The Unsolvable Legacy Problem: A Systemic Crisis in Software Engineering

J.Konstapel Leiden 25-9-2025.

Abstract

The software engineering industry faces an unprecedented crisis that transcends traditional "legacy system" challenges. This essay examines evidence suggesting that technical debt and software entropy are growing exponentially, potentially outpacing our ability to modernize systems. Through analysis of recent industry reports, research findings, and case studies, we demonstrate that current approaches to legacy modernization may be fundamentally inadequate to address what appears to be a systemic problem rooted in the physics of software entropy and accelerated by artificial intelligence adoption.

Introduction

Legacy software has traditionally been viewed as an operational challenge—old systems that need periodic updating or replacement. However, emerging evidence suggests we are confronting a fundamentally different phenomenon: a systemic crisis where software entropy grows faster than our capacity to manage it. This essay argues that the "legacy problem" has evolved into something potentially unsolvable through conventional means, requiring a paradigm shift in how we approach software development and maintenance.

The scope of this crisis extends beyond individual organizations or technologies. As Forrester analyst Carlos Casanova warns, we are witnessing "a perfect storm of technology growing, companies being far more distributed and AI coming into the equation, which will make the problem exponentially worse" (Forrester, 2024). This exponential characteristic distinguishes the current situation from previous challenges in software engineering.

The Current State: A Perfect Storm

The Scale of the Crisis

Recent research reveals the magnitude of this challenge. A 2024 Forrester report predicts that more than 50% of technology decision-makers will experience technical debt at "moderate or high levels of severity" by 2025, with this figure projected to reach 75% by 2026 (CFO Dive, 2024). This represents not gradual degradation but exponential growth in system complexity and maintenance burden.

McKinsey's analysis shows that technical debt now consumes up to 40% of entire technology estates, with more than 50% of companies dedicating over a quarter of their IT budgets to managing this debt (McKinsey, 2023). These figures represent a fundamental shift in resource allocation, where maintenance overwhelms innovation.

The Quality Collapse

Denis Stetskov's analysis of the "Great Software Quality Collapse" provides concrete examples of this crisis. He documents how Apple's Calculator application leaked 32GB of RAM—a basic utility consuming more memory than sophisticated applications of previous generations (Medium, 2024). This exemplifies how modern software development practices, while enabling rapid feature delivery, have normalized resource inefficiency and quality degradation.

The July 2024 CrowdStrike incident serves as a stark reminder of systemic fragility. A single configuration file missing array bounds checking crashed 8.5 million Windows computers globally, disrupting emergency services and critical infrastructure (Built In, 2024). Such incidents represent the compound effect of accumulated technical debt across interconnected systems.

The Physics of Software Entropy

Lehman's Laws Revisited

Manny Lehman's laws of software evolution, formulated in 1974, provide a theoretical foundation for understanding why this crisis may be inevitable. Lehman observed that complex software systems require continuous modifications to remain relevant, and such modifications increase system entropy unless specific work is done to reduce it (Wikipedia, 2025).

The second law of software entropy parallels thermodynamics: entropy in an isolated system always increases unless energy is applied to maintain order. As software systems evolve, they naturally tend toward greater complexity and decreased maintainability (Boost, 2017). This physical principle suggests that software degradation is not a management problem but a fundamental characteristic of complex systems.

The Abstraction Tax

Modern software development relies heavily on abstraction layers that simplify development but compound performance overhead. A typical modern application stack might include: React → Electron → Chromium → Docker → Kubernetes → Virtual Machine → Managed Database → API Gateways. Each layer adds approximately 20-30% overhead, resulting in 2-6x resource consumption for equivalent functionality compared to direct implementations (Medium, 2024).

This "abstraction tax" creates a compound effect where each generation of software requires exponentially more resources to deliver similar value. The abstraction layers that enable rapid development today become the performance bottlenecks and maintenance burdens of tomorrow.

