down and drew something rarer than the usual customer-versus-integrator dispute: class-action lawsuits from Revlon's own shareholders. The lesson is one of sequencing and readiness: an ERP migration is among the hardest things a company can attempt, and attempting it while still integrating an acquisition, without validated processes, invites exactly the outcome Revlon suffered.

Two other cases isolate the contractual side of the risk. Waste Management's 2005 SAP program, undertaken on the strength of projected annual benefits the vendor put at between roughly one hundred and two hundred and twenty million dollars and an eighteen-month timeline, foundered on the gap between what had been promised and what was delivered, and ended in a five-hundred-million-dollar lawsuit later settled out of court. MillerCoors, having accumulated seven separate SAP instances through years of industry consolidation, hired an integrator to unify them; the first rollout went live with eighty documented defects, eight of them critical, and the relationship collapsed into a hundred-million-dollar suit and countersuit before the parties resolved it. Neither company was undone by writing too little code. Both were undone by unverified promises and inadequate testing, and both spent in litigation what a proof of concept and a harder look at the contract would have cost a fraction of, the discipline that Appendix D and Appendix E exist to encode.

The public sector supplies the largest cautionary figure of all. Queensland Health's payroll replacement, which went live in 2010, is widely described as the most spectacular technology failure in the Southern Hemisphere; miscalculated and missed payments to tens of thousands of health workers, and years of remediation, drove its total cost past a billion Australian dollars, against an original budget a tiny fraction of that. It is the clearest illustration that these risks are not a private-sector peculiarity, and that payroll, perhaps the most commoditized capability in all of enterprise software, is precisely the kind of thing no organization should ever be building, or heavily customizing, from scratch.

When building the core was the right call

Honesty requires the other side of the ledger, because the case for buying is a default, not a dogma, and there are companies whose greatest advantages were built rather than bought. The point is not that building is always wrong; it is that building is right only where the capability is genuinely differentiating and no product can supply it, and the firms usually cited as build successes fit that description precisely.

Amazon is the canonical example. Its fulfillment and logistics software, its recommendation systems, and ultimately the cloud infrastructure it built for itself and then sold to the world as a separate business were not commodities it could have purchased; they were the substance of its competitive advantage, and building them was not a cost of doing business but the business itself. That the same company buys ordinary back-office software like everyone else only sharpens the point: it built where building differentiated and bought where it did not.

Walmart's decades-long investment in its own supply-chain and inventory systems, and Zara's tightly held software for turning a design into a store-ready garment in a matter of weeks, tell the same story.

In each case the built capability was inseparable from the strategy, fast and cheap replenishment for Walmart, rapid fashion cycles for Zara, and no vendor product could have delivered the specific advantage because the advantage was, in part, the software itself. These are not counterexamples to this guide's thesis. They are its clearest confirmations, because each company built exactly the thin, differentiating core that the third path reserves for building, and bought the vast commodity remainder around it.

What separates these successes from the failures earlier in this section is not competence or budget, though those helped, but the nature of what was built. The failures were companies buying and then customizing a commodity so heavily that they were building without admitting it. The successes were companies building the one thing that genuinely set them apart and buying everything else. The discipline is the same in both directions: know which is which, and let the classification, not the temptation, make the decision.

Shorter cautionary notes

A handful of further cases, told more briefly, round out the pattern and show its reach across sectors and delivery models.

Hertz sued Accenture to recover roughly thirty-two million dollars it had paid for a website redesign that, by Hertz's account, was never delivered in a usable state. The lesson is that the risk of a bespoke build does not vanish when the building is outsourced; it changes hands, and the contract, per Appendix E, is what determines whether it lands on the buyer or the builder.

The 2013 launch of the United States federal health-insurance exchange, an enormous bespoke integration built against a hard political deadline, failed on its first day, when only a tiny fraction of visitors could complete an enrollment. It was rescued only by an emergency team that applied the ordinary disciplines, smaller scope, real testing, phased delivery, that the original program had skipped. Scale and mission-criticality do not substitute for discipline; they raise the price of its absence.

Oregon's state health exchange spent more than three hundred million dollars attempting to build and integrate its systems simultaneously against the same deadline, and never processed a single application through the exchange before abandoning the effort. The diagnosis was scope: the project grew so large that no one could make the decisions it required in time. A smaller build, confined to what the deadline actually demanded, would very likely have worked.

A California school district's rollout of a bought human-resources and finance platform failed not because the software could not do the work but because the implementation, the data, the configuration, and the change management, was mishandled. It is a reminder that buying is no guarantee either; a bought system implemented without discipline fails in the same ways a build does, which is why Sections 15 and 16 treat readiness and vendor management as seriously as the build-versus-buy choice itself.

The fleet-management company LeasePlan sank close to a hundred million euros into an SAP program before abandoning it, later citing the monolithic nature of the system as a poor fit for its more agile

needs. The case sits on the boundary of this guide's thesis: sometimes the honest conclusion is that neither heavy customization of a monolith nor a from-scratch build is right, and the answer is the more composable architecture that Section 12 describes.

None of these is a story about code that was too slow to write. Each is a story about scope, data, testing, timing, contracts, and change, the space around the code, which is exactly the space that decides whether a project succeeds and exactly the space that AI leaves untouched.

The exception that proves the rule

There are organizations that have built supply chain technology to great effect, and they are worth understanding because they define the boundary of when building makes sense. The largest example is a company whose logistics capability is central to its competitive advantage and that has built proprietary fulfillment and optimization technology at enormous scale. That is not a counterexample to the argument here. It is a precise illustration of it. Building was justified because the capability was truly core, the source of durable advantage, and because the organization possessed engineering depth almost no other company can match, along with the scale to justify the permanent investment in maintaining it. Those conditions, a truly differentiating capability, deep and durable talent, and the scale to sustain ownership, are the conditions under which building the core is right. They are rare, and most organizations weighing a build do not meet them. The exception proves the rule: build the core when it is unmistakably your advantage and you can own it forever, and buy it otherwise.

The vendor landscape by category

Supply chain is a category built to buy, and the clearest evidence is the depth of the vendor market in every major function. Decades of competition have produced mature, specialized software for each layer of the operation, and the practical consequence is that a company building any of these from scratch is not filling a gap in the market; it is choosing to reproduce, alone, something many vendors already do at a depth a single team cannot match.

In warehouse execution, the market spans full-suite platforms embedded in enterprise resource planning systems, dedicated best-of-breed warehouse management vendors, and a newer generation of cloud-native providers, alongside the labor management, slotting, and automation-control capabilities that surround them. The functions themselves, receiving, putaway, picking, packing, shipping, cycle counting, wave and task management, are industry-standard and deeply refined. The differentiation a company feels in its warehouse is almost always in the physical operation and the configuration, not in the underlying software, which is precisely the definition of context.

In transportation, the market is equally deep: route planning and optimization, load building, carrier selection and rating, tendering, freight audit and payment, real-time visibility, and yard management are all served by established vendors, many of them specialized to a mode, a region, or an industry. Transportation optimization in particular embodies decades of operations-research investment that would be extraordinarily expensive to rebuild, and the visibility layer depends on carrier and telematics integrations that a vendor maintains across the whole network so that no single shipper has to.

In planning, the spectrum runs from demand forecasting and inventory optimization to supply and production planning, sales and operations planning, and the integrated business planning suites that tie them together, now increasingly wrapped in probabilistic and machine-learning forecasting. Planning is the function where companies most often believe their approach is uniquely theirs, and it is usually the configuration, the policies, the segmentation, the parameters, that is distinctive, sitting on top of algorithms the vendor has spent years refining. Buy and configure is the default posture precisely because the value lives in the tuning rather than the engine.

Around these three sit order management, network design, procurement and supplier management, global trade and compliance, and control-tower orchestration, each with its own field of specialized vendors. The point of surveying the landscape is not to endorse any product but to establish a base rate for the make-versus-buy decision in this domain: for almost every standard supply chain function, a mature market exists, competition has driven capability up and price toward commodity, and the burden of proof rests firmly on anyone proposing to build instead.

Patterns from the field

Beyond the categories, a few recurring patterns describe how build-versus-buy actually plays out in supply chain organizations, and they map cleanly onto the framework the rest of this guide develops. They are worth stating plainly because they recur across companies of very different sizes and sectors.

The most common regret is the home-grown system that outlived its era. Many companies built warehouse or planning software years ago, when the vendor market was genuinely thinner, and are still running it long after mature alternatives appeared. The system works, people are proud of it, and it quietly consumes the engineering capacity that could be spent elsewhere while falling further behind the capabilities that vendors now ship as standard. The trigger for change is usually a forced one, an integration that can no longer be sustained, a key engineer's departure, a capability the business needs and the old system cannot provide, and by then the migration is harder than it would have been years earlier.

The most common success is narrow and layered. Where supply chain organizations have built well, they have almost always built a thin, specific capability on top of bought systems: a proprietary optimization that reflects a genuine operating advantage, an orchestration layer that coordinates several platforms in a way no single vendor sells, a customer-facing service that differentiates on experience while standard systems run underneath. These builds succeed because they are bounded by the interfaces of the platforms they sit on and justified by a differentiation that survives the core-versus-context test.

The exception that recurs is the company whose supply chain is its product. A logistics provider, a carrier, or a marketplace whose entire value proposition is superior execution may rightly treat routing, visibility, or network optimization as core and build accordingly, because for them these capabilities are the reason customers choose them. The lesson is not that supply chain software is always a buy, but that the classification depends on whether the capability differentiates you in a customer's eyes, and

that for the large majority of companies, whose supply chain enables the product rather than being the product, the honest answer is that it does not.

What the market itself is telling us

The shape of the supply chain software market is itself an argument for buying. Independent estimates of its size vary widely, from figures in the high teens of billions of dollars to the low thirties for essentially the same year, a spread that says as much about where analysts draw the boundaries as about the underlying reality. But the estimates agree on the two things that matter here: the market is large, and it is growing at a healthy clip, somewhere from the high single digits to the low teens of percent a year. A market that size and that vitality is not a gap in the vendor landscape waiting for a company to fill by building. It is a sign that the commodity has been thoroughly commoditized.

The same research that sizes the market also quantifies the tax that building, or heavily customizing, quietly imposes. Analysts tracking implementation report that the integration work alone, the custom middleware, the data-transformation layers, and the validation testing required to connect modern software to aging systems, can consume up to sixty percent of a project's budget. Enterprise resource planning rollouts, the closest large-scale analogue to a supply chain build, still overshoot their timelines by roughly two hundred days and their budgets by about a third on average. These are not the numbers of a discipline that a capable in-house team can expect to beat by starting from scratch. They are the numbers that vendors, and their implementation partners, exist to absorb.

Consolidation is sending the same signal. The vendors themselves are increasingly buying rather than building their newest capabilities: acquisitions of AI-native planning and orchestration companies, and of emissions- and compliance-reporting specialists, have become the routine way that incumbents add frontier features. When the companies whose entire business is supply chain software conclude that acquiring a capability beats building it, a customer whose business is something else should draw the obvious inference about its own build-versus-buy math.

None of this overturns the core-versus-context test; it sharpens it. The market's size and growth mean that anything resembling a commodity capability will be available, improving, and falling in unit cost whether or not any single company builds it. The integration and overrun figures mean that even the act of stitching bought systems together is substantial work, and that is where a disciplined organization should be spending its scarce engineering capacity, rather than on rebuilding the systems themselves. The strategic reading of a large, fast-growing, consolidating vendor market is not that there is room to build, but that the returns to building have never been narrower.

