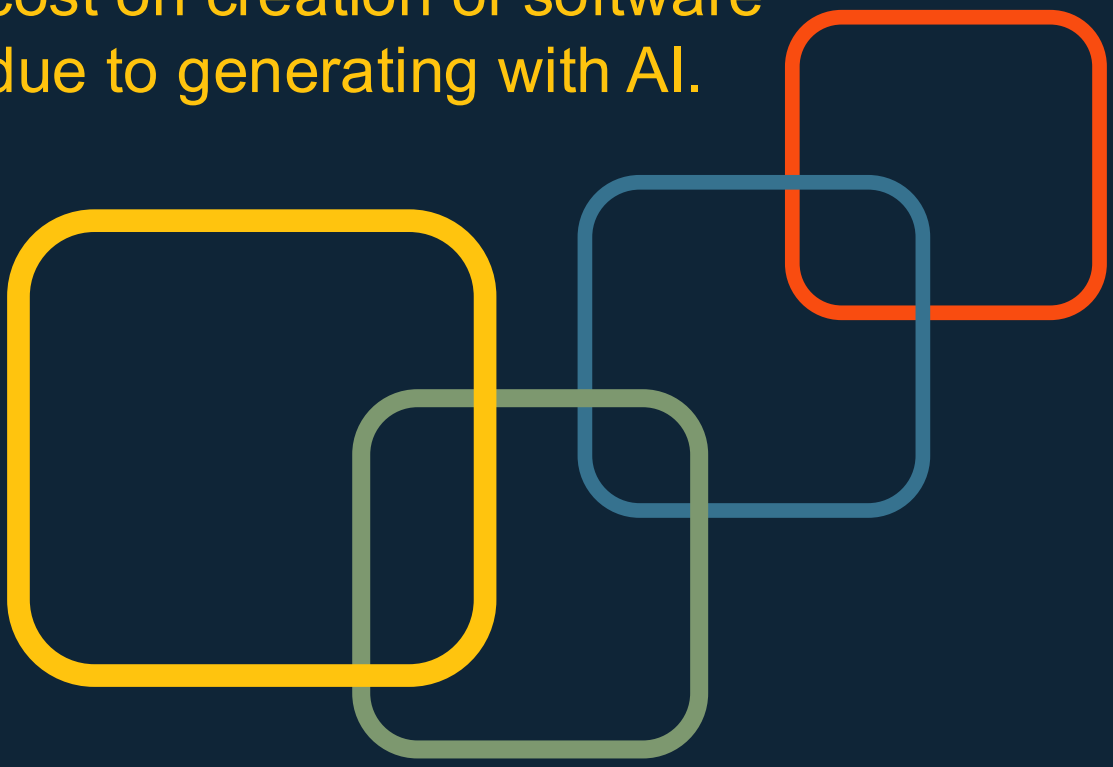
Cost in Software and Collapsing Creation Costs Through AI-Augmented Development

An economic whitepaper on collapsing cost on creation of software due to generating with AI.



Stefan Ellersdorfer

Cost in Software and Collapsing Creation Costs Through AI-Augmented Development

> An economic whitepaper on collapsing cost on creation of software due to generating with AI.

Audience: Executive leadership, engineering leadership, transformation owners
Type: Strategic economic whitepaper
Version: 1.0
Author: Stefan Ellersdorfer
Last updated: 2026-02-16

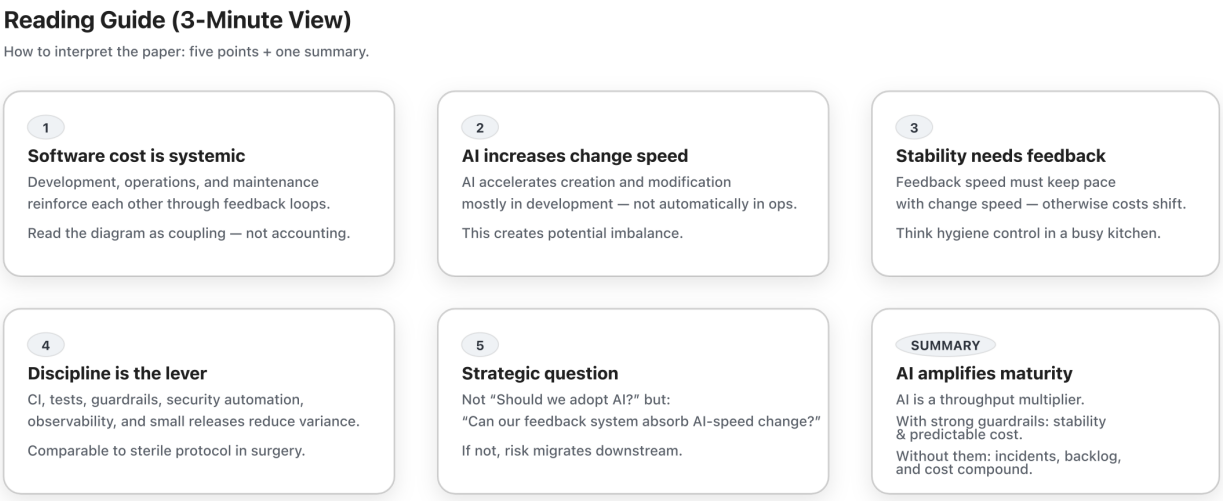


Figure: How to Interpret This Paper in Three Minutes.

AI collapses creation cost only when engineering quality scales with change amplification.

Context: AI Beyond the SDLC

Artificial Intelligence currently influences a wide range of economic activities:

- Generative AI in software development
- AI-supported operations and incident management
- Machine learning in logistics and forecasting
- Neural networks in perception systems
- Model-based simulation in engineering and finance
- AI-enhanced search and knowledge retrieval
- Autonomous agents interacting with digital infrastructure

The technological landscape is broad and continuously evolving.

This paper does not attempt to assess AI in all domains.

Instead, it deliberately focuses on one structural component:

Software engineering within the Software Development Life Cycle (SDLC).

SDLC is only a fraction of overall business economics. Organizations also manage market risk, regulatory exposure, capital allocation, talent constraints, and strategic positioning.

