Technical Debt Is a People Problem, Not Just a Code Problem.

Every $1 spent on quality training saves $10 in debt remediation

The Sprint Review That Exposed Everything

A development team gathered for their quarterly sprint review. The numbers were sobering. They had delivered just 60% of their committed story points. The remaining 40% had evaporated into fixing bugs, refactoring tangled code, and wrestling with dependencies that should never have been coupled in the first place.

The same conversation repeated every quarter.

"We need to move faster."

"We need to reduce technical debt."

"We need to hire better."

But the real problem wasn't the code. It was the people writing it.

Research consistently shows that project failures stem predominantly from people-related challenges rather than technical issues [1]. Yet most organizations continue to treat technical debt as a code problem, when it's fundamentally a people problem.

The Two Faces of Technical Debt

Technical debt manifests in two distinct ways, and understanding the difference is critical.

Code Debt refers to issues like flawed logic, inefficient algorithms, or poorly implemented patterns. These are coding-level shortcuts that create friction in day-to-day development. They're visible in the codebase and relatively straightforward to fix.

Architectural Debt is more systemic—like a poorly planned city road network that causes traffic jams everywhere. These structural flaws ripple through the system, making them harder to pinpoint and fix. Gartner predicts that by 2026, 80% of technical debt will stem from architectural issues.

Research shows that identified technical debt accounts for about half of a project's total maintenance costs, with the cost proportion of the most expensive Top-5 debt items being 1.8x more than the file proportion they contain [5]. Almost all covered maintenance costs persist as technical debt in the future, with an average increase of 6.8% between the last two releases.

The financial impact is staggering. Technical debt could account for 20–40% of an organization's technology assets. The average global enterprise wastes more than $370 million annually due to their inability to efficiently modernize outdated, inefficient legacy systems and applications [3]. Seventy-eight percent of IT decision makers agree that the time, money, and effort spent maintaining legacy

applications could be spent more productively on other projects that could make the business more effective [4].

The Human Sources of Technical Debt

Academic research has identified three primary categories of human factors that influence debt accumulation [1]:

Personal Essence and Growth

Individual capabilities, experience levels, and learning trajectories. Junior developers are more likely to introduce unintentional technical debt due to their lack of experience. They "tend to focus on the here and now, and solve today's requirement" without considering how the system will evolve.

Collaborative Perseverance

How teams work together, communicate, and share knowledge. When code reviews are rushed or nonexistent, debt slips through. When teams lack psychological safety, concerns go unraised, and debt compounds silently.

Organizational Dynamics

Leadership decisions, prioritization frameworks, and resource allocation. When leadership refuses to make hard tradeoffs, engineers are forced to build "flexible" systems that lack a clear core. That complexity becomes debt. When technical debt decisions are made by business managers who see short-term benefits while developers deal with the long-term negative consequences, debt accumulates unchecked.

A study analyzing 72 primary studies on human factors linked to technical debt identified 67 distinct human aspects, with time pressure being the most cited factor and attitude being the most frequently used category [5].

The Junior Developer Debt Trap

The data on junior developers and technical debt is clear and sobering.

Less experienced developers often unintentionally accumulate technical debt due to their lack of experience. They are less able to project outcomes to the future about how the system is likely to evolve, which means they tend to focus on the here and now and solve today's requirement rather than preparing the system for what is likely to come [1].

This creates what researchers call "Reckless-Inadvertent debt"—messy code produced without mentorship, compounded by insufficient code reviews. The code works. But it's a ticking time bomb.

The problem compounds. When new developers join a team with existing debt, they copy the patterns they see, perpetuating the cycle.

Beyond Technical Debt: The Rise of Non-Technical Debt

Research has identified five distinct types of non-technical debt that work alongside technical debt [6][2]:

Debt Type	Description	Impact
Process Debt	Inefficient or outdated workflows that hinder agility	Slows progress, misaligned processes
Social Debt	Suboptimal team dynamics and organizational culture	Poor communication, lack of trust, rework
People Debt	Issues with human resources and expertise	Inadequate training, hiring delays, overworked teams
Organizational Debt	Outdated structures, policies, or practices	Limits innovation, hinders adaptability
Culture Debt	Misaligned hires and weak organizational connections	Erodes team cohesion and effectiveness

As one leader noted: *"In today's context, technical debt is no longer purely a code issue. It's deeply intertwined with how teams are structured, how decisions are made, and how people are developed."*

The AI Factor: A New Kind of Debt

AI-assisted development is creating a new form of technical debt that organizations are only beginning to understand.

Research has identified "comprehension debt"—a novel form of technical debt where AI helps teams build systems more sophisticated than their independent skill level can create or maintain [7]. This paradox—possessing functional systems the team incompletely understands—creates fragility and AI dependency, distinct from traditional code quality debt.

A study based on 621 reflective diaries from 207 students identified four comprehension debt accumulation patterns [8]:

1. **AI-as-black-box code acceptance** – Accepting AI-generated code without understanding

2. **Context-mismatch debt** – Applying AI solutions out of context
3. **Dependency-induced atrophy** – Losing the ability to code independently
4. **Verification-bypass** – Skipping validation of AI output

Each of these patterns represents a failure of capability, not code quality. Comprehension debt resides in the collective cognition of development teams rather than in the codebase itself [8].

What Talent Management Can Do

The solution isn't to stop hiring junior developers. It's to develop them systematically.

Build skills alongside systems. Tech debt compounds when teams lack the capabilities to work with current tools and architectures. Investing in continuous skills development reduces the rate at which new debt accumulates.

Implement structured code quality programs. Training that covers code maturity, readability, manageability, and flexibility parameters. When teams understand what quality looks like, they're less likely to create debt.

Create mentorship structures. Junior developers need experienced guidance. Without it, they create unintentional debt that senior developers must clean up.

Teach AI collaboration skills. In an era where AI generates significant code, developers need the skills to evaluate, debug, and improve AI output. Without these skills, AI becomes a debt accelerator, not a productivity tool.

Address non-technical debt systematically. Implement ground rules to improve quality through clear protocols and guidelines [2]. Foster psychological safety and proactive leadership, which are essential for innovation and retention [1].

The Real Bottom Line

Technical debt isn't a code problem. It's a capability problem.

When you hire a developer who creates code debt, you're not just hiring a person. You're hiring for a future cleanup project that will consume thousands of engineering hours.

The developers who are thriving right now aren't the ones who write code fastest. They're the ones who write maintainable, scalable, secure code from the start—or who can evaluate and improve code, whether human-written or AI-generated.

Fix the people, and the code fixes itself.

References

1. https://dl.acm.org/doi/epdf/10.1145/3727967.3756846
2. https://ieeexplore.ieee.org/document/10803324
3. https://www.pega.com/about/news/press-releases/average-global-enterprise-wastes-more-370-million-every-year-through
4. https://www.barchart.com/story/news/35432513/average-global-enterprise-wastes-more-than-370-million-every-year-through-technical-debt-says-research
5. https://ieeexplore.ieee.org/document/11095915/authors#authors
6. https://www.e-informatyka.pl/attach/e-Informatica_-_Volume_18/eInformatica2024Art01.pdf#6#1
7. https://browse-export.arxiv.org/abs/2512.08942?context=cs
8. https://export.arxiv.org/abs/2604.13277