A note on total-cost benchmarks

Leaders often ask for a rule of thumb, a number to sanity-check a build-versus-buy comparison, and while every situation differs, a few benchmarks recur often enough to be useful as a first filter.

- **Maintenance runs roughly fifteen to twenty-five percent of build cost per year.** Over a system's life this typically exceeds the original build, which is why a multi-year total, not a launch cost, is the only honest basis for comparison.

- **The true build cost runs well above the naive estimate.** Once the hidden layers are added, a build's lifetime cost commonly lands at one and a half to two times the figure first quoted, and higher when integration is heavy.
- **Large IT projects overrun by wide, well-documented margins.** The research behind Section 4 puts the average large project meaningfully over budget and delivering materially less value than promised, with roughly one in six overrunning catastrophically.
- **Customization is the single best predictor of trouble.** Vendors and post-mortems alike converge on the same warning: past a modest threshold, customization drives cost, delay, and instability, as every over-customized failure in this section attests.

These are filters, not formulas. Their purpose is to catch the estimate that is obviously too optimistic, the build quoted at a fraction of what its life will cost, the customization waved away as minor, before it becomes a decision. A comparison that respects them will not always favor buying, but it will at least be comparing the real numbers.

14 A decision framework for the AI era

The argument reduces to a practical decision, and the decision can be made systematically. What follows is a framework in four stages, designed to be worked through before any tool is selected and any code is written. It updates the classic build-versus-buy criteria for a world with capable AI, weighting each dimension by what AI actually changes and what it does not.

Figure 11. Where the decision lands. The vertical axis is Moore's core-versus-context test; the horizontal axis is the maturity of the commercial market. Most supply chain capability sits in the lower-right buy quadrant.

Stage one: classify the capability

Begin, before any discussion of technology, by classifying the capability against the core-versus-context test. Would a customer choose you because of it? Would rivals need years to match it? Does your competitive story depend on it? If not, it is context, and the default is to buy. The red flag at this stage is the claim that a standard capability is unique. Most warehouse and transportation and planning requirements are context dressed as core, and the discipline is to classify objectively rather than flatter the organization's sense of its own distinctiveness.

Stage two: model the lifetime cost, not the build cost

If a capability survives the first stage as a possible build, model its cost over five to ten years, not the cost of the initial build. Budget maintenance at 60 to 90 percent of lifetime cost, and treat integration as a first-class line item rather than an afterthought. Include the cost of securing the system continuously, of operating it, and of the key-person risk it creates. Then compare that honest lifetime figure against a mature vendor's total cost. If the build does not win decisively on a fully-loaded basis, buy, and remember the base rates: assume a real probability that the build lands over budget and under value, as most do.

Stage three: choose the architecture

For capabilities that truly warrant building, default to the hybrid: buy a mature core and build a thin differentiating layer on top, connected through open interfaces. Reserve pure building for the rare capability that is both truly core and unserved by any adequate product, and even then keep the built portion as small as the advantage allows. This is the stage at which AI enters correctly, pointed at the edge, the integrations, workflows, and extensions, rather than at the core, where its weaknesses in correctness, security, and maintainability are most dangerous.

Stage four: if you build or use AI heavily, install guardrails

Where an organization does build, and especially where it uses AI to do so, the final stage is to put in place the controls that contain AI's documented risks. Review AI-generated code with senior engineers rather than trusting it, and gate delivery on more than whether tests pass, including checks for duplication and code health. Validate that every dependency actually exists before it is used, to defend against hallucinated packages. Prohibit AI-only authorship of the most sensitive components, authentication, authorization, cryptography, payments, and require expert review there without exception. Favor a disciplined approach that plans before it generates and verifies after, rather than accepting output because it looks finished.

Red flags that should stop a build

- **"It is unique."** A claim that a standard capability is differentiating, when the underlying functions are industry-standard and well served by vendors.
- **"AI makes it easy now."** A business case that rests on AI having removed the hard parts, when the evidence shows AI accelerates only the coding slice.
- **"We will maintain it later."** A plan that budgets the build but not the 60 to 90 percent of lifetime cost that maintenance, security, and operations will consume.
- **"We do not need a storefront, or the integration will be quick."** An underestimate of integration, which is where supply chain builds most reliably stall.

Benchmarks that should decide

Two simple postures capture most of the guidance. Lean toward buying when a mature vendor covers most of the requirement, when senior engineering capacity is scarce, when integration complexity is high, and when the capability is context, which together describe the great majority of supply chain software decisions. Lean toward building, and then only a thin layer, when the capability is truly core and differentiating, when no adequate product exists, when the organization has durable senior talent to own it for its whole life, and when the scope can be contained to an extension rather than a foundational system. And reassess the whole question whenever vendor capability leaps forward, as it is doing rapidly with embedded and agentic AI, because each such advance raises the bar a build must clear to be worthwhile.

A simple scoring rubric

For teams that want to make the decision concrete, the dimensions above can be turned into a lightweight scoring exercise. Rate each dimension for the specific capability in question, note whether the honest answer pulls toward building or buying, and let the overall pattern emerge rather than seeking a single number. The rubric is a discipline for structuring the conversation, not an oracle that decides for you, and its value is in forcing each dimension to be considered explicitly rather than allowing one strong feeling, usually the belief that the capability is unique, to carry the whole decision.

Dimension	What favors building	What favors buying
Strategic differentiation	Truly core; a durable advantage	Context; table stakes
Vendor market maturity	No adequate product exists	Deep, mature field of products
Seven-to-ten-year total cost	Build wins fully loaded	Buy wins on lifetime cost
Time to value	Time is not pressing	Value is needed soon
Senior talent to own it forever	Durable, deep in-house team	Talent scarce or better spent elsewhere
Integration complexity	Low and self-contained	High and connection-heavy
Maintenance commitment	Willing to own it for life	Prefer the vendor to carry it
Risk tolerance	High; overrun is survivable	Low; predictability matters
Data control and lock-in	Full control is essential	Vendor terms are acceptable

Read the completed rubric as a pattern, not a tally. If most dimensions fall on the buying side, as they do for the great majority of supply chain capabilities, buy, and spend the saved effort where it differentiates. A build should be considered only when strategic differentiation is clearly on the building side and it is joined by durable senior talent and a scope that can be contained to a thin layer. A high score on differentiation alone is not sufficient, because a differentiating capability an organization cannot afford to own and maintain is still a poor thing to build. The rubric earns its keep precisely when it stops a build that a single strong conviction would otherwise have launched.

A worked scoring example

The rubric is most useful when it is applied to a concrete case, because scoring forces the vague intuition that something is special into an explicit judgment that can be challenged. Consider a mid-sized distributor deciding whether to replace its aging warehouse management system with a packaged platform or to rebuild it in-house, now that assistants have made the rebuild feel affordable.

On differentiation, the honest score is low. Customers do not choose the distributor because of how its warehouse software works; they choose it for price, availability, and service, which the software enables but does not embody. On vendor maturity, the score is high: the warehouse management market is deep, competitive, and commoditized, with multiple credible options at every scale. On these two dimensions alone, the framework already points firmly toward buying, because a non-differentiating capability served by a mature market is the textbook definition of context.

The remaining dimensions confirm rather than complicate the verdict. On lifetime cost, a rebuild commits the distributor to funding maintenance, security, and compliance for the system's entire life, against a subscription that spreads those costs across the vendor's whole customer base. On talent, the distributor would have to hire and retain scarce engineers to own a system that is not its business, and to preserve the domain knowledge those engineers depend on. On risk, the base rates for large in-house builds are poor, while the packaged option has been proven across hundreds of comparable operations. Every dimension points the same way, and the assistant that made the rebuild feel cheap has changed none of them, because it lowered only the cost of writing code and left the cost of owning the system untouched.

The value of scoring the case explicitly is that it converts a decision that felt like a matter of ambition and pride into a matter of evidence. Had a single dimension pointed the other way, a genuine differentiation, a thin or predatory vendor market, an unusually short horizon, the framework would have surfaced it as the fact the decision turned on, and the conversation could have focused there. When every dimension agrees, as it usually does for standard supply chain functions, the discipline is simply to accept the answer the rubric gives rather than to override it with the feeling that this time is different.

A second worked example: when the score says build

For balance, consider a case where the rubric points the other way, because a framework that only ever said buy would be a prejudice rather than a discipline. Imagine a specialty logistics company whose competitive advantage is a routing and consolidation method its founders spent a decade refining, a way of combining partial loads across a particular network that no competitor has matched and no transportation-management vendor implements, because it is peculiar to this company's lanes and customers.

On differentiation, the score is high and genuine: customers choose this company specifically because its method delivers lower cost and faster transit than anyone else, and rivals have tried and failed to copy it. On vendor maturity, the score favors building, not because the transportation-management market is thin, it is not, but because the specific capability at issue, the proprietary consolidation logic, is precisely the slice no product supplies. Here the two dimensions that pointed toward buying in the first example point toward building, and for the right reason: this is core, and it is unserved.

The remaining dimensions decide how much to build, not whether. Lifetime cost still matters, so the company buys a mature transportation-management core for the commodity work, the tendering, the tracking, the settlement, and builds only the consolidation engine on top, connected through the platform's interfaces. It confirms that it has the durable senior talent to own that engine for its whole life, because a differentiator no one can maintain is a liability in waiting, and it keeps the built layer as thin as the advantage requires. The result is the third path in miniature: buy the vast commodity core, build the small differentiating edge, and point whatever AI assistance is used at the edge rather than the core. The rubric did not say build everything; it said build the one thing that is genuinely core and buy the rest, which is exactly the answer a disciplined framework should produce when the facts support a build.

Anti-patterns the rubric alone will not catch

A scoring rubric structures a decision but does not immunize it, because the most dangerous errors are the ones that corrupt the inputs rather than the arithmetic. A handful of anti-patterns recur often enough to be worth watching for by name.

- **Differentiation inflation.** The most common failure is scoring an ordinary capability as core because the organization is proud of it. Guard against it by asking whether a customer has ever chosen you because of this capability, and by requiring evidence rather than conviction.
- **The AI business case.** A build justified chiefly by the claim that AI has made it cheap has staked the decision on the one variable the evidence least supports. Strip the AI assumption out and ask whether the build still makes sense; if it survives only with the assumption, it does not survive.
- **The platform that becomes a product.** A build scoped as a thin edge has a way of accreting responsibilities until it is a foundational system no one decided to build. Name the boundary explicitly, and treat every proposed expansion of it as a new decision, not a detail.
- **The sunk-cost continuation.** Once a build is underway, every dollar already spent argues for the next one, and the question quietly changes from whether to build this to whether to abandon what has been started. Revisit builds against the same rubric that launched them, and be willing to stop, as Lidl eventually was, though only after half a billion euros.
- **The reversibility blind spot.** A decision that can be undone cheaply deserves less deliberation than one that cannot, yet the rubric scores them alike. Weight the irreversible choice, the core system, the deep customization, the data migration, more heavily, because the cost of being wrong is not symmetric.