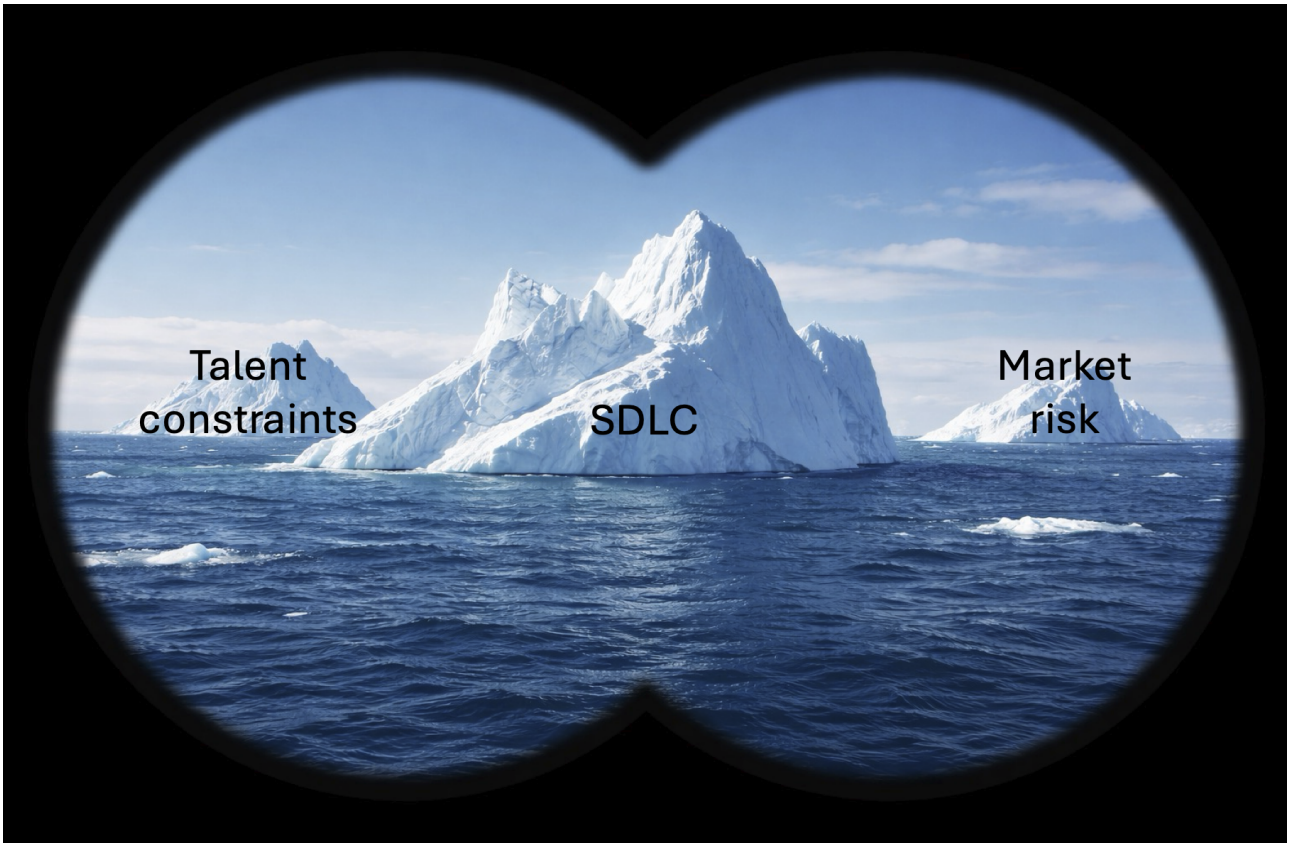


Figure: A Fraction of business economics is the SDLC.

However, SDLC represents a controllable leverage point.

Just as hygiene standards in a professional kitchen do not determine restaurant success alone — yet remain essential for operational stability — disciplined engineering does not guarantee business success, but it remains structurally decisive.

The Three-Part Structure of Software Cost

Software cost is frequently equated with development cost. This simplification obscures the systemic reality.

In practice, cost is distributed across three interconnected domains:

