ANALYSIS · ARTIFICIAL INTELLIGENCE · SOFTWARE ENGINEERING

The Subprime Risk *of Artificial Intelligence*

Are We Securitising Technical Debt?

Cristian Costantini
Software Architect

"Well written and well justified concerns."

IMAN SALEH, PHD · AI ETHICS

INTRODUCTION

The market is always right. *Until the day it presents the bill.*

In 2008 the world discovered that an apparently safe product could conceal enormous risks. Today, in the software industry, we may be constructing a dangerously similar mechanism.

For years, mortgages had been granted to borrowers whose capacity to repay was often insufficient. Those mortgages were bundled together, transformed into complex financial instruments, and sold on international markets. Some tranches of those products even received the highest reliability rating: **AAA**. On paper they were safe investments. In reality, part of their safety depended on fragile assumptions, inadequate risk models, and deeply distorted economic incentives.

The system worked. Banks earned, intermediaries collected fees, investors purchased apparently reliable products. Until it no longer worked.

> *The crisis did not arise simply from the existence of risky mortgages. It was fuelled by a system that allowed risk to be transformed, distributed, and underestimated — separating the immediate economic benefit from the future consequences.*

2008 came back to me whilst reading some commentary on artificial intelligence applied to software development. The reasoning I encounter ever more frequently runs essentially like this: *AI writes code better and faster than human beings. Traditional architecture is a romantic exercise. The market wants working applications, not elegant software. Those who continue to invest time in design risk simply being overtaken.*

Sometimes the argument concludes with an observation that is hard to contest on an individual level: *clients are satisfied and profits are growing.*

I do not doubt that this is true. The problem is that it could also be true when the real quality of the product has not been adequately verified. And this distinction deserves far more attention than we are currently giving it.

CHAPTER 01

An Extraordinary Transformation

Let us begin with a fundamental premise. Artificial intelligence is one of the most important tools ever introduced into software development. It is not necessary to deny this in order to discuss it critically.

Today an AI agent can analyse requirements, propose domain models, generate application components, write tests, identify vulnerabilities, suggest refactoring, and assist with deployment operations. It can even coordinate other specialised agents. Activities that once took days can be completed in a few hours.

+26%

tasks completed with AI
Microsoft / Accenture study, 2025

+19%

more time needed — participants were **slower**, not faster
METR, 2025 — senior developers

≠

results not directly comparable: different tasks, tools, and experimental conditions

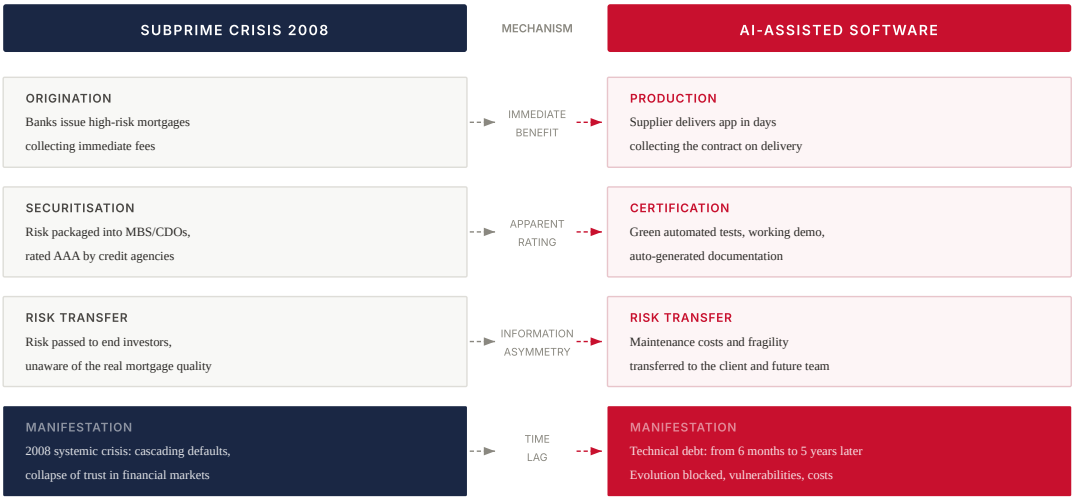
Empirical results on productivity are not uniform. In three controlled field experiments, a 2025 study found that developers with access to AI tools completed on average around **26% more tasks**[2] than comparison groups — a gain in the number of tasks completed per unit time. In a different context, a 2025 METR study of experienced developers working on familiar open-source projects found that, with the AI tools examined, participants took on average **19% longer**[3] to complete tasks: they were therefore slower, not faster. The two results should not be treated as equivalent measures — they concern different work, tools, and experimental conditions. They show, rather, how much the impact of AI depends on context and on how it is used.