None of these anti-patterns shows up as a low score, because each one operates on the judgment behind the score rather than the score itself. The rubric's discipline is necessary but not sufficient; it must be applied by people willing to challenge the inputs, to strip out flattering assumptions, and to reach the uncomfortable answer when the honest one points there.

Stage five: govern the decision and revisit it

A build-versus-buy decision is not a single event; it is the beginning of a commitment that will be tested repeatedly as circumstances change. The final stage of the framework is to govern the decision after it is made and to revisit it deliberately, because the conditions that justified an answer today may not hold in three years, and organizations rarely notice the moment when an old choice stops making sense.

For anything you chose to build, governance means treating it as a product with an owner, a budget, and a lifecycle rather than a project that ended at launch. Someone must be accountable for its maintenance, its security posture, its documentation, and the knowledge required to change it safely. The standing cost of ownership should be named in the budget every year, so that the true, ongoing price of the build stays visible rather than disappearing into general engineering overhead where no one can see it accumulating.

For anything you chose to buy, governance means managing the relationship and the risk that comes with it: tracking the vendor's roadmap against your needs, avoiding the deep customizations that quietly

recreate the lock-in and maintenance burden you were trying to escape, maintaining a credible exit path, and periodically confirming that the market still offers the competition that keeps the vendor honest. Buying is not the absence of a commitment; it is a different commitment, and it rewards attention.

Above both sits the discipline of revisiting the classification itself. A capability that was context can become core if the business changes its strategy, and a capability that was core can become context as vendors mature and commoditize what was once distinctive. A scheduled review, tied to renewal dates and to strategic planning, keeps the portfolio of build-and-buy decisions aligned with the business as it actually is rather than as it was when each decision was first made. The framework, in other words, is not a gate you pass through once. It is a lens you keep, and the organizations that use it well are the ones that are willing to reach a different answer when the facts change.

Run a pre-mortem before you commit

One practice deserves a place in any build decision because it is cheap, fast, and unusually effective at surfacing the risks a business case hides: the pre-mortem. The exercise is simple. Before committing, gather the people involved and ask them to imagine that it is two years from now and the build has failed expensively. Then ask each of them, independently, to write down why.

The value of the pre-mortem is that it licenses the pessimism the momentum of a decision normally suppresses. In the ordinary course, doubts feel disloyal once a direction is set, and the people who see the risks most clearly are the least likely to voice them. By stipulating the failure and asking only for its causes, the pre-mortem turns dissent into a requested contribution, and it reliably produces the specific, concrete risks, the data migration no one owns, the integration no one scoped, the key engineer everyone assumes will stay, that a status-quo review would never surface.

Read against this guide, a pre-mortem for a build tends to reproduce the failure modes of Section 13 in the participants' own words, which is exactly its usefulness. If the imagined post-mortem reads like Target's data disaster, or Lidl's customization spiral, or Nike's rushed cutover, the risks are not hypothetical; they are the base rate, described in advance by the people closest to the work. A build whose pre-mortem surfaces a dozen plausible causes of failure, none of which has an owner or a mitigation, is a build the pre-mortem has just talked you out of, cheaply, before the money was spent.

The pre-mortem does not replace the rubric or the readiness assessment; it complements them by attacking the weakness they share, the tendency of a structured process to feel more certain than the facts warrant. A decision that survives a rubric, a readiness assessment, and an honest pre-mortem is about as well-examined as a build decision can be, and one that cannot survive all three is telling you something you would be wise to hear before the invoice arrives.

Triggers that should prompt a revisit

Stage five committed you to revisiting the decision, but a commitment to revisit periodically tends to mean never. It is better to name in advance the specific events that should reopen a build-versus-buy decision, so the review happens by trigger rather than by good intention.

- **A capable product appears where none did.** The most common reason a build decision goes stale is that the market catches up; a capability you built because nothing could be bought should be reconsidered the moment something can.
- **The differentiator stops differentiating.** If customers no longer choose you for the capability you built, it has become context, and context should be bought, not maintained.
- **The maintenance burden outgrows its value.** When the standing cost of owning a built system exceeds what it earns, the honest move is to migrate to a bought alternative, however reluctant the sunk cost makes you.
- **The team that owned it disperses.** A build that loses its owners without replacing them has become a black box, and a black box is a liability to be retired, not an asset to be preserved.
- **The regulatory ground shifts.** A change that turns your system into a regulated one, per Section 10, can alter the economics enough to reopen the decision entirely.

A decision reviewed against triggers like these stays current with the facts rather than frozen at the moment it was made. The discipline of restraint applies not only to the first decision but to every one that follows, and the willingness to reverse a build when the facts change is as much a part of good judgment as the reluctance to start one.

15 Organizational readiness: the capacity to own

Every argument in this guide eventually reduces to a single, uncomfortable question that the excitement around AI tends to skip: not whether you can build the thing, but whether you can own it. Building is an event; owning is a condition, and it is the condition that determines whether a build becomes an asset or a liability. Before committing to any build, an organization should assess its own capacity to carry the system for its entire life, honestly and in advance.

Readiness is not a matter of ambition or engineering talent alone. It is a matter of whether the organization has the structures, the people, and the discipline to treat a piece of software as a permanent responsibility rather than a project that ends at launch. Most organizations that regret a build did not lack the ability to write it; they lacked the capacity to sustain it, and they discovered the gap only after the system was load-bearing and the original team had moved on.

The three questions of readiness

The first question is about ownership. Is there a named owner, a person or team accountable for this system for years, not a project sponsor who disbands at launch? Software without a standing owner degrades by default, because no one is responsible for the unglamorous, continuous work of keeping it current, secure, and understood. If you cannot name the owner and fund them permanently, you are not ready to build.

The second question is about knowledge. Can the domain understanding and the design intent behind the system be captured and preserved, so that the system survives the departure of the individuals who created it? A build that lives only in the heads of two engineers is one resignation away from becoming a black box. Readiness means having the practices, documentation, review, pairing, tests that encode intent, that turn private knowledge into institutional knowledge before it walks out the door.

The third question is about capacity over time. Can the organization commit the standing engineering capacity that maintenance requires, every year, in competition with everything else that will demand those same engineers? Maintenance is not a one-time cost; it is a permanent claim on the scarcest resource the company has. If that claim will always lose to the next urgent priority, the system will rot, and the honest conclusion is to buy rather than to build something you cannot afford to keep.

The signals that you are not ready

Certain signals reliably predict that a build will become a burden, and they are visible before a line of code is written. The first is the absence of a maintenance budget in the business case. A proposal that quotes only the cost to build, with no line for the years of ownership that follow, is not a plan; it is an underestimate, and the missing line is usually the largest one.

The second signal is a dependence on heroes. If the plan relies on one or two exceptional individuals, and has no answer for what happens when they leave, it is not a plan the organization can sustain; it is a bet on the tenure of specific people. The third signal is a history of unfinished internal systems, the graveyard of half-built tools that every long-lived organization accumulates. A company that has

repeatedly failed to sustain its own builds should treat that record as evidence, not as bad luck to be overcome by trying harder this time.

The point of naming these signals is not to discourage building, but to make readiness a precondition rather than an afterthought. An organization that can answer the three questions convincingly, and that shows none of the warning signals, is genuinely equipped to own what it builds, and for a true point of differentiation it should. An organization that cannot should recognize that the constraint is real, that AI has not removed it, and that buying is not a failure of nerve but a correct reading of its own capacity.

A maturity model for the capacity to own

The three questions admit of degrees rather than simple yes-or-no answers, and it helps to see readiness as a spectrum. The five levels below describe how an organization's capacity to own software typically matures, and where each level's build-versus-buy default should sit.

Readiness level	What it looks like	Sensible build-versus-buy default
1 — Ad hoc	No standing owners; builds live in individuals' heads; no maintenance budgets	Buy almost everything; build nothing load-bearing
2 — Reactive	Some ownership, but maintenance competes for attention and usually loses; documentation thin	Buy the core; build only trivial, disposable tools
3 — Defined	Named owners and funded maintenance for key systems; basic testing and documentation exist	Buy the core; build a thin, well-bounded edge where it truly differentiates
4 — Managed	Standing platform practices, review, testing, and security gates; knowledge survives departures	Build differentiating capabilities deliberately; point AI at the edge, with guardrails
5 — Optimizing	Software ownership is a core competence; the organization runs internal platforms at scale	Build where it is genuinely core, at scale, as a strategic choice

A maturity model for the capacity to own software. The right build-versus-buy default depends on where an organization actually sits, not where it aspires to be.

Two things follow from placing yourself on this ladder honestly. The first is that the right build-versus-buy default is not universal; it depends on where you actually sit. An organization at level two that adopts a level-four posture, building differentiating systems it cannot sustain, is writing the first chapter of one of the failures in Section 13. The gap between ambition and capacity is exactly where builds go to die.

The second is that the ladder is climbable, and climbing it is often a better investment than any single build. The practices that move an organization from reactive to managed, standing ownership, funded maintenance, real testing, and documentation that outlives its authors, are the same practices that determine, per the evidence in Section 9, whether AI assistance helps or harms. Building the capacity to own is a prerequisite for building anything worth owning, and it is the one investment that pays off regardless of which side of the build-versus-buy line any particular decision falls on.

The adoption problem the failures share

Return, finally, to the failures of Section 13, and notice what almost all of them share beneath their specific causes. Nike's planners did not fully understand the system they were handed. Hershey went live without the testing and training that adoption requires. Target's staff entered data no one had told them had to be correct. Lidl's people never trusted or embraced the platform. Beneath the technical diagnoses sits a human one: the systems were not adopted, and a system that is bought or built but not adopted is money spent for nothing.

This matters for build versus buy in a way that is easy to miss. The change-management burden, teaching an organization to work in a new way, is largely independent of whether the system was bought or built; it falls due either way, and it is frequently the larger and more neglected cost. But building adds to that burden rather than reducing it. A bought system arrives with training materials, documentation, a user community, and a vendor whose business depends on its customers succeeding with it. A built system arrives with none of these; the organization must create the training, write the documentation, and sustain the support entirely on its own, on top of building and maintaining the software itself.

The implication is not that adoption is a reason to buy, though it is one more weight on that side of the scale. It is that the adoption cost belongs in the decision explicitly, on both sides, and that an organization with a weak record of change management should treat that weakness the way it treats a weak record of maintenance: as a real constraint that argues for buying the commodity, leaning on the vendor's adoption machinery, and reserving its scarce capacity for change for the few differentiating systems where the effort is genuinely worth it.

The most sophisticated build-versus-buy analysis in the world is worthless if the resulting system sits unused. A leader who remembers that the failures in this guide were, at bottom, failures of people adopting change, and who plans for adoption as seriously as for architecture, has internalized the lesson those expensive cases were paid to teach.

16 Vendor selection and the contract that protects you

If the presumption for context is to buy, then choosing well and contracting well become the disciplines that determine whether buying delivers on its promise. A poor selection or a careless contract can recreate the very problems, lock-in, spiraling cost, dependence on a single party, that building was supposed to avoid. Buying is not the end of diligence; it is where a different kind of diligence begins.

Running an evaluation that predicts reality

The purpose of a vendor evaluation is not to admire demonstrations but to predict how the software will behave in your operation, under your data, at your scale. The single most useful step is to insist on a proof of concept using your own data and your own highest-volume, most awkward scenarios, rather than the vendor's curated examples. The same gap between a demonstration and a product that undoes in-house prototypes applies to vendor demos, and the antidote is the same: test the unhappy paths before you sign.