The AI Acceleration Factor

The Democratization Paradox

Artificial Intelligence tools have democratized software development, enabling non-technical individuals to generate code. However, this democratization has introduced new sources of technical debt. Studies show that junior developers using AI-assisted tools cause damage 4x faster than without such assistance, while 70% of hiring managers trust AI output more than junior developer code (Built In, 2024).

The Synopsys 2024 Open Source Security and Risk Analysis report reveals that nearly three-quarters of commercial codebases contain high-risk vulnerabilities, with a sharp increase attributable to contributions from less experienced developers using AI tools (Built In, 2024). This suggests that AI, while increasing development velocity, may be systematically degrading code quality at scale.

The Generation and Management Paradox

AI presents a fundamental paradox in technical debt management. McKinsey analysts note that "generative AI is leading to a classic catch-22. On the one hand, it is creating new technical debt. On the other hand, when used appropriately, generative AI can help manage tech debt remediation as well as minimize tech debt creation" (CFO Dive, 2024).

This paradox illustrates the complexity of the current crisis. The same technology that promises to solve our technical debt problems is simultaneously accelerating their creation, potentially faster than it can resolve them.

Why Traditional Solutions Are Failing

The Mathematics of Modernization

Traditional approaches to legacy modernization assume that technical debt can be paid down through focused effort and investment. However, Carnegie Mellon University research from 2015 found that much of current technical debt has persisted for over a decade, with architectural issues being the most significant and persistent source (vFunction, 2024).

If technical debt that is 10+ years old persists despite ongoing modernization efforts, and new debt accumulates exponentially due to AI-accelerated development and complexity growth, then the mathematical relationship becomes unsolvable: debt accumulation rate > debt resolution rate.

The Developer Paradox

The software industry faces a counterintuitive talent crisis. Despite 112,720 computer science graduates in 2023 (a 4.3% increase), unemployment among new CS graduates reached 7.8% (Medium, 2025). This occurs because entry-level positions are increasingly automated through no-code platforms and AI tools, while the remaining positions require experience that new graduates cannot obtain.

This creates a systemic problem where the industry needs experienced developers to manage complex technical debt, but the pathway to developing such expertise is being eliminated by the same technologies driving debt accumulation.

The Wrapper Trap

The most common "solution" to legacy system integration is wrapper-based modernization—encasing old systems in modern interfaces. However, as noted in the ABN AMRO case study, this approach creates ideal conditions for security vulnerabilities. Wrappers hide underlying system complexity without addressing fundamental architectural problems, often making systems less secure and more difficult to maintain long-term.

Open Source: From Solution to Problem

The Heritage Loss

Even foundational open-source projects suffer from software entropy. The original SHRDLU program, a landmark in natural language processing, "cannot be run on any modern-day computer or computer simulator, as it was developed during the days when LISP and PLANNER were still in development stage and thus uses non-standard macros and software libraries which do not exist anymore" (Wikipedia, 2025).

This phenomenon extends beyond historical artifacts. Modern open-source components routinely become unmaintainable as their dependency chains evolve, maintainers move on, or underlying platforms change. The very solutions we implement today to solve legacy problems become tomorrow's legacy problems.

The Maintenance Burden Transfer

Open-source adoption often represents a transfer of maintenance burden rather than its elimination. Organizations adopting open-source rich text editors, for example, carry approximately 29% ongoing technical debt compared to the industry average of 18%, while dedicating only 31% of resources to growth activities compared to the industry average of 40% (TinyMCE, 2024).

This data suggests that relying on open-source components may accelerate short-term development while increasing long-term maintenance burden, contributing to rather than solving the legacy crisis.