Even assuming that in the coming years agents manage to drastically reduce development times across the board, an essential question would remain:

> *Are we accelerating only the production of software, or also our capacity to understand, verify, and maintain it? These are different activities. And conflating them could become extremely costly.*

Structural Parallel — 2008 Subprime Crisis vs AI-Assisted Software Development

Structural mechanisms compared: distorted incentives, information asymmetry, and risk transfer



Common mechanism: the benefit is immediate and visible; the risk is deferred and invisible at the point of transaction.
The parallel is structural, not dimensional: the risks of AI-assisted software do not imply a systemic shock on the scale of 2008.

Fig. 1 — Structural Parallel. The comparison concerns incentives and the transfer of risk: immediate benefit for the producer, deferred risk for the user. It does not demonstrate macroeconomic consequences analogous to those of the subprime crisis.

CHAPTER 02

Software Is More Than the Code That Comprises It

Imagine a company that commissions a platform to manage orders, payments, and shipments. The supplier uses AI agents. Within a few days it delivers a complete application. The interface is modern. Payments work. Orders are recorded. Shipments are trackable. There are even hundreds of automated tests.

Six months later, however, the system must be modified. A new business requirement demands the ability to handle partial payments, refunds, and orders split across multiple shipments. Another developer takes on the project. They discover that the payment logic is spread across several components, that some rules are duplicated, and that the boundaries between order management and accounting are not clearly defined.

THE CENTRAL PROBLEM

The documentation describes the classes, methods, and how the APIs work. It does not explain, however, *why* that structure was chosen. It does not indicate which alternatives were discarded, which constraints guided the project, or which trade-offs were accepted. The code exists. The knowledge of the decisions, perhaps, does not.

The Delivery Price Is Not the Cost of the Software

Illustrative example — not based on observed data

ITEM	SOFTWARE A	SOFTWARE B
Initial development	€5,000	€12,000
Estimated annual maintenance	€6,000	€2,000
Total cost of ownership after 3 years	**€23,000**	**€18,000**

Software A costs €7,000 less at delivery, but €5,000 more after three years. The difference does not depend on who — person or AI agent — wrote the code. It depends on the characteristics of the product, the technical choices made, and the capacity to evolve it without multiplying costs. The economic point is straightforward: **the initial price is visible; the maintenance risk often is not.**

Maintenance estimates vary considerably by project, and in many cases the less expensive system at the outset remains the better value over time. The table is intended to illustrate the mechanism, not to predict the cost of any real software.

A software architecture is not simply a collection of diagrams, patterns, or conventions. It is the result of decisions made to govern complexity and to make the consequences of change predictable. A good architecture does not guarantee the absence of errors. But it can make the system more comprehensible, verifiable, modifiable, and recoverable.

> *The problem is not that an AI agent cannot design a valid architecture. The problem arises when nobody takes on the task of genuinely evaluating that design, challenging its assumptions, and preserving its rationale.*

CHAPTER 03

Invisible Risk Accumulates Over Time

One of the most insidious aspects of AI-assisted technical debt is that the risk does not manifest at the moment of delivery. The curve of perceived value and that of real risk travel on separate tracks — at least in the initial phases.