- Development
- Operations
- Maintenance

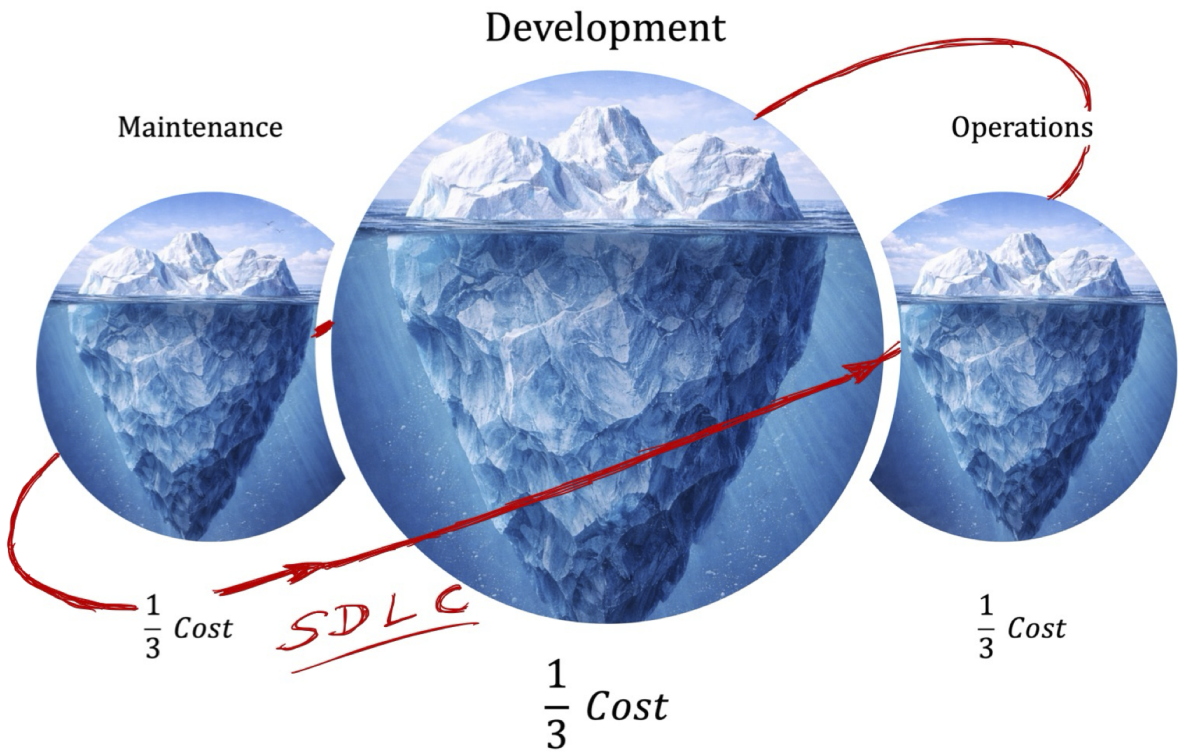


Figure: Cost distribution along the software development lifecycle (SDLC).

Each domain represents roughly one third of total cost. More importantly, they are dynamically coupled.

Decisions made during development directly influence:

- Incident frequency
- Operational workload
- Recovery complexity
- Long-term maintainability
- Change effort

Software behaves less like a factory and more like a hospital: preparation quality determines treatment complexity.

Development is visible.
Operations and maintenance often remain beneath the surface — until instability emerges.

Reinforcing Dynamics

Software systems are governed by reinforcing loops.

Two structural patterns can be observed.

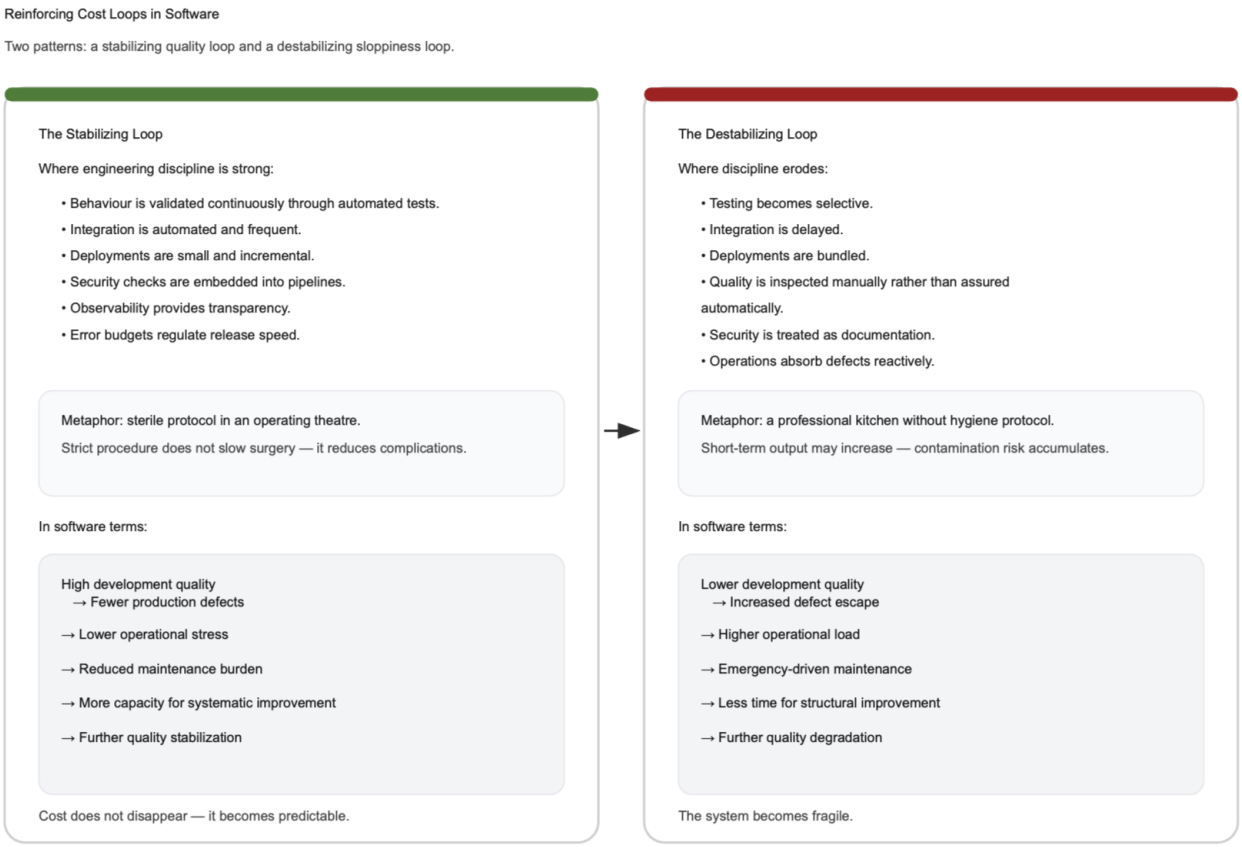


Figure: Reinforcing Cost Loops in Software.

Reinforcing dynamics in software systems are not theoretical constructs. They can be observed in everyday organizational experience.

Software systems are cumulative in nature. Every architectural decision, every shortcut, every abstraction and every dependency becomes part of the system's future. Unlike a physical production process, where defective units can be discarded, software decisions remain embedded in the **structure**. They shape how easily the system can be changed, how predictable its behaviour is, and how much effort future work will require. Over time, these decisions accumulate. Good decisions simplify future change. Poor decisions complicate it. This accumulation creates a compounding effect.

At the same time, software systems rarely become smaller. They grow. Features are added, integrations multiply, compliance requirements increase, dependencies evolve. Growth increases interaction surfaces. With every new interaction, the number of possible failure modes expands. If quality practices are strong, this growth is managed deliberately. If not, complexity begins to

outpace control. Engineers then spend more time understanding the system than improving it. The growing complexity feeds back into slower delivery, more defects, and higher operational stress. What began as a small imbalance becomes self-reinforcing.