Proposed Solutions: A Paradigm Shift

From Technical Debt to Technical Wellness

Deloitte proposes abandoning the "debt" metaphor in favor of "technical wellness"—treating technology systems like biological systems requiring preventive care rather than reactive fixes (Deloitte, 2024). This approach advocates for:

1. **Holistic Health Assessments**: Regular comprehensive evaluations of entire technology stacks rather than isolated system fixes
2. **Preventive Maintenance**: Proactive identification and remediation of potential issues before they become critical
3. **Integrated Treatment Plans**: Coordinated modernization efforts across all system components

Engineering Fundamentals Renaissance

Multiple sources advocate for returning to fundamental engineering principles that have been abandoned in favor of rapid development:

Quality Over Velocity: Organizations must "accept that quality matters more than velocity. Ship slower, ship working. The cost of fixing production disasters dwarfs the cost of proper development" (Medium, 2024).

Resource Efficiency Metrics: Measure actual resource usage rather than feature delivery. Applications using 10x more resources than previous versions for identical functionality represent regression, not progress.

Fundamental Skills: Reintroduce array bounds checking, memory management, and algorithm complexity analysis as core competencies rather than "legacy" skills.

Architectural Observability

vFunction advocates for "architectural observability"—systematic understanding and analysis of software architecture at its fundamental level to identify entropy patterns before they become critical (vFunction, 2024). This includes:

1. **Automated Architecture Analysis**: Tools that continuously monitor system structure and dependency evolution
2. **Entropy Measurement**: Quantitative metrics for system complexity and maintenance burden
3. **Predictive Maintenance**: Early warning systems for architectural decay

Developer Velocity Optimization

McKinsey's Developer Velocity Index research identifies four critical factors for sustainable software development:

1. **Tools**: Best-in-class development tools are the top contributor to business success, yet only 5% of executives recognize this link
2. **Culture**: Environment that empowers developers and removes friction points
3. **Product Management**: Alignment between technical and business objectives
4. **Talent Management**: Sustainable approaches to developer recruitment and retention

Systemic Solutions: Beyond Traditional Approaches

Regulatory and Industry Standards

The crisis may require industry-wide standards and regulations similar to those in other engineering disciplines:

Software Engineering Licensing: Professional licensing requirements for software architects and senior developers, similar to civil engineering

Quality Standards: Mandatory performance and security baselines for critical software systems

Liability Frameworks: Legal accountability for software quality in critical applications

Economic Model Transformation

Current software business models incentivize rapid feature delivery over long-term maintainability. Alternative models might include:

Total Cost of Ownership Pricing: Software licensing based on long-term maintenance requirements rather than initial purchase price

Technical Debt Insurance: Financial instruments that incentivize quality development practices

Sustainability Metrics: Business valuation models that account for technical debt as a liability

Education System Reform

Computer science education must evolve to address the entropy crisis:

Maintenance-First Curriculum: Teaching software maintenance and evolution as primary skills rather than secondary concerns

System Thinking: Emphasis on understanding complex system interactions rather than isolated component development

Quality Engineering: Treating software quality as an engineering discipline with measurable outcomes

Case Study: Learning from Failure

The ABN AMRO Merger (1991)

The 1991 merger of ABN and AMRO banks provides a historical example of how technical integration challenges can threaten organizational survival. Hans Konstapel, who worked directly on the merger's IT integration, documented how political decisions over technical considerations led to choosing the worst software components from both banks, including undocumented load-files that even IBM couldn't manage.

The merger "brought the company to the brink of collapse" due to legacy system incompatibilities and integration challenges (Konstapel, 2023). This case illustrates how legacy problems scale with organizational complexity and political constraints.

Konstapel's subsequent accurate predictions about the Fortis/ABN AMRO merger failures demonstrate that these challenges are predictable and systemic rather than isolated incidents.