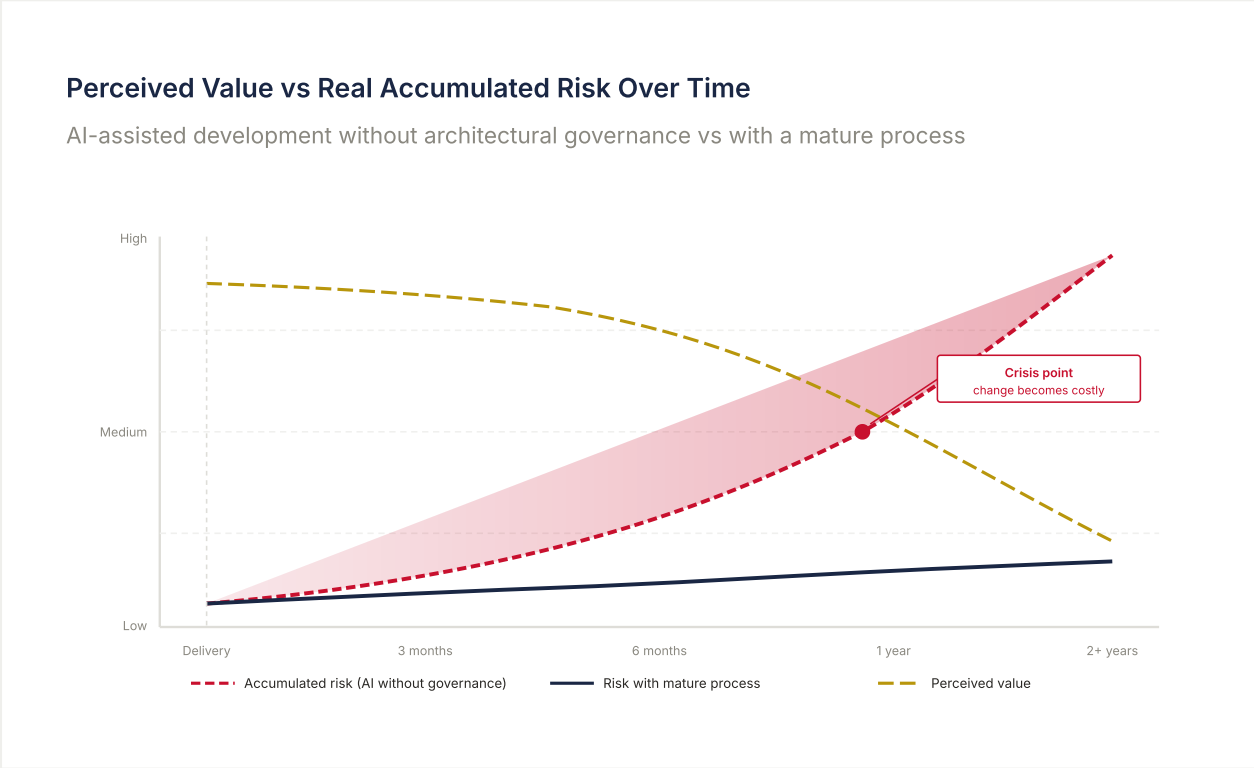


Fig. 2 — Risk accumulation over time. The author's conceptual schema: an illustrative trend, not an empirical estimate. The divergence between real risk and perceived value is a qualitative representation of the mechanism described in the text. The crisis point and time horizons are indicative and depend heavily on the project.

The 2025 DORA research[5] highlighted a significant finding: greater AI adoption was associated both with an increase in software delivery throughput and with greater delivery instability. The association does not demonstrate that AI directly causes stability problems. But it indicates that productivity gains do not automatically translate into better outcomes across the full lifecycle.

DORA 2025 — AI AS AN AMPLIFIER

If an organisation has robust processes, effective tests, reliable platforms, and mature engineering practices, AI can amplify its strengths. If instead the organisation works with confused requirements, fragile infrastructure, and insufficient verification, AI can also contribute to accumulating technical debt more rapidly.

CHAPTER 04

Information Asymmetry — The Market for Lemons

Suppose two suppliers submit bids to build the same application. The first promises delivery in five days. The second proposes a process of analysis, design, AI-assisted implementation, risk verification, and documentation. Their proposal costs more. Both use AI agents. Both can show a convincing demonstration. Which will the client choose?

If they lack the means to evaluate hidden quality, it is entirely reasonable for them to attach great importance to price and delivery times. And this does not mean they are a poor client. It means they lack information.



Fig. 3 — Information asymmetry. The more interesting case is not a supplier who knowingly conceals: often the internal quality has not been verified even by those who produced the code. The dimensions in grey are not simply invisible to the client — they are unverified by anyone at the moment of delivery. When the supplier knows aspects of the quality that the client cannot evaluate, the information asymmetry described by Akerlof in 1970 [7] emerges. When nobody has verified them, a problem of shared uncertainty is added.

> *The buyer may not have the tools to distinguish a system that is genuinely robust from one that merely appears robust under initial conditions.*

If the costs of design, verification, and maintainability remain invisible, those who bear them may simply appear less competitive. Those who do not bear them may seem more efficient. At least initially. The more responsible engineering comes to be perceived as the least productive, because it is the only one that makes the cost of risk explicit.

CHAPTER 05

What to Measure Properly — The Six Dimensions of Verifiable Quality

The solution cannot be to return to manual development. That would be the wrong response to a real problem. And it makes no sense to impose the same level of design rigour on a temporary prototype as on a platform handling payments, sensitive data, or critical processes. The response must be proportionate to the risk.

For software intended to endure, a mature delivery process should provide **verifiable evidence**, not mere declarations.



Fig. 4 — Verifiable quality checklist. The six dimensions are not an invention to counter AI: they are established principles of software engineering. The novelty is that today we have tools capable of enormously accelerating the production of the evidence that demonstrates them.

