AI & Software

The economics of software when code becomes abundant

When producing code stops being the scarce part of the system, where does value accumulate? A first attempt at mapping the new economics.

Abstract. This essay examines what happens to the economics of software once producing code stops being the scarce resource in the system. Drawing on controlled studies of AI-assisted programming, enterprise security telemetry, and three decades spent building software companies, it argues that AI's real effect is not a productivity multiplier but a relocation of value — away from writing code and toward the judgment, architecture, and accountability that determine what should be built and whether it can be trusted. Software businesses still pricing by the hour are, in effect, financing their own replacement.

For most of the history of our industry, writing software was the bottleneck. Teams were sized around it, budgets were built around it, and competitive advantage often came from being able to produce more of it, faster. Every methodology we invented — waterfall, agile, scrum, DevOps — sought, among other things, to increase how much useful software the same number of hands could produce and run.

Artificial intelligence is loosening that constraint. And when a constraint relaxes, the whole system reorganizes around whatever becomes scarce next.

I spent thirty-five years building technology companies across Ibero-America, most of them as founder and CEO of a software company that grew, over three decades, by selling exactly what is now becoming cheaper: the capacity to produce code that works, reliably and at scale. So this question is not academic for me. It is the question my own industry has to answer in the next few years, and the honest answer is more unsettling than the marketing around AI coding tools would suggest.

What actually changed

The productivity gains from AI-assisted development are real. They are also smaller, and stranger, than the popular narrative admits. A synthetic benchmark task, done by a developer with no prior context, can be solved in half the time with a good coding assistant. That result gets repeated in keynote after keynote. What gets repeated less often is what happens when researchers measure experienced developers working in their own mature repositories: in a controlled study published in 2025, developers using AI were measurably slower on real tasks — nineteen percent slower [1] — despite believing, both before and after, that the tools had made them faster. The gap between what a coding assistant demonstrably does and what it feels like it does is not a rounding error. It is a central fact that anyone deciding on AI adoption has to confront.

One of the broadest syntheses of the evidence available to date — a meta-analysis pooling more than twenty independent studies [2] — places the real productivity effect in a moderate range: useful, uneven, highly dependent on the type of task and the maturity of the codebase, and far from the tenfold multiplier promised in sales presentations. Greenfield work that is well-scoped and low-context benefits the most. Large, brownfield, high-risk systems benefit the least, and sometimes do not benefit at all.

None of this means AI does not matter. It means it does not matter where most people are looking.

Watch where the bottleneck actually moves and the pattern becomes clear. Code generation accelerates. Code review does not necessarily accelerate, and can slow down, because there is more to review, it arrives in larger batches, and a human now has to evaluate work they did not produce and cannot fully trust. Testing, integration, and deep understanding of a system also see far smaller gains

than code generation does. The bottleneck did not disappear. It moved a few steps further down the process, from the person typing to the person deciding what the typing was worth.

One structural — not merely operational — consequence follows from this dynamic: the internal shape of a software company is changing along with it. When generating is cheap, a team does not need to be large to produce a large amount of code; it needs to be small and have an unusually high density of judgment, because judgment is now the scarce input being multiplied, not labor. I have started to see teams of four or five people, with very little ceremony among them, produce what used to require much larger teams — not only because each of them produces faster, but because there are fewer places for a bad decision to hide before someone with real judgment catches it. That is a different kind of organization than the one most software companies, including my own, were built to run.

Where the risk actually moved

There is a deeper impact than the productivity debate: when producing a plausible piece of code gets cheaper, errors are not eliminated, their shape changes. Trivial syntactic errors drop sharply, because that is exactly the kind of error a language model knows how to avoid. But errors that require holding an entire system in your head — an authorization boundary, a trust assumption, a design decision made three modules away — those errors increase, sometimes dramatically. A large-scale enterprise telemetry report found triple-digit increases in certain risk categories, including architectural flaws and privilege-escalation paths, even as more superficial errors declined. [3] There is no contradiction in this. It is what you would expect once you understand that generating code and understanding a system are two different skills, and only one of them got radically cheaper.