Lessons Learned

1. **Technical Expertise Must Drive Decisions**: Political and business considerations alone are insufficient for complex system integration
2. **Documentation is Critical**: Undocumented systems become unmaintainable within years, not decades
3. **Integration Complexity Scales Non-linearly**: Merging two complex legacy systems often creates problems greater than the sum of individual system complexities

The Path Forward: Accepting Fundamental Limits

Embracing Constraints

Rather than fighting software entropy, successful organizations may need to accept it as a fundamental constraint and design accordingly:

Planned Obsolescence: Designing systems with explicit lifecycle limits and replacement schedules

Entropy Budgets: Allocating specific resources to entropy management as a core operational expense

Modular Decomposition: Architecting systems for easy component replacement rather than long-term stability

Sustainable Development Practices

Quality Gates: Mandatory quality thresholds that cannot be bypassed for schedule pressures

Technical Debt Limits: Quantitative limits on acceptable technical debt levels with automatic enforcement

Refactoring Cycles: Regular, scheduled refactoring as part of normal development process rather than exceptional activity

Organizational Changes

Executive Education: Technology literacy programs for business leaders to understand technical debt implications

Incentive Alignment: Compensation structures that reward long-term system health over short-term feature delivery

Cross-functional Integration: Breaking down silos between business and technical decision-making

Conclusion

The evidence suggests we are confronting a crisis that transcends traditional legacy system challenges. Software entropy, accelerated by AI democratization and complex abstraction layers, may be growing faster than our capacity to manage it through conventional modernization approaches.

This crisis is characterized by:

1. **Exponential Growth**: Technical debt accumulation rates that exceed management capabilities
2. **Systemic Nature**: Problems embedded in fundamental development practices rather than isolated to specific technologies
3. **Physical Constraints**: Entropy increases that follow thermodynamic principles
4. **Compounding Effects**: AI and complexity layers that accelerate rather than solve underlying issues

Traditional solutions—modernization projects, wrapper-based integration, and incremental refactoring—may be fundamentally inadequate to address exponential growth in system complexity. The industry requires a paradigm shift toward sustainable development practices that treat entropy as a fundamental constraint rather than a manageable problem.

Success may depend on accepting that software systems, like biological systems, have natural lifecycles and require continuous energy investment to maintain order. Organizations that embrace this reality and architect their systems accordingly may achieve sustainable competitive advantages over those that continue pursuing traditional modernization approaches.

The legacy crisis is not just a technical challenge—it represents a fundamental test of the software industry's ability to evolve beyond growth-at-any-cost models toward sustainable engineering practices. The stakes are high: as software becomes increasingly critical to global infrastructure, the cost of failure extends far beyond individual organizations to society itself.

The question is not whether we can solve the legacy crisis, but whether we can evolve our engineering practices fast enough to manage entropy before it manages us.

References

Built In. (2024). "The Software Industry Is Facing an AI-Fueled Crisis. Here's How We Stop the Collapse." September 4, 2024.

Boost. (2017). "Technical debt and entropy: what the laws of physics teach us about refactoring code."

CFO Dive. (2024). "AI rush is fueling tech debt 'tsunami': Forrester." November 26, 2024.

Deloitte. (2024). "Core workout: From technical debt to technical wellness." June 11, 2024.

Forrester. (2024). "The State Of Technical Debt In The US, 2024."

Konstapel, H. (2023). "Over de Legacy van de Legacy Software (Deel 2)." Hans Konstapel Blogs, April 8, 2024.

Konstapel, H. (2008). "Why I have Decided to Leave ABN AMRO." Hans Konstapel Blogs, September 29, 2008.

McKinsey & Company. (2023). "Breaking technical debt's vicious cycle to modernize your business." April 25, 2023.

McKinsey & Company. (2020). "Developer Velocity: How software excellence fuels business performance." April 20, 2020.

Medium. (2024). Stetskov, D. "The Great Software Quality Collapse: How We Normalized Catastrophe." QuillTech, September 2024.

Medium. (2025). Omalley, F. "The Decline of Traditional Coding: A Crisis in the Software Development Sector?" Digital Society, March 20, 2025.

TinyMCE. (2024). "Opportunity cost of technical debt | TinyMCE White Paper."

vFunction. (2024). "How to Manage Technical Debt in 2025." July 21, 2025.

Wikipedia. (2025). "Software entropy." Accessed January 2025.

Wikipedia. (2025). "Technical debt." Accessed January 2025.