FOR NON-TECHNICAL CLIENTS — WHAT TO ASK BEFORE ACCEPTING DELIVERY

You do not need to be a programmer to request evidence proportionate to the risk of your project. Before sign-off, a client can ask:

1. **An independent review:** a code review or assessment conducted by someone who has not simply replicated the generation system's assumptions.
2. **Evidence on critical cases:** acceptance tests covering errors, interruptions, inconsistent data, and exceptional conditions — not only a demonstration of the most favourable scenarios.
3. **The rationale behind key decisions:** a brief record of architectural decisions explaining constraints, alternatives, and trade-offs — not merely a description of classes and APIs.
4. **Proof of recovery:** for services that hold important data, evidence of a genuinely tested restoration — not simply a declaration that backups are taken.
5. **Verified ability to change supplier:** access to the code, dependencies, and documentation needed, accompanied by a proof-of-handover. A different team should be able to start the system, run the tests, and make a small change using the delivered artefacts. Any difficulties encountered and assistance required must be recorded.
6. **A statement of residual risks:** what has been verified, what has not, and what limitations or dependencies remain.

Not every project requires extensive verification: the level of scrutiny should be proportionate to the criticality of the software. But a promise of reliability ought to be accompanied by **verifiable evidence** — not merely green tests or a convincing demonstration.

CHAPTER 06

When Risk Depends on an External Model

It is important to distinguish two different situations. In the first, AI is used to develop the software: the generated code is part of the product, but the model remains a tool in the process. In this case, the responsibility for verification falls on the supplier, who must check what is delivered regardless of how it was produced.

In the second, AI is embedded in the product: the system calls an external model at runtime, and its behaviour also depends on that model. In this case, the training data, internal workings, and limitations of the model may not be fully knowable by whoever integrates the solution. The opacity is not necessarily intentional, but it is real.

This distinction makes both risk assessment and the attribution of responsibilities more complex — across the model provider, the integrator, and the application supplier. The model provider's declarations are one element of the assessment, but they must be supplemented by verification of the specific use: testing on representative and critical cases, version tracking, continuous monitoring, and procedures for handling errors or unavailability.

Some degree of uncertainty may remain. This too must be made explicit to the client, together with the dependencies and the limits of the checks carried out.

CHAPTER 07

When Risk Becomes Liability

The analogy with subprime mortgages is not only about the difficulty of assessing the quality of what one purchases. It also concerns a more concrete question: who is accountable when a risk, barely visible at the point of sale, produces damage?

Directive (EU) 2024/2853[8] on liability for defective products introduces a significant element into this debate: it expressly includes **software** within the definition of product. The Directive entered into force on 8 December 2024, must be transposed by Member States by **9 December 2026**, and the new regime applies to products placed on the market or put into service from that date.

The legislature does not prescribe the adoption of any particular architectural style, nor does it prohibit software from being written, tested, or distributed with the assistance of AI agents. Defectiveness is assessed in relation to the safety that a person is entitled to legitimately expect from the product, taking into account the relevant circumstances, including cybersecurity requirements.

The Directive also provides, under certain conditions, mechanisms relating to the disclosure of evidence and presumptions that may be relevant in technically complex disputes. In such contexts, being able to reconstruct the checks carried out, the known constraints, and the decisions relevant to safety is not merely good engineering practice: it may also help to clarify the facts and responsibilities.

It is important, however, to avoid an improper equivalence: **technical debt does not automatically mean a defective product**. Nor does the Directive render all maintenance costs, project delays, or purely economic losses compensable in themselves. Its scope covers specific categories of damage and particular conditions of liability.

This limitation does not weaken the argument: it makes it more precise. The market may reward ever-shorter delivery times, but speed alone does not demonstrate the safety of the result. **Producing faster does not eliminate the need to be able to verify and explain the decisions upon which the safety of the software depends.**

CONCLUSION

Technical Debt Does Not Disappear. It Changes Creditor.

I do not believe we are necessarily heading towards a new 2008. The subprime crisis was the result of specific financial, regulatory, and macroeconomic conditions. The risks of AI-assisted software are different, and there is currently no demonstration that they could provoke an analogous systemic shock. It would be incorrect to suggest otherwise.

But the crises of the past can teach us to recognise recurring mechanisms: the underestimation of risk, information asymmetry, the separation between those who benefit immediately from a transaction and those who bear its consequences over time.