Another observable characteristic is delay. Many consequences of development decisions do not appear immediately. A concurrency flaw may only surface under production load. A security weakness may remain invisible until exploited. An architectural shortcut may function for months before scaling exposes its fragility. When feedback is slow, defects accumulate silently. Correction becomes more expensive the later issues are detected. In systems with rapid integration and deployment, feedback interrupts this accumulation early. In systems without it, delay amplifies the reinforcing effect.

Organizational behaviour strengthens these patterns. When incidents increase, engineering time shifts toward firefighting. Preventive work is postponed. Refactoring is delayed. Automation improvements are deprioritized. This reduction in preventive effort further increases the likelihood of incidents. Conversely, when a system is stable, engineers have capacity to improve tests, automation, and architecture. **Stability creates room for further stabilization.** Over time, the organization drifts toward either virtuous or vicious cycles.

These patterns are visible in operational indicators. Rising incident rates, growing maintenance backlogs, declining deployment frequency, or increasing recovery times rarely appear as isolated events. They tend to move together. Organizations often experience extended periods of apparent stability followed by sudden degradation once structural thresholds are crossed. This behaviour is characteristic of reinforcing systems with delayed feedback.

For this reason, reinforcing dynamics in software are observable rather than speculative. They arise from the cumulative nature of code, the steady growth of complexity, and the delayed visibility of consequences. Artificial Intelligence does not create these dynamics. It increases the rate of change within them. Where stabilizing patterns exist, acceleration improves outcomes. Where destabilizing patterns dominate, acceleration intensifies deterioration.

In this sense, AI acts as a gain amplifier within an already reinforcing system.

AI as Throughput Multiplier

Artificial Intelligence changes the pace at which software can be produced. In practical terms, it lowers the effort required to generate code, to refactor existing structures, to implement features, and to explore alternative approaches - architectural but also business wise. Tasks that previously required substantial manual effort can now be executed faster and at scale. The immediate and visible effect is increased development throughput.

What AI does not automatically change, however, is the system’s capacity to absorb that acceleration. Recovery processes do not become faster simply because code was written more quickly. Operational resilience does not improve merely because features were implemented at

higher speed. Organizational learning does not accelerate unless deliberate feedback mechanisms are strengthened. The ability to change a system and the ability to stabilize it are governed by different mechanisms.

This tends to create a structural asymmetry. AI expands the organization’s capacity to introduce change, but it does not proportionally expand its capacity to evaluate, integrate, and stabilize that change. The technical act of modification becomes cheaper; the processes required to ensure safety, coherence, and reliability have always been harder to establish and remain bounded by cognitive and operational constraints.

From an economic perspective, this imbalance is consequential. If change accelerates while absorption capacity remains constant, the system begins to accumulate risk. The cost of instability, incident response, and corrective work may rise even as development effort declines. AI therefore functions less as a universal efficiency engine and more as a throughput multiplier whose economic outcome depends on whether stabilizing mechanisms evolve at the same pace.

In this sense, AI increases the velocity of movement within the system. Whether that movement leads to sustainable progress or structural strain depends on the strength of the feedback structures that surround it.

AI accelerates

- Code generation
- Refactoring
- Feature implementation
- Architectural experimentation

Although it can, it does not automatically increase:

- Recovery speed
- Operational resilience
- Organizational learning capacity

AI automatically expands the ability to change the system.

It does not automatically expand the human capacity to safely absorb large changes.

This asymmetry has economic consequences.

The Stability Condition

Every organization, whether AI-enabled or not, operates under a fundamental structural constraint: the speed at which it introduces change must not exceed the speed at which it can evaluate and absorb that change. In practical terms, feedback must keep pace with modification.

> Feedback speed must be at least equal to change speed.

Software systems evolve continuously. Features are added, interfaces adjusted, dependencies updated, configurations refined. Each modification introduces uncertainty. Feedback mechanisms - automated tests, integration pipelines, observability, incident analysis, user response — exist to reduce that uncertainty. When feedback arrives quickly, deviations are detected early, correction remains inexpensive, and the system maintains equilibrium.

If change begins to outpace feedback, the balance shifts. Modifications accumulate faster than they can be validated. Defects surface later, often under real production conditions. Recovery becomes more complex, and engineering capacity shifts from improvement to repair. Over time, operational stress and economic pressure increase. The issue is not the existence of change, but the mismatch between change and control.

This dynamic can be understood through the metaphor of driving a race car. The physical laws of acceleration and steering do not change when speed increases. However, higher speed demands significantly greater control. Precision in steering becomes more critical. Reaction time shortens. Minor deviations have amplified consequences.

Under close observation, this control is not improvisation. It consists of trained muscular memory, mechanical sympathy with the vehicle, and repeated practice under simulated racing conditions. The driver prepares systematically for high-speed conditions before operating at high speed.

AI increases the velocity of software change in a similar way. The underlying system principles remain unchanged. What changes is the tolerance for imprecision. At higher velocity, small weaknesses in testing, integration, deployment, or observability produce larger consequences. Sustainable acceleration therefore requires strengthened control mechanisms: automated validation, continuous integration, small batch releases, and disciplined operational practice.

Speed itself is not the risk. Insufficient control at high speed is.

When feedback scales with change, acceleration remains manageable. When it does not, instability becomes likely.

If change speed exceeds feedback speed:

- Defects escape detection.
- Incident frequency rises.
- Maintenance backlog grows.
- Operational stress increases.
- Economic pressure intensifies.

If feedback keeps pace:

- Issues are detected early.
- Blast radius remains limited.
- Recovery stays manageable.
- Learning compounds positively.

The Dual Amplifier Effect

AI does not merely accelerate the pace of software development; it alters the scale at which software is produced. It increases the speed of change and lowers the effort required to generate additional functionality. As a result, organizations tend to produce more software and to modify it more frequently. These two effects reinforce one another.

However, frequency alone does not determine stability. High change frequency is not inherently destabilizing. In fact, when changes are small, reversible, and validated immediately, frequent modification increases stability. Small increments reduce uncertainty. Each change has limited scope, limited blast radius, and clear attribution. Feedback arrives quickly. Corrections remain inexpensive.

> Instability emerges when change accumulates.