The same logic applies to talent. A junior engineer used to learn the shape of a system by doing its tedious parts: reading someone else's code, tracing a bug through five files, writing the repetitive code that teaches you a codebase's idioms before you are trusted with its architecture. If an agent now does that work, the learning that work used to provide disappears along with it. A controlled experiment on skill formation found exactly this: developers who delegated a learning task to an AI assistant retained less conceptual understanding and were worse at debugging afterward, with no measurable efficiency gain in exchange. [4] Multiply that across an entire generation of engineers who never did the tedious part, and you are not looking at a productivity story. You are looking at the risk of a slow-motion talent shortage that could start becoming visible toward the end of the decade, at which point it is already a structural problem, not a hiring one.

> *The pattern, in both cases, is the same: cheap generation does not eliminate difficulty. It shifts it toward what cannot be generated — judgment, context, and the years it takes to build both.*

The part that should worry a software company more than any of this

I built and ran a company for three decades whose commercial logic, like that of almost every software company of its generation, rested on an implicit unit of value: the hour of skilled work. Time and materials, staff augmentation, license plus implementation hours, maintenance contracts billed by the ticket — all of it assumes that producing software is the scarce, valuable thing a client is paying for.

That assumption is expiring, and it is expiring faster for some business models than for others. If your client's real problem was ever “I need someone to write this,” AI has already made that problem smaller than it was two years ago, and it will keep shrinking. That does not mean software companies stop having value to sell. It means the value has to migrate toward what remains scarce: domain knowledge nobody has encoded into a model yet, the judgment to decide what deserves to be built, the architecture that keeps a system coherent after the tenth agent has touched it, the willingness to put your name on the result and be accountable for what it does in production. None of these capabilities is a commodity. They never were. They were simply cheaper to overlook when the hour of code was what everyone was actually buying.

This was never a hypothetical question for me. My own company spent decades on one side of a debate the industry never fully settled: build a product of your own and sell services around it, or sell services as the product itself — skilled hours wrapped in a service contract, with no asset left behind

once payment was made. We chose the first path, and stayed on it through long stretches when the second path was easier to sell and faster to scale. An engineer improving a product the company owns is accumulating an explicit asset. An engineer executing someone else's specification can also accumulate knowledge, reputation, and capability, but the principal value produced belongs to the client and is monetized once, on the invoice.

I think that choice reads better today than it did during most of the years we held to it. When code gets cheaper, the company that only ever sold hours discovers it was selling the one thing that stopped being scarce. The company that built a product still has something a language model does not hand over just because it can write code fast: the data it has accumulated, the workflow it is embedded in, the thousand edge cases it has already quietly solved.

But I want to resist the comfortable version of that conclusion, because this essay has tried not to reach for comfortable conclusions. Betting on product gave us a better starting point. It did not buy us immunity. The same abundance that erodes the value of an hour also erodes the value of a moat built on what it would take a competitor years to rebuild, given that those years compress into months once building is cheap, and a client's internal team can now do in-house what used to require buying your product. Standing on the product side is a better place to be when this wave arrives. It is not a place where you get to stop paying attention.

Companies that keep pricing by the hour in a market where the hour is worth less every quarter are not protecting their margin. They are financing their own replacement. Those that hope to still be relevant in five years should already be renegotiating what they sell — toward outcomes, toward platforms, toward the kind of accountability that only a human with something real at stake can offer.

I watched something like this happen once before, on a smaller scale, when the cloud made infrastructure cheap and companies that sold servers had to decide, quickly, whether they were in the business of selling boxes or in the business of solving problems that happened to require boxes. Many of the ones that survived that transition were not the ones with the best hardware. They were the ones that had already figured out, before the transition forced the question, what they actually sold once hardware stopped being the point. I think software is having its own version of that moment now, compressed into a much shorter window, and with much less patience from the market for companies that take too long to answer.

That does not mean every contract needs to become outcome-based overnight, and I would distrust anyone who tells a room full of CEOs that it does. Time and materials still has its place, particularly in early-stage discovery, when nobody — not even the client — yet knows what the right scope is. But a company whose entire portfolio is still priced by the hour in 2026 is a company betting, whether it says so or not, that this shift is temporary. I do not think it is.