A serious evaluation also weighs the things that will matter for a decade, not just the features that impress in a quarter. Integration with the systems you already run, the depth and responsiveness of support, the credibility of the product roadmap, the vendor's financial stability, and the health of the surrounding ecosystem of implementers and talent all predict the lived experience of ownership more reliably than a feature checklist. Reference calls with customers who resemble you, and who have run the software for years rather than months, are worth more than any scripted demonstration.

Finally, a disciplined evaluation prices the genuine alternative on a lifetime basis. The comparison that matters is not license fee against build salaries, but total cost of ownership against total cost of ownership: implementation, subscription, integration, and internal effort for the bought option, weighed against build, maintenance, security, and the probability-weighted cost of failure for the in-house option. Making that comparison explicit is what keeps the decision honest, and it is the same lifetime discipline the framework applies everywhere else.

The terms that matter most

A good contract protects the buyer against the risks that buying introduces, and a handful of terms matter far more than the rest. The first is exit and data portability: the right to retrieve your data in a usable form, and a realistic path to leave, are what preserve the competitive pressure that keeps a vendor honest for years. A relationship you cannot leave is a relationship you cannot negotiate, and lock-in is a cost that accrues quietly until the day you try to move.

The second is price predictability. Renewal caps, transparent usage-based pricing, and protection against steep increases guard against the familiar pattern in which an attractive initial price rises sharply once the software is embedded and switching has become expensive. The third is a clear allocation of responsibility for security, availability, and compliance, expressed as commitments with consequences rather than aspirations, so that the shared investment you are renting actually carries the obligations you are relying on it to carry.

None of this eliminates the fundamental trade of buying, which is that you accept a capability shaped for the average of the market rather than for you alone. But good selection and good terms convert that trade from a source of hidden risk into a managed relationship, and they preserve the very advantages, shared cost, absorbed risk, continuous improvement, that made buying the right presumption in the first place. The discipline of buying well is, in the end, the same discipline as building well: name the lifetime cost, keep your options open, and revisit the decision as the facts change.

The implementation itself: where good decisions still fail

A sound build-versus-buy decision, a well-chosen vendor, and a well-drafted contract can still end in the failures of Section 13, because none of them is the implementation, and the implementation is where the money is actually won or lost. Every case in this guide, whether the software was bought or built, failed in the same handful of ways during implementation, and the disciplines that prevent those failures are worth stating plainly, because they apply regardless of which side of the build-versus-buy line the decision fell on.

The first discipline is phasing. Nearly every catastrophic failure in Section 13 shared a big-bang cutover, everything old switched off and everything new switched on at once, on a single date and often an unforgiving one. A phased rollout, by region, by function, or by business unit, contains the blast radius of any problem and creates the chance to learn before the next phase. The big bang is seductive because it promises to be over quickly; what it delivers instead is a failure with no safety net. Phase what you can, and never cut over everything at once if there is any way to avoid it.

The second is timing. Hershey went live weeks before its peak season; National Grid went live days after a hurricane. A go-live is a business event, and scheduling it into or beside the busiest, most fragile stretch of the year converts ordinary teething problems into existential ones. Choose a quiet window, leave real slack before the next peak, and treat the go-live date as a decision made by the business rather than a milestone imposed by a project plan.

The third is testing. In case after case, testing was the phase compressed to hit a date, and the defects it would have caught surfaced in production instead. Testing must replicate real volumes and real data, not curated happy paths, because what passes in a quiet test environment routinely fails under production load. The time spent testing is not a delay to the value; it is the insurance that the value arrives at all.

The fourth is data. Target Canada's collapse was, at bottom, a data failure: tens of thousands of records entered at speed, without validation, by people never told that accuracy was essential. Data migration deserves first-class attention, profiling, cleansing, validation on entry, and a schedule that treats the discovery of bad data as an expected event rather than a launch-day surprise. A system fed bad data will fail however good the software, and neither a vendor nor a build can rescue it.

The fifth is training and adoption, the human discipline of Section 15, without which even a technically flawless implementation sits unused. These five, phasing, timing, testing, data, and adoption, are not exotic, and that is precisely the point: the expensive failures were not undone by unknowable risks but by known disciplines skipped under pressure. Buying does not exempt an organization from them, and

building multiplies the burden of every one. Whichever path a decision takes, the implementation is where it is finally kept or lost, and it deserves the same rigor as the decision that preceded it.

Using AI to run a better buy

If AI's value on the build side is smaller and more fragile than the marketing claims, its value on the buy side deserves a mention, because the same tools that struggle to build a production system are genuinely useful in running a rigorous purchase.

The work of buying well, drafting requirements, comparing vendor responses, analyzing contracts, summarizing reference calls, is bounded, document-heavy, and low-stakes in the sense that a human reviews every output before it matters. That is precisely the profile of work AI does well. A team can use it to turn messy internal requirements into a structured request, to compare a dozen vendor responses against a rubric, to surface the unusual clauses in a contract for a lawyer's attention, or to draft the first version of an evaluation scorecard. None of these replaces the human judgment the decision requires, but each removes friction from the disciplined process this guide recommends, and each is a use of AI whose weaknesses are contained by the review that surrounds it.

There is a quiet irony worth naming. The most reliable way for many organizations to get value from AI is not to point it at building the systems they should be buying, but to point it at buying them well. The tool that makes a mediocre build feel deceptively easy can also make a rigorous purchase genuinely easier, and the second is where its help is real.

17 Conclusion: the discipline of restraint

The arrival of capable AI coding tools is a real and welcome development, and it would be as foolish to dismiss it as to be swept away by it. The measured conclusion is neither. AI has lowered the cost of writing code, which is a meaningful change, and it has left the cost of owning software essentially where it was, which is the change that did not happen but that the easy-button narrative assumes. Building software has always been mostly about the parts that surround the code, understanding the domain, designing the system, integrating with what exists, getting the data right, proving correctness, securing the result, and maintaining the whole for years, and those parts remain stubbornly, expensively human. A faster way to write code does not move the base rate of project failure, does not shrink the maintenance iceberg, and does not remove the forever commitment that a build entails.

For supply chain leaders in particular, the implication is clear and, in a sense, unchanged. The commercial market is deep, the domain is complex, the integration burden is heavy, and the failure history is a warning, all of which point toward buying the mature core and reserving building for the thin layer that sets you apart. AI belongs in that thin layer, at the edge, accelerating the integrations and extensions where its speed helps and its weaknesses are contained, not at the core, where the stakes of correctness and security and maintenance are highest. And the buy side is itself being transformed by AI, as vendors invest at a scale no single organization can match, which strengthens rather than weakens the case to buy the commodity.

The deepest discipline in technology decisions has always been restraint: the willingness to buy what does not set you apart so that you can pour your scarce talent and attention into what does. AI does not relax that discipline. It raises the stakes of getting it right, because it makes building feel easy while leaving building hard, and it tempts organizations to construct systems they will struggle to understand and cannot afford to maintain. The leaders who thrive will be the ones who resist the temptation to build everything simply because building has become faster to start, and who apply, with clear eyes about what AI has and has not changed, the oldest and best question in enterprise technology: is this the thing that makes us different, and can we own it for as long as we will depend on it? Where the answer is yes, build, and build a thin, well-guarded layer with AI's help. Where the answer is no, and for most of the supply chain stack it is no, buy the best that exists, integrate it well, and spend your building energy where it will actually set you apart.

The discipline of restraint is unglamorous, and that is precisely why it is valuable. It asks a leader to resist the most exciting version of a decision, the bespoke system built to exact specification, in favor of the version that is more likely to work, cost less over its life, and free the organization's scarce talent for the few things that genuinely set it apart. Restraint is not caution for its own sake; it is the deliberate concentration of finite resources on the small number of bets that can actually change the company's position.

Artificial intelligence has not repealed this logic. It has made the first draft of software cheaper and left the cost of owning software almost exactly where it was, which means the temptation to build has grown while the underlying economics have not. The leaders who navigate this moment well will be the

ones who hold both facts in mind at once: that the tools are genuinely useful and worth adopting, and that they change the answer to almost none of the questions that actually decide whether to build or buy.

The practical posture that follows is simple to state and hard to live by. Buy the commodity, because a shared investment you rent will almost always beat a solitary investment you own. Build only the difference, and keep the difference small enough to own for as long as it matters. Use AI where the work is bounded and the judgment remains yours, and govern its output with more discipline, not less. And revisit every decision as the market and the strategy change, because the willingness to reach a different answer when the facts change is the last and most important form of restraint.

Principles to carry forward

If the argument of this guide reduces to a handful of principles a leader can carry into the next decision, they are these.

- **Buy the commodity; build the differentiator.** The whole framework is an elaboration of this single line. Most capabilities are context to be bought; reserve building for the thin core that genuinely sets you apart.
- **Price the lifetime, not the build.** The cost that matters is the whole-life cost, and it is dominated by the lines the build estimate omits. Add them back before you decide.
- **AI changed the tip, not the iceberg.** AI lowers the cost of writing code, which was never the expensive part, and leaves the maintenance, integration, security, and ownership that dominate a build's life untouched.
- **The demonstration is not the product.** A working demo proves an idea is worth doing properly; it is not the thing itself, and the gap between the two is where the cost lives.
- **Owning is harder than building.** Building is an event; owning is a permanent condition. If you cannot name the owner and fund the maintenance, you are not ready to build.
- **Customization past a point is building in disguise.** Bending a bought system far enough inherits a build's costs while forfeiting the vendor's guardrails, as the most expensive failures in this guide attest.
- **The implementation is where decisions are kept or lost.** Phasing, timing, testing, data, and adoption undo more good decisions than the decisions themselves. Respect them.
- **Classify honestly, and revisit.** The columns shift with technology and strategy; the discipline is to ask the differentiation question anew rather than to defend yesterday's answer.

None of these is new, and that is their strength. The temptations change, the tools change, and the marketing changes, but the discipline that separates a wise technology decision from an expensive one has been stable for decades, and it rewards restraint. The easy button is a mirage. The discipline is not.

Appendix A The build-versus-buy scoring worksheet

The worksheet below turns the framework into a single page you can complete for any capability under consideration. Score each dimension from 0 to 4, where 0 argues strongly for buying and 4 argues strongly for building, then read the total against the guidance beneath the table. This worksheet condenses the fuller rubric in Section 14 into the six dimensions that most often decide the outcome; use the Section 14 rubric when a case is close and this one-page version to frame the first conversation. The value of the exercise is less in the arithmetic than in forcing an explicit, defensible judgment on each dimension.

Dimension	The question to answer	Score (0 = buy · 4 = build)
Differentiation	Would a customer pay a premium, or need years to replicate this?	0 if no · 4 if clearly yes
Vendor maturity	How deep and competitive is the market of alternatives?	0 if deep market · 4 if none exists
Lifetime cost	Does building win on total cost of ownership, not build cost alone?	0 if buy wins · 4 if build wins
Talent & ownership	Can you fund an owner and maintenance for the system's whole life?	0 if no · 4 if yes
Risk tolerance	Can you absorb the base-rate risk of a large custom build?	0 if no · 4 if yes
Integration need	Is this connective glue between systems that no vendor sells?	0 if a vendor sells it · 4 if unique glue

A total near the low end, roughly 0 to 8, is a clear buy. A total near the high end, roughly 20 to 24, may justify a build, or a thin build at the edge. Scores in the middle point to the third path: buy the core and build only the differentiating layer. Treat the number as a prompt for discussion rather than a verdict, because a single decisive dimension, a genuine differentiation or the total absence of a credible vendor, can rightly override the sum.