When multiple modifications are bundled into a single release event, their risks combine. The probability of a successful release becomes the product of the probabilities of each contained change. Even if each individual change is likely to succeed, combining many of them multiplies uncertainty. Risk grows stepwise rather than continuously.

From a probabilistic perspective, the safest number of simultaneous changes introduced into production is one.

One change is observable. One change is attributable. One change is reversible.

When many changes are introduced together, attribution becomes blurred. Root cause analysis slows. Recovery becomes more complex. Feedback loses precision. Operational effort increases.

AI amplifies this dynamic in two ways. It increases the velocity of change and lowers the barrier to producing additional changes. If this leads to larger accumulated batches, instability compounds. If instead AI is embedded within a disciplined system of continuous integration and micro-incremental deployment, risk can remain bounded.

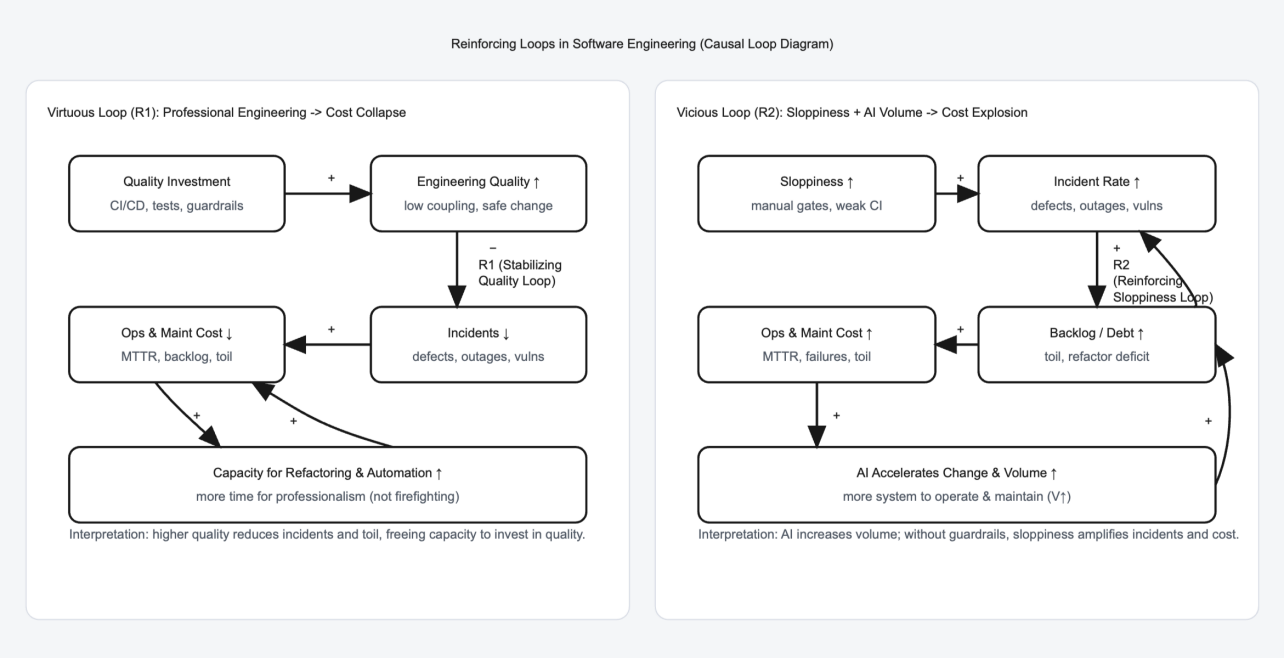


Figure: Reinforcing Loops in Software Engineering.

Operational cost does not scale simply with system size. It scales with defect density, structural complexity, and the interaction between change and validation. When complexity increases and validation weakens, cost grows disproportionately. When complexity increases but changes are introduced in small, continuously validated increments, stability can be maintained.

AI does not remove cost. It redistributes and amplifies it.

Where discipline is strong, amplification strengthens positive reinforcing loops. Where discipline is weak, amplification accelerates fragility.

The decisive variable is not speed itself. It is the size and validation of each step taken at that speed.

AI introduces two amplifiers:

1. Increased change speed.
2. Increased volume of software produced.

> If engineering discipline weakens while throughput rises, cost does not grow linearly.

Development as Leverage Point

This whitepaper concentrates deliberately on development. While artificial intelligence affects many domains within and beyond the software development lifecycle, development occupies a particular structural position within the reinforcing loops that connect development, operations, and maintenance.

Development is where change originates. Architectural decisions are made, abstractions are defined, dependencies are selected, and behaviour is implemented. These decisions shape not only how software functions today, but how it will behave under load, how easily it can be modified, and how costly it will be to maintain. In this sense, development acts as the upstream source of downstream complexity or stability.

The analysis presented here compares development practices with and without AI acceleration. It emphasizes the role of professional engineering discipline: a mindset that treats quality as intrinsic rather than optional, that values incremental progress over large releases, and that integrates validation into everyday work. Practices such as Extreme Programming principles, Test-Driven Development, Continuous Integration, Continuous Delivery, automated guardrails, and embedded security are not presented as methodological preferences. They represent structural mechanisms that regulate change and feedback.

These foundations, articulated in **The Effective Software Engineer**, are not intended as prescriptive doctrine. They describe the conditions under which reinforcing loops remain stabilizing rather than destabilizing. When development is disciplined, defects are detected early, integration remains continuous, and risk is contained. Operational stability benefits accordingly, and maintenance effort remains bounded. The effect propagates forward.

When discipline weakens, the opposite occurs. Shortcuts in development accumulate as operational incidents and maintenance backlog. Recovery becomes reactive rather than controlled. Engineering capacity shifts from improvement to correction. The downstream domains absorb the consequences of upstream decisions.

For this reason, development functions as a leverage point within the broader cost structure. Adjustments made here influence the entire system. Strengthening discipline in development does not eliminate complexity, but it prevents complexity from amplifying uncontrollably. Neglecting discipline does not merely reduce development quality; it reshapes the economic behaviour of the entire software system.

The body of this work examines development with and without AI, emphasizing:

- Professional engineering attitude
- XP principles
- Test-Driven Development
- Continuous Integration
- Continuous Delivery
- Automated guardrails
- Embedded security

These practices reflect the structural foundations described in *The Effective Software Engineer*.

