International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2019): 7.583

Preserving Architectural Integrity: Addressing the Erosion of Software Design

Raghavendra Sridhar

Independent Researcher
Email: *princeraj01[at]gmail.com*

Abstract: ***The escalating costs, complexity, and risks of new software development have driven organizations to extend the lifespan of existing systems across multiple decades. This extended utilization necessitates prolonged maintenance periods interspersed with intensive upgrade phases, contributing to the software's continuous evolution. While initial architectural frameworks are established during design, the software undergoes numerous modifications over time, both deliberate and inadvertent, resulting in architecture erosion where implementations diverge from original design intentions. These architectural deviations manifest as various technical issues ranging from minor performance inefficiencies and maintenance challenges to critical quality defects that can render systems completely inoperable or prohibitively difficult to maintain. This paper investigates the fundamental causes and consequences of software architecture erosion, evaluates remediation approaches, and establishes foundational work toward an Architectural Maturity Model Integration framework for assessing organizational capabilities in architectural governance and preservation.***

Keywords: System Architecture, Architectural Degradation, System Upkeep, Code Quality, Structural Decay in Software, Inherited Codebases, Architecture Maturity Framework

1. Introduction

In today's world, software projects are growing more complex than ever before. Think about modern cars, which now rely on up to 100 million lines of code, or the massive systems powering companies like Google, where the codebase is estimated at a staggering 2 billion lines [1]. With this complexity comes higher costs, longer timelines, and bigger teams-not to mention an increased risk of running over budget, missing deadlines, or even failing to deliver what customers really want. Because of these challenges, organizations often choose to keep their existing software running for as long as possible, investing in regular maintenance and upgrades instead of starting from scratch. This means some software systems are expected to last for decades, evolving to meet new technology standards, user expectations, and security needs along the way.

But as software ages, it faces a hidden challenge: architectural erosion. This happens when the way the software is actually built starts to drift away from its original design. The architecture of a system is like its blueprint, created early in the project to organize all the moving parts and ensure everything works together smoothly. A strong architecture makes complex projects more manageable by breaking them into focused areas and keeping different concerns separate [2]. Whether it's formally documented or just understood by the team, the architecture is a crucial decision point that shapes the entire project, ideally reflecting what customers need and the environment in which the software will operate.

Over time, though, it's easy for the actual implementation to stray from that original blueprint-sometimes without anyone noticing until problems start to pile up. This kind of erosion can make software harder to maintain, slow down development, hurt performance, and lower overall quality. In the worst cases, the software becomes so fragile that it can't be fixed or improved without a major overhaul [3]. Often, teams only address these issues after bugs or failures have already caused trouble, rather than preventing them in the first place. That's why many experts recommend proactive, process-driven approaches to help spot and fix architectural erosion before it becomes a serious problem.

2. Software Architecture Erosion: Challenging Common Assumptions

Traditional views of software architecture erosion often assume it occurs gradually and unintentionally during maintenance phases. However, erosion can begin as early as architectural selection or design stages, accelerate during intensive upgrade sprints, and even be deliberately incurred as technical debt to meet deadlines. While incremental changes over time contribute to degradation, rapid erosion emerges during periods of concentrated modifications (e.g., platform shifts or UI overhauls) under tight production constraints.

Contrary to the perception of erosion as purely accidental, teams may intentionally deviate from architectures to address perceived flaws or defer fixes, creating debt that compounds if unresolved. Such decisions, often driven by schedule pressures or evolving requirements, highlight the need for proactive architectural governance [4]. Erosion's multifaceted causes-spanning intentional trade-offs, rushed implementations, and uncoordinated changes-underscore the importance of continuous monitoring and structured evolution plans to mitigate systemic risks.

2.1 The Slow Creep of Software Problems: When Good Code Goes Bad

Think of software erosion like a tiny crack in a foundation – you might not notice it at first, but over time, it spreads and starts causing all sorts of headaches. In the beginning, it might just be a few weird glitches or a button that doesn't always work, annoying but manageable. But as a software project gets older, these little issues can snowball into a full-blown

crisis. Suddenly, your team is spending more time fixing bugs than building new things, customers are complaining, and what used to be a reliable system feels like it's constantly on the verge of breaking [5]. It's not just about slow performance; it's about the whole experience becoming a struggle for everyone involved – the people using it, the people trying to keep it running, and the people paying the bills. It's that sinking feeling when a simple update turns into a week-long ordeal, or when the software just can't do what it's supposed to anymore, no matter how much effort you pour into it.

Here's how that slow decay really starts to bite:

Things Just Don't Work Right Anymore: Imagine trying to use a feature that only works half the time, or when every attempt to fix one bug mysteriously creates two more. It's frustrating for users who can't get their work done and for developers who feel like they're constantly fighting fires.

Like Wading Through Treacle (Performance Slumps): Even if the software *technically* still does its job, it can become painfully slow and clunky. And ironically, sometimes the very attempts to speed things up can make the underlying problems even worse.