Appendix B A guardrail checklist for AI-assisted development

If you build, or use AI heavily within a build, the guardrails below convert the speed of assistants into an asset rather than a hidden liability. They are deliberately practical, and most cost little beyond the discipline to apply them consistently on every change.

- Treat AI output as a draft from an unvetted contributor: review every line, and never merge code that no human on the team understands.
- Require human sign-off on architecture and interfaces. Let assistants fill in implementation; do not let them decide the design.
- Keep tests first-class. Generated code that passes a weak suite is a false sense of safety; strong tests are what make speed safe.
- Verify that every third-party dependency an assistant suggests actually exists and is trustworthy, to guard against hallucinated and squatted packages.
- Run automated security and dependency scanning on AI-generated code, and treat findings as blocking rather than advisory.
- Watch duplication and churn as leading indicators of decay, and budget time for refactoring, not only for generating the next thing.
- Preserve intent: document why non-obvious code exists, so the knowledge survives the person, and the assistant, that produced it.
- Assign a standing owner and a maintenance budget before launch, not after. Ownership is the precondition, not the afterthought.

Appendix C Glossary of key terms

The terms below recur throughout this guide. The definitions are deliberately practical rather than exhaustive, and are intended as a quick reference for readers sharing the framework across a team.

Core versus context — Geoffrey Moore's distinction between activities that create durable, differentiating advantage (core) and everything else that is necessary but does not differentiate (context). Pour resources into core; minimize, standardize, or buy context.

Total cost of ownership — The full lifetime cost of a system, including build, integration, hosting, security, compliance, and years of maintenance, as opposed to the initial build cost alone. It is the number that should drive a build-versus-buy decision.

Maintenance — The continuous work of keeping software running and current after launch. It typically accounts for the large majority of a system's lifetime cost and comes in adaptive, corrective, and perfective forms.

Adaptive, corrective, and perfective maintenance — Adaptive keeps a system working as its environment changes; corrective diagnoses and repairs defects found in production; perfective makes the enhancements the business requests after launch. All three are permanent obligations of ownership.

Technical debt — The accumulated future cost of choices that were expedient in the moment, such as skipped tests, duplication, or code no one fully understands. Like financial debt, it accrues interest, paid as slower and riskier future change.

Vibe coding — Building software by describing what you want to an AI assistant and accepting its output, often without fully understanding it. It produces convincing demonstrations quickly and tends to hide the debt that surfaces when the demo must become a product.

The seventy percent problem — The pattern in which AI assistants get a system most of the way to done quickly, while the remaining portion, the hard integration, edge cases, security, and reliability, consumes most of the real effort and judgment.

Base rate — The historical average outcome for a class of decisions, such as the share of large custom builds that run late, over budget, or deliver less value than promised. Base rates are the antidote to the belief that this project will be the exception.

Lock-in — The cost and difficulty of switching away from a vendor or a system once it is embedded. Managed through data portability, exit rights, and avoiding deep customization, it is what preserves negotiating leverage over time.

The third path — The disciplined middle ground of buying the commodity core and building only the thin, differentiating edge on top of it, keeping the build small enough to own for as long as it matters.

Slopsquatting and hallucinated dependencies — A supply-chain risk in which AI assistants suggest software packages that do not exist, which attackers then create under those exact names so that unwary developers install malicious code. It is a new attack surface created by generated code.

Agentic AI — AI that plans and carries out multi-step tasks with limited human intervention, increasingly embedded in enterprise software. It is a leading edge of what vendors now ship, and part of why the case to buy is strengthening.

Big bang versus phased rollout — Two approaches to going live. A big bang switches everything over at once; a phased rollout proceeds in stages, containing the blast radius of any problem. Nearly every catastrophic failure in this guide shared a big-bang cutover.

Requirements gap — The distance between what a business wants and what a packaged product does as designed. A small gap means little customization; a large gap is a signal to change the process rather than the product.

Provider and deployer — The two roles the EU AI Act assigns along the AI supply chain. A provider builds or materially alters a system and carries the heavy obligations; a deployer merely uses one and carries lighter duties. Building, or heavily customizing, can turn a deployer into a provider.

Conformity assessment — The process of demonstrating that a high-risk AI system meets its legal requirements, a provider obligation that a buyer inherits the moment it becomes a provider by building or substantially modifying a system.

Composable architecture (MACH) — A design approach, associated with microservices, APIs, cloud-native delivery, and headless front ends, that lets an organization buy a robust core and build differentiating extensions on top through well-defined interfaces.

Absorptive capacity — The organizational ability to take in new code safely, through platform practices, testing, and review. It is what determines whether AI assistance improves delivery or destabilizes it.

Batch size — The amount of change shipped at once. Larger batches are harder to review, test, and unwind, and AI's tendency to enlarge them is a leading explanation for its measured effect on delivery stability.

Black-swan project — A large project that overruns catastrophically, by 200 to 400 percent, threatening the organization undertaking it. Research suggests roughly one in six large IT projects becomes one.

System integrator — An outside firm engaged to implement software. Its incentives do not always align with the buyer's, which is why internal ownership and a well-drafted contract matter.

Change management — The discipline of helping an organization adopt a new way of working. Largely independent of the build-versus-buy choice, it is frequently the larger and more neglected cost, and its absence underlies most implementation failures.

Appendix D A request-for-proposal and evaluation scorecard

The quality of a build-versus-buy decision, once it points toward buy, is only as good as the evaluation that follows. This appendix distills Section 16 into a request-for-proposal outline and a weighted scorecard a team can adapt directly. The aim is an evaluation that predicts reality rather than one that rewards the best demonstration.

What to put in the request

- **Requirements tied to real workflows.** Describe the actual processes the system must support, in your own language, and ask the vendor to show how the product handles each without customization. A requirement stated as a workflow is far harder to answer with a slide than one stated as a feature.
- **Integration and extensibility.** List the systems the product must connect to and ask how, through which interfaces, and with what limits. The answer determines whether you can later build the thin differentiating edge on top of the bought core.
- **Data migration.** Specify the volume, shape, and quality of the data to be migrated, and ask the vendor to describe the tools, the validation, and the failure modes. Every failure in Section 13 turned on data; treat it as a first-class part of the evaluation.
- **Security and regulatory posture.** Ask for security certifications, breach-notification commitments, and, for AI features, the vendor's role and yours under applicable regulation, including who is the provider and who is the deployer.
- **Implementation plan and references.** Require a phased plan with milestones and named references at comparable scale, and insist on speaking to a customer whose rollout went badly, not only ones that went well.
- **Total cost of ownership.** Ask for a five- to seven-year cost model that includes licenses, implementation, integration, support, and the price of the changes you already know you will need, rather than the first-year license alone.
- **Portability and exit.** Ask, before you sign, how you would leave: what formats your data exports in, what transition assistance is offered, and what it would cost. The answer is easiest to get while you still hold the leverage of an unsigned contract.

A weighted scorecard

Score each finalist against weighted criteria rather than trusting a gestalt impression, because a structured score forces the evaluation onto the dimensions that predict success rather than the ones that demonstrate well. The weights below are a starting point; tune them to the decision, but agree on them before you see the demonstrations, not after.

Criterion	Weight	What a strong answer looks like
Fit to core workflows	25%	The product supports your real processes close to standard, with little customization

Criterion	Weight	What a strong answer looks like
Integration and extensibility	15%	Open, documented interfaces that let you build the differentiating edge on top
Implementation risk and references	15%	A phased plan and credible references at your scale, including a hard one
Total cost of ownership (5-7 yr)	15%	A complete lifetime model, not a first-year license, with the known changes priced in
Security and compliance	10%	Current certifications, clear breach terms, and a defined role under AI regulation
Vendor viability	10%	Financial stability and a roadmap that does not depend on your customization
Data portability and exit	10%	Open export formats and defined, priced transition assistance

A weighted evaluation scorecard. Agree the weights before the demonstrations, and score the dimensions that predict success rather than the ones that demonstrate well.

A disciplined scorecard will sometimes point at the less impressive demonstration, and that is the point. The vendor who answers the hard integration and exit questions honestly is usually a safer partner than the one whose demonstration was flawless and whose contract later turns out to be a cage.

Appendix E The contract terms that protect a buyer

A good product on a bad contract is still a bad outcome. Section 16 argued that the terms matter as much as the technology; this appendix collects the ones worth insisting on. None is exotic, but each closes a gap through which value routinely leaks after signature, once leverage has passed to the vendor.

- **Pricing and renewal protection.** Cap renewal increases, fix the price of the modules and seats you will predictably add, and avoid auto-renewal terms that quietly remove your leverage.
- **Service levels and remedies.** Define availability, support response, and performance commitments, and attach real remedies, service credits or termination rights, so that a service level is a promise rather than a sentiment.
- **Data ownership and portability.** State plainly that your data is yours, specify the open formats it exports in, and require export on demand, not only at termination.
- **Security and breach obligations.** Require current certifications, defined breach-notification timelines, and cooperation obligations, and align them with your own regulatory duties as a deployer.
- **AI-specific terms.** Where the product uses AI, allocate the provider and deployer roles explicitly, require notice of material model changes, obtain warranties on training-data rights, and secure indemnity for intellectual-property and output-related claims.
- **Implementation milestones and acceptance.** Tie payment to accepted milestones against written acceptance criteria, so that a troubled implementation is the vendor's problem to fix rather than yours to pay for.
- **Termination and transition assistance.** Secure the right to terminate for sustained failure, and a defined, priced obligation on the vendor to help you migrate off, negotiated now while you still can.
- **Liability and indemnity.** Ensure the liability cap and indemnities are proportionate to the damage a failure of this system could actually do to your business, not to the size of the annual fee.
- **Audit and reporting rights.** Reserve the right to verify security, usage, and billing, so that the vendor's assurances can be checked rather than merely trusted.

Most of these terms cost nothing to obtain before signature and are close to impossible to obtain after it. The hour spent negotiating the exit clause is the cheapest insurance a buyer ever buys, because it is purchased with leverage the buyer will never have again.

Appendix F A migration and exit-planning checklist

Lock-in is not a single event but the slow accumulation of dependencies that make leaving harder than staying, even when staying no longer serves you. The way to keep the exit affordable is to plan it before you enter, and to keep planning it while you operate. This checklist applies whether you are migrating onto a bought platform or preserving the option to migrate off one.

- **Inventory the dependencies.** Maintain a living map of the data, integrations, and workflows that rely on the system, because you cannot leave cleanly what you have not catalogued.
- **Insist on open export.** Ensure your data can be exported, on demand, in documented and open formats, and test that the export is actually complete and usable rather than nominal.
- **Migrate data with respect for its messiness.** Profile data quality early, validate on entry, and schedule the migration so that discovering bad data is a manageable event rather than a launch-day catastrophe, the lesson of every case in Section 13.
- **Keep the differentiating layer decoupled.** Build your custom edge behind the platform's interfaces rather than woven into its internals, so that replacing the core does not mean rebuilding the edge.
- **Preserve a rollback path.** For any major cutover, keep the ability to run in parallel and to revert, and do not decommission the old system until the new one has survived a full business cycle.
- **Negotiate transition assistance in advance.** Secure the vendor's help with an eventual exit as a contractual obligation, priced and defined, while the deal is still being won.
- **Test portability periodically.** Rehearse the exit on a schedule, because a portability right you have never exercised is a right you do not know you actually have.
- **Protect knowledge, not just data.** Document configurations, decisions, and integrations so that the ability to operate, and to leave, does not walk out the door with a departing engineer.