Technical debt is not a creation of artificial intelligence. It has existed as long as software itself. The novelty is not necessarily the nature of the risk, but **the scale at which we can produce it**. If feature generation accelerates faster than our capacity to verify, understand, and maintain what is built, it becomes possible to accumulate in a short time a quantity of risk that in the past took far longer to build up. The reverse is equally true: well-governed AI tools can accelerate verification, improve documentation, and reduce technical debt. The difference lies in process, incentives, and responsibilities.

AI did not invent technical debt. We have always known projects delivered in haste, insufficient documentation, fragile systems, and poor architectural decisions.

But it could multiply the speed at which we accumulate it, if we do not develop equally rapidly our capacity to prevent, detect, and manage it.

And it could also multiply the speed at which we produce high-quality software, if we learn to use the same tools correctly.

Technology makes both directions possible. What decides which will prevail will not be the capabilities of the models alone. It will be the economic incentives, the professional responsibilities, and the engineering culture we choose to preserve.

> *Are we genuinely reducing the cost of building good software, or are we merely deferring part of its costs into the future?*

There is, however, an objection that deserves to be addressed directly. If AI agents were to become capable enough to make not only production but also the understanding, modification, and rewriting of software economically viable, part of the technical debt could become less burdensome. A system that is difficult to understand could be analysed, rewritten, and replaced in far shorter times than those currently required. The reasoning has its own logic.

Several dimensions nonetheless remain that this prospect does not resolve easily. The business rules embedded in a system — often not written down anywhere, inherited from years of operational decisions — cannot be rewritten correctly without an authoritative source of truth. Data migrations require an understanding of what that data means in the real context of the organisation. Integrations with third-party systems depend on contracts, technical constraints, and implicit behaviours that the code alone may not fully document. And operational continuity requires an understanding of incidents that have occurred in the past, the solutions adopted, and the known fragilities.

The difference is enormous. Because technical debt, like any debt, does not simply disappear because we stop accounting for it. Someone will have to address it — and in the absence of documentation of decisions, business rules, and critical integrations, even a more capable AI agent may need to reconstruct missing information, formulating hypotheses to be verified with those who know the domain and the operations.

> *The future of software should not be a choice between artificial intelligence and engineering. It should be the meeting of the extraordinary productive capacity of the former and the responsibility of the latter.*

The problem is not that AI writes code. **The problem is selling as certainty that which we have not verified.**

Sources and Bibliography

The following sources document the data and historical references used in this essay. The parallel between the financial crisis and the software industry is the author's interpretation, not a conclusion of the cited research.

1 **United States Senate, Permanent Subcommittee on Investigations (2011)**
Wall Street and the Financial Crisis: Anatomy of a Financial Collapse. Analysis of securitisation, conflicts of interest, and risk valuations.
fraser.stlouisfed.org → Wall Street and the Financial Crisis

2 **Cui, Demirer, Jaffe, Musolff, Peng, Salz (2025)**
The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers. Controlled experiments at Microsoft, Accenture and others: +26% tasks completed on average.
mertdemirer.com → Demirer_AI_productivity.pdf

3 **METR (2025)**
Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. 16 experienced developers working on familiar open-source projects: participants took on average 19% *longer* to complete tasks with AI tools — they were slower, not faster. See also the 2026 methodological update (Source 4).
metr.org → early-2025-ai-experienced-os-dev-study

4 **METR (2026)**
We are Changing our Developer Productivity Experiment Design. Methodological update and limitations of estimates for new AI tools (February 2026).
metr.org → uplift-update

5 **Google Cloud DORA (2025–2026)**
State of AI-assisted Software Development and *Balancing AI tensions.* Studies on the impact of AI on software productivity and instability. AI as an amplifier of existing organisational characteristics.
dora.dev/research/2025/dora-report · dora.dev/insights/balancing-ai-tensions

6 **NIST (2022)**
SP 800-218: Secure Software Development Framework. Secure development practices, risk management, and verification throughout the development lifecycle.
csrc.nist.gov → SP 800-218

7 **Akerlof, G. A. (1970)**
The Market for "Lemons": Quality Uncertainty and the Market Mechanism. The Quarterly Journal of Economics, 84(3), 488–500. 2001 Nobel Prize in Economics. Theoretical model of information asymmetry in markets.

8 **European Parliament and Council of the European Union (2024)**
Directive (EU) 2024/2853 on liability for defective products. EUR-Lex, CELEX:32024L2853. In force from 8 December 2024; transposition required by 9 December 2026. In particular Arts. 4 (definitions, including software), 6 (damage), 7 (defectiveness), 8 (responsible economic operators), 9 (disclosure of evidence), 10 (burden of proof and presumptions). eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32024L2853

Cristian Costantini — Software Architect
November 2026 · costantini.me

This document was written with the assistance of artificial intelligence.