What replaces the hour, in practice, tends to take one of three shapes for a mid-sized software company. The first is value-based pricing, tied to a metric the client already cares about — a percentage of cost saved, revenue enabled, risk reduced — which requires the harder, earlier work of agreeing on that metric before the engagement starts, not after. The second is the platform model: instead of billing for the labor behind a solution, you sell recurring access to the solution itself, and labor becomes your cost of goods rather than your invoice. The third is the performance agreement — narrower in scope than the other two, but useful as a bridge — where a fixed fee covers a defined outcome, with an upside or a penalty attached to hitting it or missing it. None of the three is trivial to set up, and all three ask a mid-sized software company to know its own economics better than billing by the hour ever required. That is precisely why they are defensible: they are hard to copy quickly, one of the few kinds of advantage that still matter once code itself stops being scarce.

What I am actually asking

I did not write this to argue that AI is overhyped, or that it is underhyped — both framings assume the interesting question is how much code a machine can produce, and I no longer think that is the interesting question. The interesting question is the one in this essay's subtitle: when producing code stops being the scarce part of the system, where does the value go?

My working answer, after three and a half decades of watching this industry reinvent itself every few years, is that it goes to the same place it usually shifts to when a technology matures: toward whoever

can say, faster and more reliably than anyone else, what is actually worth building — and is willing to be responsible for that answer.

Taking on that responsibility is harder than writing fast code, but it is also more valuable. It rewards judgment over volume, architecture over quantity, and thirty years of having been wrong in expensive, memorable ways over any amount of fluency with a new tool.

In the economics of software where code becomes abundant, we are not facing a paralyzing crossroads for the industry: we are facing a mirror. Abundance finally frees us from the goldsmith's work of syntax only to raise the bar, marking the exact moment the game becomes truly interesting. Value will be measured less and less by how many lines we know how to generate, and more and more by a question whose answer we will not be able to delegate: whether what we are building truly deserves to exist, and whether we are willing to stand behind that decision with our own signature.

Notes

1. *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. METR, July 2025. Randomized trial, 16 experienced developers, 246 real tasks in their own repositories.*
2. *A meta-analysis of the effect of generative AI on productivity and learning in programming. Maier et al., LMU Munich, 2026. Pools 23 studies and 27 effect sizes.*
3. Enterprise codebase telemetry on AI-generated code, Fortune 50 companies, Dec. 2024–Jun. 2025. Apiiro, 2025.
4. *How AI Impacts Skill Formation.* Shen & Tamkin, 2026. Randomized controlled trial on learning outcomes with AI assistance.

ANIBAL CARMONA
anibalcarmona.com
Barcelona · August 2026
Founder KAIROSIA · LinkedIn

Aníbal Carmona is a technology entrepreneur and systems engineer with more than 35 years of experience in the software industry.
In 1991 he founded Unitech, a company he led for more than three decades, spanning several technology generations, economic cycles, and international expansion, eventually operating in Argentina, Chile, Peru, Colombia, and Spain. Along the way, he specialized in technology applied to government, justice, and security.
His career extended beyond the business world. He served as vice president and later president of the Argentine Software Industry Chamber (CESSI), representing the Argentine technology sector before companies, universities, and public institutions. He took an active part in debates on digital transformation, talent development, intellectual-property exports, and the Knowledge Economy, as well as in public-private dialogue on the future of the technology industry.
In 2021 he founded Unitech Iberoamericana in Barcelona and led the company's market development in Spain and its European expansion. During this period he became part of the Spanish business-technology ecosystem, taking part in AMETIC, and deepened his work on digital transformation, artificial intelligence, and the application of new technologies to the justice system.
His interest in the impact of technology on organizations, employment, and society predates the current artificial-intelligence revolution. In recent years he has focused his work and research especially on AI, the transformation of the software industry, and the changes these technologies are producing in business models and in the way organizations are built.
After more than three decades building Unitech and five years developing its European chapter from Barcelona, he began a new professional chapter through KAIROSIA, where he focuses on strategic advisory, research, investment, and support for entrepreneurs and organizations navigating the transformations brought about by artificial intelligence.
His essays combine the perspective of someone who has lived through several technology generations building companies with the outlook of someone now trying to understand a transformation that is challenging many of the premises on which the software industry and organizations were built over the past decades.