None of this presumes you intend to leave. It presumes that the freedom to leave is what keeps a vendor relationship honest and a build decision reversible. The organizations that suffer most from lock-in are not the ones that chose to buy; they are the ones that bought, or built, without ever planning for the day they might want to change their mind.

Appendix G Answers to the common objections to buying

The case for buying meets the same objections again and again, often raised sincerely and sometimes raised to justify a decision already made. This appendix collects the most common and answers each briefly, so that a team can meet them with reasoning rather than reflex.

- **"Building gives us exactly what we want."** It gives you exactly what you specified, which is not the same thing, and it also gives you everything that follows: the maintenance, the security, the integration, and the obligation to keep it current for as long as it runs. A product gives you most of what you want and none of that burden. Reserve "exactly what we want" for the capability that genuinely differentiates you.
- **"AI makes building cheap now."** AI lowers the cost of writing code, which was never the expensive part of a build. The recurring costs that dominate a build's lifetime, maintenance, integration, security, and ownership, are untouched, and the evidence in Sections 7 through 9 suggests the coding gains themselves are smaller and more fragile than the demonstrations imply.
- **"We are a technology company; building is what we do."** Being able to build is not a reason to build everything, any more than being able to cook is a reason to grow your own wheat. Even the most capable engineering organizations buy the vast commodity remainder and build only their differentiating core; that discipline is what lets them build the core well.
- **"Buying locks us in to a vendor."** Lock-in is a real cost, and it is managed through the contract and exit-planning discipline in Appendices E and F, not avoided by building, which locks you in to yourself: to the specific engineers who understand the system and to the indefinite obligation to staff it. Self-lock-in is the harder kind to escape.
- **"The product does not fit our process."** This is the objection most worth taking seriously, because sometimes it is true and points to a genuine differentiator. Far more often, as Lidl discovered, the fit gap reflects a process worth changing rather than a product worth rejecting, and the disciplined move is to adopt the standard and change the process, not to customize the product into a de facto build.
- **"We will just customize the bought system."** Heavy customization is building wearing a buyer's clothes. Every case in Section 13 that turned on over-customization ended the same way, and the more you bend a product past its design, the more you inherit a build's costs while forfeiting the vendor's guardrails and upgrades.
- **"Our requirements are unique."** Some are; most are not. The test is whether a customer has ever chosen you because of the requirement in question. If the answer is no, it is probably a preference, and preferences are what standard products are designed to accommodate through configuration rather than code.
- **"Vendors are expensive."** Vendors are expensive in a way you can see, a line item you can budget and cap. Builds are expensive in a way you cannot see until later, in maintenance, security, and opportunity cost, and the total reliably exceeds the naive estimate by a wide margin, as Section 6 shows. The visible cost is usually the smaller one.
- **"We can build it faster than a procurement cycle."** You can often build a demonstration faster than you can buy, which is precisely the trap in Section 8: the demonstration is not the product, and the

gap between them is where the time and money actually go. A slow procurement that ends in a working, supported system beats a fast build that ends in an unsupported one.

- **"If we buy, what is our engineering team for?"** For the differentiating edge, where their effort produces advantage a customer will pay for, rather than the commodity core, where their effort merely reproduces what the market already supplies. Buying the commodity is what frees your scarce engineers to build the things that actually matter.

None of these answers is that buying is always right. Each is that the objection, examined, usually points back to the same discipline: buy the commodity, build the differentiator, and be honest about which is which.

Appendix H A readiness self-assessment

This instrument turns Section 15's three questions into a short assessment a team can complete before committing to a build. Score each statement from zero, strongly disagree, to three, strongly agree, then read the total against the guidance that follows.

#	Statement	Score (0–3)
1	There is a named owner who will be accountable for this system for years, and the role is funded.	
2	Maintenance is a line in the business case, sized realistically for the system's whole life.	
3	Design intent and domain knowledge are captured in documentation and tests, not only in people's heads.	
4	We can commit standing engineering capacity to this system every year, against competing priorities.	
5	We have a track record of sustaining the internal systems we have built before.	
6	The capability is one that customers have chosen us for, or plausibly would.	
7	No packaged product fits the need closely enough to adopt with modest configuration.	
8	We have the platform practices, review, testing, and security gates, to absorb new code safely.	
9	We have planned the exit and the data portability before committing.	
10	The decision is reversible, or we have weighted its irreversibility accordingly.	

A readiness self-assessment. Statements 1 through 4 map to Section 15's three questions of ownership, knowledge, and capacity; the remainder test differentiation, fit, absorptive capacity, exit, and reversibility.

Total the scores out of a possible thirty. A total in the upper range, roughly twenty or above, with no zero among the first four statements, indicates genuine readiness to own a differentiating build. A middling total suggests buying the core and confining any build to a thin, well-bounded edge. A low total, or any zero among the first four, is the instrument telling you what the excitement will not: that the constraint is real, that AI has not removed it, and that buying is the correct reading of your own capacity.

The assessment is a mirror, not a gate to be gamed. Its value lies in the honesty of the answers, and the teams that most need to complete it are usually the ones most tempted to skip it. Completed candidly,

it converts the vague confidence that surrounds a build into a small number of concrete questions, and it is far cheaper to fail this assessment on paper than to discover its verdict after the system is load-bearing and the original team has gone.

Appendix I A phased implementation playbook

Whether an organization buys or builds, the implementation follows the same arc, and the failures of Section 13 are, almost without exception, failures to respect it. This playbook lays out the phases and the discipline each demands. It is deliberately generic, because the disciplines are the same for a bought platform and a built system; only the party responsible for each step changes.

Phase one: foundations

- **Name the owner and fund the role.** Before anything technical, name the person or team accountable for this system for its whole life, not merely to launch, and fund them permanently.
- **Write the business case with the maintenance line in it.** Size the whole-life cost, not the first-year cost, so the decision is made against the real number rather than the flattering one.
- **Define success in the business's terms.** Agree, in advance and in writing, what the system must do for the business and how you will know it did.

Phase two: design and configuration

- **Adopt the standard; resist deviation.** For a bought system resist customization, and for a built one resist scope. Every deviation from the standard is a cost you will pay for as long as the system runs.
- **Keep the differentiating layer thin and decoupled.** Build only the edge that differentiates, behind clean interfaces, so the core can be replaced without rebuilding the edge.
- **Document the design intent as you go.** Capture why, not just what, so the system survives the departure of the people who designed it.

Phase three: data migration

- **Profile the data early.** Understand its quality before you depend on it; the discovery of bad data should be an early event, not a launch-day one.
- **Validate on entry.** Build checks that reject bad data at the point of entry rather than letting it propagate through the system.
- **Keep a source of truth.** Record verified data so the work is not redone, and so errors can be traced to their origin.

Phase four: testing

- **Test with real volumes and real data.** What passes on curated happy paths fails under production load; replicate reality before you depend on it.
- **Test the integrations, not just the components.** Most failures live in the connections between systems, which is exactly where component testing does not look.
- **Re-test after every fix.** A fix that is not re-tested is a hope, not a correction.

Phase five: go-live

- **Phase the cutover.** Contain the blast radius; never switch everything at once if there is any way to avoid it.
- **Choose the timing deliberately.** Avoid peak seasons and fragile periods, and leave real slack before the next one.
- **Keep a rollback path.** Preserve the ability to run in parallel and revert until the new system has survived a full business cycle.

Phase six: stabilization and adoption

- **Staff the stabilization period.** Expect problems after go-live and resource their resolution; the weeks after launch are not the time to reassign the team.
- **Invest in training and support.** A system no one can use is a system no one uses; adoption is the last mile where value is realized or lost.
- **Review the decision against reality.** After stabilization, compare the outcome to the business case, and feed what you learn into the next decision.

None of these steps is glamorous, and that is why they are skipped under pressure and why skipping them is so reliably expensive. The playbook is not a substitute for judgment, but it is a defense against the specific, repeated, well-documented ways that good build-versus-buy decisions are squandered in execution.

Appendix J An annotated reading list

For readers who wish to go to the sources, this list gathers the most important evidence behind the guide's argument, grouped by theme, with a note on what each contributes. Full citations appear in the Sources section.

On the base rate of large software projects

- **McKinsey and the University of Oxford (2012).** The large-sample study behind the forty-five-percent-overrun and black-swan figures; the empirical anchor for Section 4.
- **Standish Group, CHAOS research.** The long-running survey of software project outcomes, useful for its trend and instructive for the debate over its methodology.

On what AI does and does not do for developers

- **The GitHub Copilot controlled trial (2023).** The most-cited evidence of a speed-up on a bounded task, and the high-water mark for optimistic claims.
- **METR (2025).** A rigorous randomized study that found experienced developers slower with AI on real tasks; the sharpest counterpoint to the demonstrations.
- **Google DORA (2024).** The delivery-performance evidence on throughput, stability, and the role of platform quality; the basis for Section 9's mechanism.
- **GitClear (2025).** Large-scale evidence on code churn and duplication; the quality side of the story.
- **Osmani (2024).** The essay that named the seventy-percent problem, and the clearest statement of why the last stretch stays hard.

On the security of generated code

- **Stanford (2023).** The study finding that developers with AI assistants wrote less secure code while believing the opposite.
- **Veracode (2025) and the slopsquatting research (2025).** The introduced-vulnerability and hallucinated-dependency evidence behind Section 9's security argument.

On the cost of building and the failures of implementation

- **The enterprise-software post-mortems.** The documented Nike, Hershey, Target Canada, Lidl, National Grid, and Revlon cases behind Section 13, drawn from contemporaneous reporting and later analyses.
- **ScienceSoft and industry maintenance data.** The basis for the maintenance figures behind Section 5's iceberg.

On architecture and regulation

- **The MACH and composable-architecture literature.** The design principles behind the third path in Section 12.

- **The EU AI Act, Regulation 2024/1689, and deployer-obligation analyses.** The regulatory framework behind Section 10.

No single source is decisive, and several are contested; read together, they support the guide's central claim more firmly than any one of them could alone.

Appendix K Sector-by-sector build-versus-buy guidance

This appendix applies the framework to specific sectors, naming for each the capabilities that are almost always context to buy and the narrow places where building the differentiator can be justified. The lists are illustrative rather than exhaustive, and the point in every case is the same: a large commodity core bought, a thin differentiating edge built.

Manufacturing and industrials

Buy the enterprise resource planning core, the general ledger, human resources, procurement, and standard quality and maintenance management, all of which are deep, mature, and undifferentiating. Build, if anything, the process capability that constitutes the plant's actual advantage: a proprietary production or scheduling method, a yield-optimization model, or the shop-floor integration that encodes a particular way of operating. The recurring failure is to customize the bought system into a de facto build; the recurring success is to adopt it near-standard and layer the one differentiating method on top.