Development is the leverage point within the reinforcing loops.

Neglecting discipline destabilizes them.

Small Increments as Structural Safeguard

Small increments have always been the most effective structural safeguard in software engineering.

Independent of AI, the safest way to evolve a complex system is to introduce change in minimal, clearly attributable units. A single, well-defined modification is observable, testable, and reversible. Its effects can be evaluated without ambiguity. If an issue arises, causality is clear. Recovery is straightforward.

Large batches behave differently. When multiple changes are introduced simultaneously, uncertainty compounds. Attribution becomes blurred. Diagnosis requires more effort. Rollback becomes broader and riskier. Even when individual changes are sound, their interaction may produce unintended side effects. The probability of success decreases as batch size increases, not because engineers are careless, but because complexity multiplies.

This structural property predates artificial intelligence. It is inherent to complex adaptive systems.

AI does not change this principle. It intensifies it.

Because AI lowers the effort required to generate change, it increases the volume and velocity at which modifications can be produced. The temptation to accumulate changes before release grows accordingly. When batch size expands in parallel with throughput, systemic risk increases.

The human capacity to absorb change does not scale at the same rate as machine-assisted generation. Cognitive load, coordination effort, and operational resilience remain bounded. At higher velocity, small imprecisions have amplified consequences.

Therefore, what has always been good engineering practice becomes essential under AI acceleration. Micro-incremental deployment preserves control. It limits blast radius. It maintains clear feedback loops. It ensures reversibility. It prevents uncertainty from accumulating invisibly.

Human organizations struggle with large blast radii.

Large batch releases increase:

- Cognitive load
- Rollback complexity
- Incident impact
- Recovery time

Micro-incremental releases reduce:

- Risk surface
- Feedback latency
- Coordination overhead
- Operational stress

> Small increments are not a response to AI. They are a structural principle of stable system evolution. AI increases the cost of ignoring that principle.

Economic Implications

When organizations introduce AI into software development without simultaneously strengthening their feedback systems, the initial effect often appears positive. Development output increases. Features are delivered faster. Refactoring becomes easier. Backlogs seem to shrink. From a narrow development perspective, productivity improves.

However, the broader economic picture unfolds differently over time.

If validation, integration, and operational safeguards do not scale with the increased rate of change, defects begin to escape detection more frequently. Incident rates rise. Maintenance work accumulates. Operational teams absorb the consequences of accelerated throughput. Engineering capacity gradually shifts from systematic improvement toward reactive stabilization.

What appeared as efficiency in development does not vanish — it migrates.

The reduction in development effort reappears in operations and maintenance. Costs shift downstream, often with amplification due to delayed feedback and accumulated complexity.

In organizations where engineering discipline is strong, the trajectory differs. Automated testing, continuous integration, embedded security, and micro-incremental deployment ensure that increased change velocity remains bounded by rapid feedback. Development effort compresses, but operational stability does not deteriorate. Maintenance effort remains predictable because complexity is managed deliberately. In such environments, AI strengthens an already stabilizing system.

Where discipline is weak, the opposite pattern emerges. Throughput increases, but validation lags behind. Complexity grows faster than control mechanisms can compensate. Stability decreases gradually at first, then more abruptly once structural thresholds are crossed. Operational effort rises disproportionately. Maintenance backlog compounds. Economic pressure intensifies.

AI does not determine which of these trajectories unfolds.

It amplifies the existing maturity of the system.

> If reinforcing loops are stabilizing, AI accelerates improvement. If reinforcing loops are destabilizing, AI accelerates deterioration.

The following system dynamics models illustrate these patterns. They compare cost growth and feature growth over time under different quality regimes — average practice, deliberate quality investment, and sloppiness — each observed with and without AI acceleration. The diagrams make visible what is otherwise difficult to observe in short-term financial reporting: the long-term divergence between disciplined and undisciplined systems under amplified change.