Retail and consumer

Buy point-of-sale, merchandising, warehouse execution, and payments, the machinery of commerce that every retailer needs and none competes on. Build the customer-facing experience, the way shoppers discover, personalize, and complete a purchase, and, where it is genuinely proprietary, a distinctive merchandising or pricing method. The cautionary history of the sector, from Nike to Target, is a history of treating a commodity, demand planning, replenishment, inventory data, as if it were special, and paying for the mistake in lost sales.

Healthcare and life sciences

Buy the electronic health record, billing, scheduling, and the standard clinical and administrative systems, where the regulatory and interoperability burden is enormous and best shared across a vendor's base. Build only where a clinical algorithm, a device's embedded intelligence, or a research capability is itself the differentiator, and note that for a health-technology company such a build is frequently the high-risk AI system that makes its builder a regulated provider under Section 10. Here the build decision and the compliance decision are one.

Financial services

Buy the core banking or trading platform, the payments rails, and regulatory reporting, all prohibitively expensive to build and maintain to standard and best shared across an industry. Build the proprietary model, the pricing engine, the risk model, the fraud or credit algorithm, that constitutes the firm's edge. Institutions that tried to build their own core platforms mostly repeated the failures of Section 13; those that bought the platform and built the model tend to prosper.

Logistics and transportation

Buy warehouse and transportation management, which are context for most shippers and retailers. Build only where a superior routing, consolidation, or yield method is the entire value proposition, as it is for some third-party logistics providers and carriers, and even then build it thinly, on a bought operational core. This is the sector where the exception is most real, which is precisely why it demands the most disciplined classification.

Public sector

Buy the standard administrative systems, payroll, finance, and case management, and resist the twin temptations of bespoke building and heavy customization, both of which have produced some of the largest failures on record, from public payroll disasters to abandoned health exchanges. Public bodies face particular pressure from hard political deadlines and complex procurement, which makes the disciplines of phasing, testing, and scope control in Section 16 matter even more than in the private sector.

Technology and software

Build the differentiating product, which is the business, and buy the enormous commodity remainder, billing, customer-relationship management, analytics infrastructure, and human resources, precisely so that scarce engineering effort concentrates on the product. The discipline for a software company runs in the same direction as for everyone else, applied to a boundary that sits further toward building: even here, the most capable builders are disciplined buyers of everything that is not the product.

Across every sector the shape repeats: a large commodity core bought, a thin differentiating edge built, and a classification made honestly rather than by reflex. The sectors differ in where the line falls, never in whether the line exists.

Appendix L A build-proposal review checklist

When someone proposes to build rather than buy, this checklist helps a reviewer separate the proposals that deserve support from the ones that are the base rate wearing optimism. Treat any unchecked item as a question to be answered before the proposal proceeds, and treat a proposal that leaves several unchecked as the warning it is.

- **Differentiation is evidenced, not asserted.** The proposal shows that the capability is something customers choose you for, with evidence rather than conviction.
- **No product fits.** The proposal documents a real evaluation of the commercial and open-source alternatives and explains specifically why each falls short, rather than assuming none exists.
- **The scope is thin and bounded.** The build is a narrow layer on a bought core, with an explicit boundary, not an open-ended platform.
- **The lifetime cost is modeled.** The business case includes maintenance, integration, security, operations, and opportunity cost, not just the cost to build.
- **The owner is named and funded.** There is a person or team accountable for the system's whole life, funded permanently, not a sponsor who disbands at launch.
- **The knowledge will survive.** The plan captures design intent in documentation and tests, and does not depend on the tenure of one or two individuals.
- **The implementation is disciplined.** The plan phases its rollout, schedules the go-live away from peak periods, tests against real data, and resources the stabilization period.
- **The AI assumption is stripped out.** If the case depends on AI making the build cheap, it has been re-run without that assumption and still holds.
- **The exit is considered.** There is a plan for what happens if the build must be abandoned or replaced, and the data and interfaces are structured to make that possible.
- **The decision's reversibility is weighted.** An irreversible build, a core system, a deep customization, a data migration, has been scrutinized more heavily than a reversible one.

A proposal that passes every item is a candidate for one of the minority of builds that succeed. A proposal that fails several is not necessarily wrong, but it is asking the organization to accept the base rate of Section 4 without the traits that beat it, and it should not proceed until the gaps it reveals have been closed or consciously accepted.

Appendix M A one-page decision quick-reference

For the reader who wants the whole framework on a single page, here it is, compressed to the questions that decide, in the order they should be asked.

- **Classify.** Is this capability a genuine differentiator that customers choose you for, or context you merely need? If it is context, buy it, and stop here. Most capabilities stop here.
- **Check the fit.** For a differentiator, does any commercial or open-source product fit closely enough to adopt with modest configuration? If yes, buy or adopt it and build only the thin layer that differentiates.
- **Model the lifetime cost.** Add back maintenance, integration, security, operations, and opportunity cost. Compare the complete build cost, not the naive estimate, against the cost to buy.
- **Assess readiness.** Can you name and fund an owner, preserve the knowledge, and commit the standing capacity to maintain this forever? If not, buy.
- **Strip out the AI assumption.** If the build makes sense only because AI supposedly made it cheap, it does not make sense.
- **Run a pre-mortem.** Imagine the build failed expensively; if the reasons are many and unowned, heed them.
- **Decide, then build the edge, not the core.** If you build, keep it thin, decoupled, and disciplined in implementation, and point AI at the edge.
- **Revisit.** Re-examine the decision as the facts change, and be willing to stop.

Everything else in this guide is elaboration. If a decision passes these questions honestly, it is as sound as a build-versus-buy decision can be; if it cannot, the questions have shown you where it is weak.

Appendix N Questions to ask a vendor's references

A vendor's own references are chosen to impress, but a disciplined reference call can still yield the truth if the questions are pointed enough. These are designed to surface what the sales process conceals.

- **What went wrong, and how did the vendor respond?** The most useful question of all; a reference who cannot name a single problem is either lucky, guarded, or not comparable to you.
- **How long did implementation actually take, versus the plan?** The gap between the two is the vendor's real track record on schedule.
- **How much did you customize, and do you regret any of it?** A reference deep in customization is a warning, per the failures of Section 13.
- **What did it cost in total, including the things not in the original quote?** The reference's surprises will be your surprises.
- **Who owns and maintains it now, and how large is that team?** The answer reveals the ownership burden the sales process omitted.
- **How did the data migration go?** The most common point of failure, and the one references remember most vividly.
- **If you were deciding again, would you buy the same product?** The single most revealing question, and the one to leave time for.

Ask these of a reference the vendor did not hand-pick if you can find one, and weigh a reluctant, specific answer more heavily than an enthusiastic, general one. The reference call is the cheapest due diligence available, and these questions are what turn it from a formality into evidence.

Appendix O A first-ninety-days ownership checklist

Whether you built or bought, the first ninety days after go-live determine whether the system becomes an asset or a liability. This checklist covers the ownership work that the launch adrenaline tends to crowd out.

- **Confirm the owner is in place and funded.** The named owner from the business case should be actually accountable now, not still being recruited.
- **Staff and run the stabilization period.** Expect defects, and resource their resolution rather than reassigning the team the week after launch.
- **Measure adoption, not just uptime.** Track whether people are actually using the system as intended; a system that is up but unused has not succeeded.
- **Capture the knowledge while it is fresh.** Document the configuration, the decisions, and the integrations now, before the implementation team disperses.
- **Establish the maintenance rhythm.** Put the recurring work, dependency updates, security patching, and monitoring, on a schedule, so it happens by routine rather than by crisis.
- **Verify the exit remains open.** Confirm that data still exports cleanly and that the portability you negotiated actually works, before you depend on the system completely.
- **Review the decision against reality.** Compare the outcome to the business case, honestly, and record what you learned for the next decision.

None of this is glamorous, and all of it is the difference between a system that is owned and a system that merely exists. The organizations that get the first ninety days right are the ones for whom the build-versus-buy decision, whichever way it went, actually pays off.

Appendix P A build-versus-buy decision record

Because a build-versus-buy decision should be revisited as the facts change, it helps to record it in a consistent form when it is made, so that a future review has something concrete to revisit rather than a vague memory. This template captures the minimum that makes a decision auditable.

- **The capability and its classification.** What was decided about, and whether it was judged core or context, with the evidence for that judgment.
- **The options considered.** The commercial, open-source, and build options evaluated, and why each was or was not chosen.
- **The lifetime cost estimate.** The complete cost of the chosen path, including the layers of Section 5, not the naive figure.
- **The readiness assessment.** The result of the Appendix H assessment, and any gaps accepted knowingly.
- **The decision and its owner.** What was chosen, who is accountable, and how the ownership is funded.
- **The key assumptions.** The facts the decision depends on, especially any that could change, so a future review knows what to check.
- **The revisit trigger.** The conditions, a date, a cost threshold, a change in the market or the strategy, that should prompt the decision to be reconsidered.

A decision recorded this way costs an hour and saves far more, because it converts an implicit choice into an explicit one that can be reviewed, defended, and, when the facts change, reversed without relitigating from memory. The absence of such a record is how organizations end up trapped in decisions no one remembers making.

Appendix Q A field guide to vendor claims

Vendor marketing has a vocabulary of its own, and learning to translate it is part of buying well. None of these phrases is dishonest exactly, but each conceals a question the buyer should ask. This field guide pairs the common claim with the question it should prompt.

- **"AI-powered."** Ask which specific task the AI performs, how it was evaluated, and what happens when it is wrong. The phrase describes a feature, not a benefit, and the benefit is what you are buying.
- **"Seamless integration."** Ask for the integration to be demonstrated with your systems and your data, because seamless usually means possible with effort, and the effort is yours.
- **"Turnkey."** Ask whether it is turnkey for the demonstration or for your data, your volumes, and your edge cases; the gap between the two is the implementation, per Section 16.
- **"Fully customizable."** Treat this as a warning rather than a feature. The capacity to customize deeply is the capacity to build a de facto build, with all the risk of Section 13; ask instead how well it works unmodified.
- **"Trusted by industry leaders."** Ask to speak to a customer at your scale whose rollout went badly, using the questions in Appendix N; a logo wall is not a reference.
- **"Compliant" or "enterprise-grade security."** Ask what, specifically, the vendor certifies, and remember that a compliant vendor does not make you compliant, per Section 10; the vendor supplies evidence, and you still own your controls.
- **"Rapid time to value."** Ask over what horizon and for whom; a fast start followed by a slow, costly integration is a common shape, and the total time to value is the number that matters.

The point is not cynicism. Good vendors make all of these claims and back them up when asked; the questions simply separate the vendors who can from the vendors who cannot. A buyer fluent in this translation runs a more honest evaluation, and an honest evaluation is the whole of buying well.

Appendix R A summary of the evidence

For quick reference, the table below consolidates the principal findings the guide relies on, with their source and the section in which they appear. The figures come from studies using different models, tasks, and populations and are not directly comparable; they are best read as directional, and the fuller discussion in each section carries the necessary caveats.