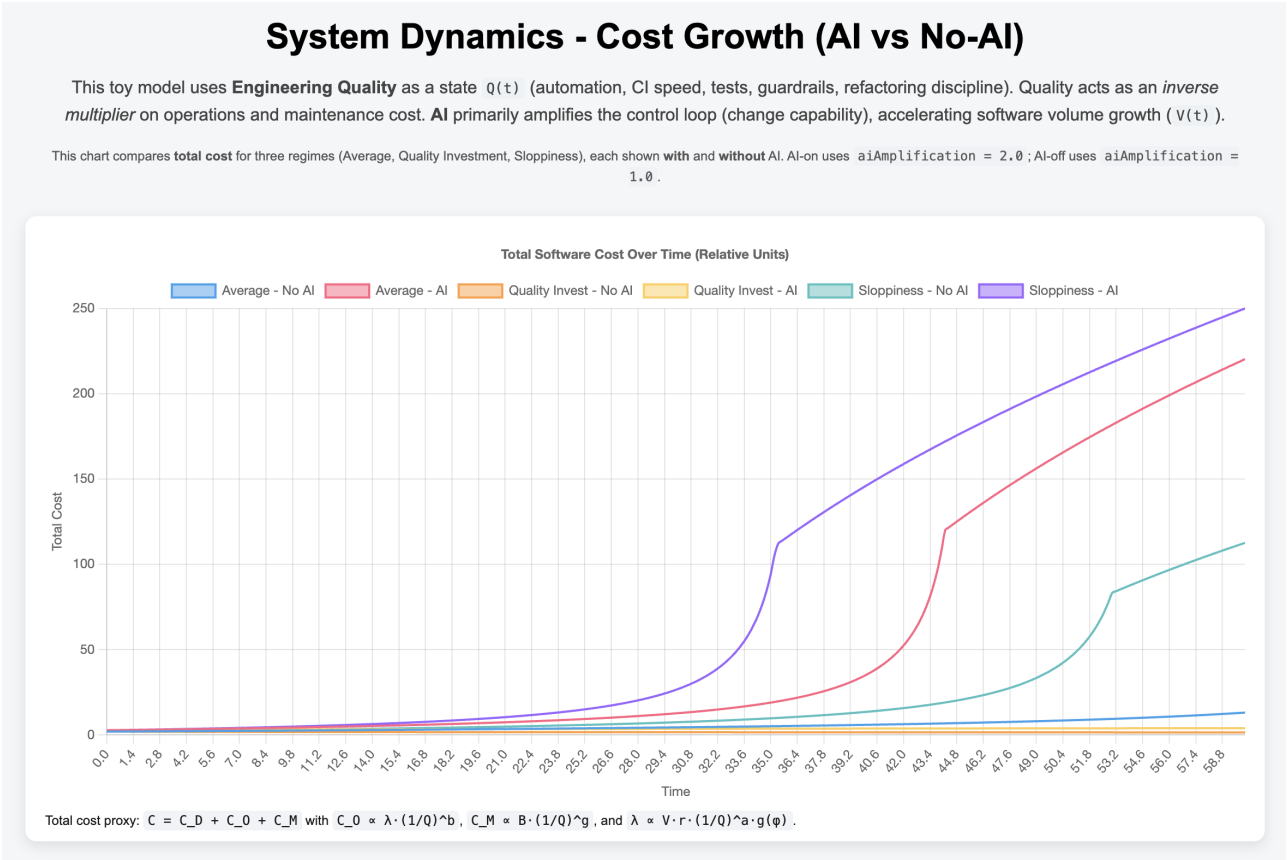


Figure: Cost growth (AI vs No-AI). Illustrative model — relative units for structural comparison.

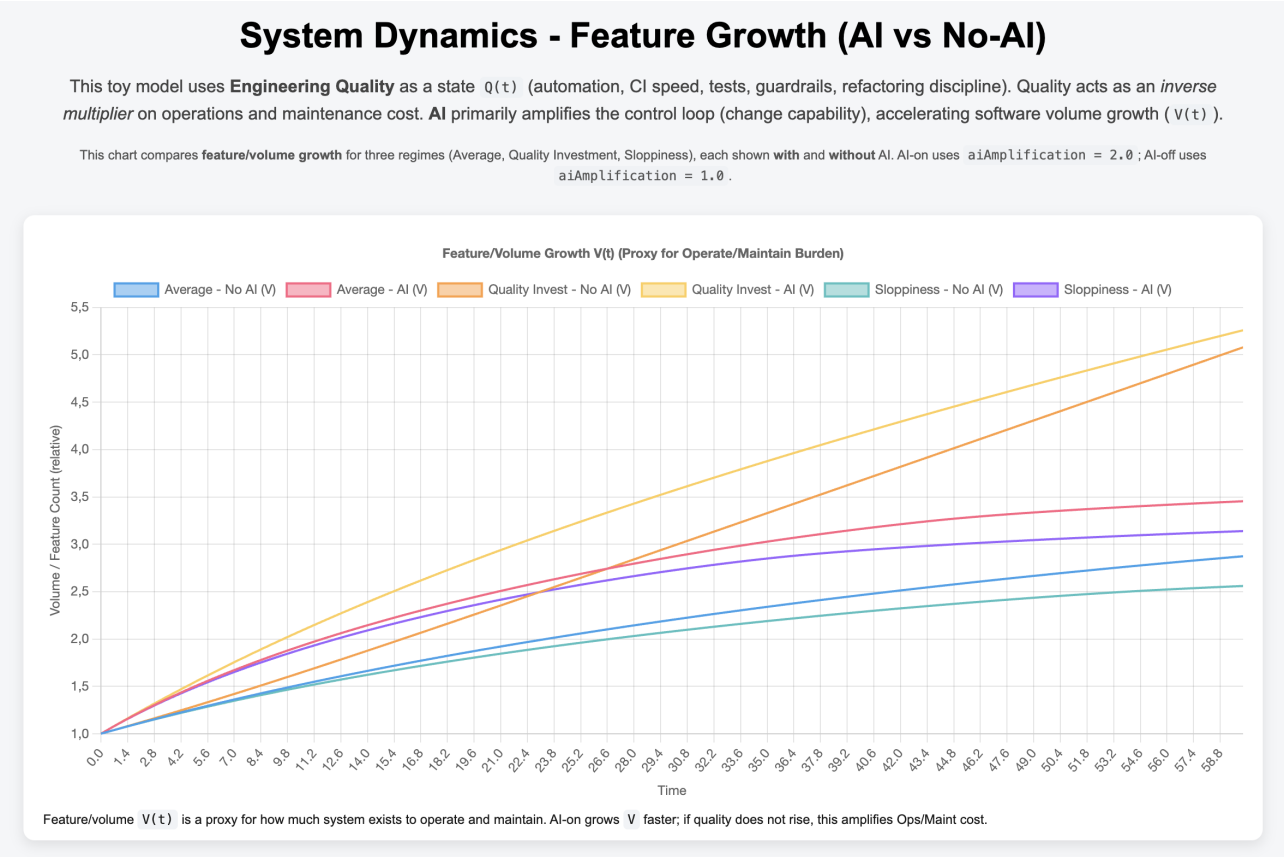


Figure: Feature growth (AI vs No-AI). Illustrative model — relative units for structural comparison.

The AI Stability Equation

Summary Formula

$$C_T \propto V \cdot \alpha \cdot (1 / Q)^a$$

Where:

- V = system volume (feature surface)
- α = AI change amplification
- Q = engineering quality
- a > 1 = fragility exponent, implies nonlinear fragility amplification

---

Interpretation

AI multiplies change rate.

Engineering quality multiplies cost resilience.

If AI doubles change rate (α = 2) while Q remains constant and a = 1.5, cost increases by approximately:

$$2 \times (1/Q)^{1.5}$$

If Q = 0.8 and a = 1.5, then cost ≈ 2 × (1/0.8)^1.5 ≈ 2 × 1.40 ≈ 2.8x baseline.

Strategic Implication

If:

Q ↑ proportionally to α

→ Total cost stabilizes or collapses.

If:

α ↑ while Q stagnates

→ Total cost grows nonlinearly.

Translation

AI is safe only when feedback and engineering quality scale at least as fast as AI increases throughput.