Finding	Source	§
Large IT projects run about 45% over budget and deliver roughly 56% less value than promised; about one in six is a catastrophic "black swan."	McKinsey & University of Oxford (2012)	4
Only about a third of software projects fully succeed; roughly two-thirds are challenged or fail.	Standish Group, CHAOS	4
A controlled trial found a meaningful task speed-up with an AI coding assistant.	GitHub Copilot RCT (2023)	7, 9
Experienced open-source developers were about 19% slower with AI on real tasks.	METR (2025)	9
For every 25% increase in AI adoption, delivery throughput fell about 1.5% and stability about 7.2%.	Google DORA (2024)	9
About 39% of developers reported little or no trust in AI-generated code.	Google DORA (2024)	9
Developers using AI assistants tended to write less secure code while believing the opposite.	Stanford (2023)	9
A large share of AI-generated code samples contained security vulnerabilities.	Veracode (2025)	9
Hallucinated software dependencies ("slopsquatting") form a new supply-chain attack surface.	Spracklen et al. (2025)	9
AI can produce roughly the first 70% of a system quickly; the final 30% stays slow and human.	Osmani (2024)	9
Spending on supply-chain software with agentic AI is forecast to grow sharply through 2030.	Gartner (2026)	11

A directional summary of the evidence. Figures are drawn from studies with differing methods and populations; see the cited sections for the full context and caveats.

18 Methodology, caveats, and sources

Methodology

- This article synthesizes peer-reviewed and industry research, controlled studies, independent software analytics, vendor disclosures, and contemporaneous reporting, current to early 2026. Supply Chain Research is independent and accepts no payment from the vendors discussed.
- Where studies are cited, we have preferred primary sources and controlled trials, and we have tried to represent both the gains and the limitations of AI-assisted development rather than either the optimistic or the pessimistic case alone.

Caveats

- The AI-productivity evidence is early, mixed, and moving quickly. The controlled trial showing a 56 percent speed-up and the study showing experienced developers 19 percent slower both used specific models, tasks, and populations, and are not directly comparable. Both can be true: AI helps on bounded tasks and can hurt on complex work in mature codebases.
- Several figures are vendor-sourced or come from firms with a commercial interest, on both sides of the argument. Security-testing, code-analytics, and software-development firms each publish figures that can serve their positioning. The most robust anchors are the McKinsey and Oxford overrun study, long-running maintenance-cost research, and the independent academic security studies.
- The Standish CHAOS data is a proprietary, unaudited survey whose definitions have been criticized by academics; it is used here as directional. Software maintenance-cost shares are rules of thumb that vary by context.
- The failure case studies draw on contemporaneous reporting and published post-mortems. Their financial figures mix lost sales, write-downs, remediation, and abandoned-project costs, and vary by source; some, such as the Lidl figure, are reported rather than officially published. They are directionally consistent and precisely uncertain.
- This piece anchors to business-to-business enterprise and supply chain software. Its conclusions may differ for consumer software, true greenfield ventures, or capabilities with no viable vendor market.
- The worked example and scoring case in this guide are illustrative rather than empirical. They are constructed to show how the framework behaves, not to report a specific company's results, and the figures should be replaced with your own before any decision.
- The pace of change on the AI side is rapid enough that specific model behaviors, vendor capabilities, and study results cited here may shift within months. The structural argument, that AI lowers the cost of writing code but not the cost of owning software, is expected to outlast any individual figure, and readers should weight the argument more heavily than any single number.

A note for different readers

This guide serves several audiences, and each will use it differently.

For the executive who must approve or reject a build, the essential material is the base rate of Section 4, the iceberg of Section 5, and the decision framework of Section 14, together with the one-page quick-

reference in Appendix M. The single most valuable habit is to ask, of any build proposal, whether the capability is a genuine differentiator and whether the lifetime cost, not the build cost, has been modeled. The objections in Appendix G and the field guide in Appendix Q are there for the conversations that follow.

For the architect or engineering leader, the heart of the guide is the third path of Section 12 and the readiness questions of Section 15: buy the commodity core, build the thin differentiating edge, keep it decoupled, and point AI at the edge rather than the core. The evidence in Sections 7 through 9 is there to calibrate expectations of what AI on the build side actually delivers, and the guardrail and implementation material in Appendices B and I to execute well once a decision is made.

For those running the purchase, the operative sections are 16 and its companion appendices, D, E, N, and Q: run an evaluation that predicts reality, negotiate the terms that protect you before you sign, ask a vendor's references the questions that surface the truth, and translate the marketing into the questions it conceals. The regulatory material in Section 10 matters here too, because the provider-and-deployer distinction increasingly shapes both the contract and the compliance posture.

No reader needs the whole guide at once. The argument is a single thesis elaborated across many angles, and any one angle, read in the moment a decision demands it, carries most of the point on its own.

A living document

This guide is a snapshot of a fast-moving subject, and it is written in the knowledge that parts of it will date. The evidence on AI-assisted development is accumulating month by month; the regulatory framework is being implemented and amended on a published but shifting schedule; and the vendor landscape reshapes itself continuously as capabilities that were differentiators become commodities. The thesis is stable, but the facts that illustrate it are not, and a reader consulting this guide well after its writing should treat the specific figures, dates, and product claims as starting points to be re-checked rather than settled conclusions.

That impermanence is not a weakness of the argument but a feature of the domain, and it is, in a sense, the argument in miniature. The reason to buy the commodity is precisely that the commodity keeps improving under someone else's ownership, so that what you would have built and frozen at the moment of building instead advances without your effort. The reason to build only the differentiator is that the differentiator is the one thing whose change you want to own. A guide about the wisdom of letting others carry what you need not build yourself can hardly complain that the world keeps moving; it can only recommend, one more time, the discipline of deciding well and revisiting often.

On the limits of this guide

A guide that argues for intellectual honesty owes some candor about itself. Several limits are worth stating plainly, so that the reader can weigh the argument accordingly.

- **The evidence is directional, not decisive.** The productivity and quality studies use different models, tasks, and populations, and the tools are improving quickly; a figure true in 2025 may not hold in

2027. The pattern the studies form is more durable than any single number, but the numbers should be read as indicative.

- **The case studies are selected.** The failures in Section 13 are famous because they were large and public; the many quiet successes of both building and buying are underrepresented, as they always are, because success rarely makes the news. The base-rate research in Section 4 is the corrective, and it should be weighted more heavily than any individual story.
- **The framework is a default, not a rule.** Buy the commodity, build the differentiator is a strong prior, not a law, and this guide has been at pains to name the cases where building is correct. A framework applied without judgment becomes the very reflex it was meant to replace.
- **The regulatory picture is moving.** The obligations described in Section 10 are being implemented on a schedule that is itself being amended, and specific classifications are fact-dependent and should be confirmed with counsel rather than inferred from this guide.
- **Vendor claims and capabilities change fast.** The specific products and capabilities named in this guide will evolve, and any concrete claim about a vendor should be verified directly before it is relied upon.

None of these limits undoes the argument; each simply marks where the reader's own judgment must supply what a general guide cannot. The thesis, buy the commodity and build only what truly sets you apart, is as old as the discipline and as current as the newest tool, and it survives every one of these caveats. The discipline of restraint is not a formula to be applied but a habit to be practiced, and the purpose of this guide is met if it makes that habit a little easier to keep.

Sources

[1] Bloch, Blumberg, and Laartz, McKinsey with University of Oxford (2012). Delivering large-scale IT projects on time, on budget, and on value.

[2] Standish Group, CHAOS research. Summary of software project outcomes (success, challenged, failed).

[3] Geoffrey Moore. Core versus context (from Dealing with Darwin), overview.

[4] ScienceSoft. Software maintenance costs: how to estimate and optimize.

[5] Peng, Kalliamvakou, Cihon, and Demirer (2023). The impact of AI on developer productivity: evidence from GitHub Copilot (controlled trial).

[6] McKinsey (2023). Unleashing developer productivity with generative AI.

[7] METR (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity.

[8] GitClear (2025). AI Copilot code quality: 2025 data on code clones and churn.

[9] Google DORA (2024). Accelerate State of DevOps Report: effects of AI on delivery.

[10] Perry, Srivastava, Kumar, and Boneh, Stanford (2023). Do users write more insecure code with AI assistants?

[11] Veracode (2025). GenAI Code Security Report.

[12] Spracklen et al. (2025). We have a package for you: hallucinated package dependencies and slopsquatting.

[13] FOSSA (2025). Slopsquatting: AI hallucinations and a new software supply-chain risk.

[14] Addy Osmani (2024). The 70% problem: hard truths about AI-assisted coding.

[15] MACH Alliance. Composable and MACH architecture principles and adoption.

[16] Virto Commerce. Composable commerce versus MACH architecture.

[17] LoadSys. Why AI-generated code is only 70% done, and what that means for a rebuild.

[18] Datasoft Technologies (2026). TMS versus WMS for logistics: a build-first decision guide.

[19] XDA Developers (2025). Testing every AI vibe-coding tool on a real project: they choke at the same point.

[20] Particula Tech (2025). AI code churn, cloning, and technical-debt data.

[21] Stack Overflow (2025). Annual Developer Survey 2025: AI tool adoption, usage, and sentiment.

[22] Gartner (2026). Supply chain management software with agentic AI: spend and adoption forecast, 2025–2030.

[23] Mordor Intelligence (2026). Supply chain management software market: size, share, and industry report.

[24] Technavio (2026). Supply chain management (SCM) software market analysis and forecast, 2026–2030.

[25] Panorama Consulting Group. Post-mortem case studies of enterprise software failures: Nike and i2; Target Canada; and Lidl's SAP ERP project.

[26] Henrico Dolfing (2019–2020). Project failure case studies: Nike's $100 million supply-chain “speed bump”; Target's cross-border expansion; and Lidl's €500 million SAP debacle.

[27] CIO Magazine and CFO Magazine (1999–2001). Contemporaneous reporting on the Hershey SAP R/3, Manugistics, and Siebel implementation and on the Nike i2 project.

[28] Consultancy.uk (2018). Lidl cancels SAP introduction having sunk €500 million into it.

[29] Canadian Business and Maclean's (2016). The last days of Target Canada.

[30] European Commission (2024–2026). The EU Artificial Intelligence Act, Regulation (EU) 2024/1689: risk tiers, provider and deployer obligations, and implementation timeline.

[31] Latham & Watkins (2024). EU AI Act: obligations for deployers of high-risk AI systems, Articles 25 to 27.

[32] CIO (2025). “18 famous ERP disasters, dustups, and disappointments”: reporting on the National Grid, Revlon, Waste Management, and MillerCoors implementations.

[33] The Register and UpperEdge (2018). Coverage of the National Grid–Wipro SAP dispute and the seventy-five-million-dollar settlement.

[34] Henrico Dolfing. Additional project-failure case studies: Queensland Health payroll; Hertz and Accenture; Cover Oregon; LeasePlan; and Sacramento City Unified School District and Workday.

Additional context drawn from vendor disclosures (Blue Yonder, Kinaxis, o9 Solutions, SAP, Oracle, Manhattan Associates) and from contemporaneous reporting and published post-mortems of the enterprise software implementations referenced in Section 13. AI-productivity figures are drawn from studies using different models, tasks, and populations and are not directly comparable; security and code-quality figures include vendor-sourced research and should be read as directional. Vendor AI capabilities and claims should be verified directly, as the pace of change is rapid.

Supply Chain Research is an independent, vendor-neutral research platform for supply chain and technology leaders. We accept no payment from the vendors discussed. This article is analysis, not procurement or legal advice, and its conclusions should be validated against your own requirements before any decision.