AI without guardrails is an economic multiplier of fragility.

ROI Framing Through DORA Performance Indicators

From a financial perspective, the economic impact of AI in software engineering can be evaluated through DORA performance indicators: deployment frequency, lead time for changes, change failure rate, and mean time to recovery (MTTR). AI predictably improves deployment frequency and reduces lead time by accelerating code generation and modification.

However, unless engineering quality and feedback systems scale proportionally, change failure rate increases and MTTR deteriorates, offsetting development gains with operational cost escalation.

High-performing organizations — those with strong automated validation, continuous integration, and disciplined release practices — can leverage AI to improve the first two metrics without degrading the latter two. In such systems, AI increases throughput while maintaining reliability, resulting in measurable economic return.

In low-maturity environments, AI improves speed metrics while destabilizing reliability metrics, shifting cost downstream and reducing net ROI.

AI therefore acts as a multiplier of existing DORA maturity: it compounds the financial advantages of elite performers and amplifies the instability costs of underperforming systems.

AI as Quality Amplifier

Public discourse around Artificial Intelligence in software engineering focuses predominantly on its ability to generate code. This framing is incomplete.

AI undeniably increases the speed at which software can be written, refactored, and extended. Yet the same underlying capabilities—pattern recognition, context analysis, structured reasoning, and automated synthesis—can also be applied to strengthen validation, architectural coherence, and workflow discipline.

In this broader perspective, AI is not merely a throughput multiplier. It can become a feedback multiplier.

The economic significance of this distinction is substantial. When AI is used exclusively to increase output, it raises the rate of change without proportionally strengthening control mechanisms. When AI is deliberately embedded into validation and governance workflows, it increases the gain of the stabilizing loops within the system.

This shift is not conceptual; it is practical.

AI can assist in generating and refining executable specifications. It can enforce test-first development patterns. It can continuously evaluate architectural consistency against defined constraints. It can review changes for policy compliance. It can generate and verify documentation. It can orchestrate structured approval flows that prevent unvalidated changes from entering production. It can monitor pipeline signals and surface anomalies before they propagate.

In effect, AI can operate as an automated layer of engineering hygiene.

Structured, multi-stage workflows supported by AI illustrate this approach. Rather than allowing unrestricted generation of code, such systems embed guardrails, approval gates, architectural validation, and test enforcement directly into the development process. They treat AI not as a replacement for discipline, but as an instrument for scaling discipline.

The critical economic metric is not the volume of code produced. It is the ratio between validated change and unvalidated change.

If AI increases both generation and validation capacity proportionally, the reinforcing loops remain stabilizing. If AI increases generation without increasing validation, destabilization follows.

> A practical implication follows from this observation. If an organization has not yet internalized engineering discipline—if validation is inconsistent, integration is manual, architectural boundaries are loosely enforced, and deployment lacks structured feedback—then introducing AI as a primary driver of change is premature. Acceleration without embodied discipline amplifies weakness rather than strength.
>
> Engineering maturity cannot be substituted by tooling. AI can scale a discipline that already exists. It cannot create that discipline in its absence.
>
> For this reason, organizations that have not yet established automated validation, continuous integration, embedded security, and micro-incremental deployment should treat these capabilities as foundational investments. Only once these stabilizing mechanisms are operational does it become economically rational to amplify change velocity through AI.
>
> In other words, AI should not be the first step toward engineering excellence. It should be the multiplier applied after excellence has begun to take structural form.

In this sense, AI-supported development frameworks represent an architectural response to accelerated change. They attempt to ensure that the same intelligence that produces modifications also strengthens the mechanisms that evaluate and constrain those modifications.

The question is therefore not whether AI writes code.

The question is whether AI is positioned within the system as a producer of change only, or as a guardian of structural integrity as well.

Only in the latter configuration does acceleration translate into sustainable economic advantage.

Conclusion

Artificial Intelligence alters the economics of software engineering in a fundamental way. It increases both the speed at which change can be introduced and the volume of software that can be produced. The technical capability to modify systems expands rapidly.

The structural limits of human organizations, however, do not expand at the same rate. Cognitive capacity, coordination ability, operational resilience, and recovery mechanisms remain bounded. Software systems continue to obey the same laws of complexity, probability, and feedback as before. What changes is the velocity at which those laws operate.

For this reason, the central question is not whether AI should be adopted. The relevant question is how the organization structures its feedback systems once change accelerates.

Automation, continuous integration, embedded security, micro-incremental deployment, and systematic validation have long been considered engineering best practices. In an AI-accelerated environment, they cease to be optimizations. They become prerequisites for economic stability. Without them, increased throughput translates into accumulated uncertainty and downstream cost.

In professional kitchens, hygiene is not a matter of preference. It is a structural requirement for safe operation. In operating theatres, sterility is not optional; it is a condition for survival. The same principle applies in AI-enabled software engineering. Discipline in engineering practice is not an aesthetic choice. It is the mechanism that prevents acceleration from turning into instability.

AI does not guarantee cost reduction. It magnifies the structural properties of the system in which it operates. Where discipline is present, amplification strengthens stability. Where discipline is absent, amplification compounds fragility.

Acceleration without control does not reduce cost.

It redistributes and multiplies it.

Smarter Software

Stefan Ellersdorfer is the author of *The Effective Software Engineer* and one of the Managing Directors of **Smarter Software**, a consultancy focused on sustainable software delivery, technical excellence, and organizational effectiveness.

While leading Smarter Software, Stefan continues to work hands-on alongside software engineers, architects, and product teams. He deliberately remains close to day-to-day engineering work—pairing, reviewing code, facilitating technical decision-making, and supporting teams in real delivery environments.

This ongoing involvement ensures that his perspective as an author and consultant is rooted in practical experience rather than theory alone.
His work is strongly influenced by modern engineering practices such as Test-Driven Development, Continuous Delivery, and socio-technical systems thinking.

Drawing from both regulated enterprise contexts and fast-moving product organizations, Stefan focuses on what enables software teams to build systems that are not only correct, but easy to change, resilient over time, and humane to work with.

This whitepaper reflects that stance: pragmatic, experience-based, and written from within the reality of software development—not from a distance.