"Quick Updates" Become Ancient History: Simple maintenance tasks or adding a new feature starts to take forever. Your team gets bogged down, deadlines slip, and getting anything new out to your customers feels like a marathon.

The Money Pit Opens Up: It's not just about developers' salaries; it's the missed deadlines, the unhappy customers who might leave, the damage to your reputation when your product is seen as unreliable. The costs just keep piling up.

It's Just a Mess to Work With (Quality Plummets): The code becomes a tangled web. Good programming practices go out the window, making it harder to change anything, harder to test, and harder to roll out updates without breaking something else.

Walking on Eggshells (Brittle Code): The software becomes so fragile that even the smallest tweak can cause the whole thing to crash or lose important features. It's a stressful, high-stakes situation where everyone is afraid to touch anything.

2.2 Why Good Software Plans Can Go Astray: The Human Side of Code Decay

Think of building a house. You have a blueprint (the software architecture), but along the way, things can start to drift from that original plan. This "software erosion" isn't always some mysterious technical gremlin [6]; often, it's rooted in very human and team-related challenges. These issues can pop up right from the design stage or later during the nitty-gritty of coding and implementation. On the other side of the coin are the organizational and staffing headaches, which we'll touch on a bit later (in section 2.3).

So, what are some common ways the technical blueprint itself starts to crumble?

When Small Decisions Clash with the Big Picture: Sometimes, individual design choices made during development, especially when teams are moving fast (like in agile projects) or tackling unexpected problems, can unknowingly chip away at the overall architectural plan.

Lost in Translation: If the original architectural vision isn't crystal clear to everyone building the software, it's almost guaranteed that the final product won't quite match. Fuzzy documentation, team members coming and going, or constantly changing requests can all lead to misinterpretations and deviations.

Breaking the Architectural "Rules of the Road": Good architectures have guidelines on how different parts should talk to each other. Violations happen when, for instance, developers take a shortcut and make parts of the software communicate directly when they're supposed to go through specific channels, a bit like ignoring "no entry" signs in a well-planned city.

Digital Hoarding (Orphan Elements): Imagine a workshop cluttered with old tools and parts that no one uses anymore but are still taking up space. In software, these are "orphan elements"-bits of code left in the system that serve no purpose, adding confusion and unnecessary complexity.

Seeing Double (Duplicate Code): Sometimes, developers create multiple, nearly identical pieces of code to do the same job ("clone elements"). This becomes a headache because if you need to make a change, you have to find and update every single copy, or things will get out of sync.

Mixing Old and New (Legacy Integration): Bringing in older software or reusable components can seem like a time-saver. But often, it's like trying to fit puzzle pieces from different sets together – compromises are made, and these awkward connections can become weak points where erosion starts.

Things Get Too Tangled (Increased Coupling): As erosion sets in, different parts of the software that should be independent can become overly reliant on each other. This makes the system more complex, harder for anyone to understand fully, and a nightmare to test because a change in one place can cause unexpected problems elsewhere.

Losing Focus (Decreased Cohesion): The flip side of tangling is when individual components lose their clear, singular purpose. They might end up as a jumble of unrelated functions, often because teams try to avoid creating new pieces and just stuff more into existing ones. This makes the architecture muddled.

Overly Complicated Family Trees (Deep Inheritance): In programming, creating "families" of code (inheritance) can be useful. But if these family trees get too deep and complex, it's like having a very fragile ancestor – changing something at the top can have widespread and problematic consequences for all its "descendants."

Just Plain Too Complex: As the architecture grows more intricate, it simply becomes harder for anyone to keep the whole picture in their head. This confusion naturally leads to more mistakes and deviations from the intended design.

Wrong Blueprint for the Job (Incorrect Architecture): Sometimes, the problem starts right at the beginning: the chosen architecture might just be a poor fit for what the software actually needs to do. Later attempts to patch and work around these fundamental flaws, if not managed very carefully, can often make the erosion far worse than just sticking with the original, albeit flawed, plan.

2.3 The Human Element: Non-Technical Drivers of Software Erosion

It's not always about complicated code or tricky technical details when software starts to lose its way. Often, the real culprits are found in how a company works, the pressures people are under, and the unspoken rules of the workplace. Think of it like this: even the best blueprint for a building won't help if the construction crew is rushed, a new foreman takes over every week with different ideas, or there's no clear plan everyone is following. The same thing happens with software. When the environment and the way people are expected to work don't actively protect the software's design, even the most brilliant architecture will start to crumble over time [8]. It's the company's culture and day-to-day habits that can either build a strong defense against this decay or accidentally make it happen faster.

Here are some common ways the human and organizational side of things can lead to software headaches:

"Just Get It Done!" (The Deadline Crunch): When everyone's scrambling to hit a tight deadline, taking the time to do things "the right way" architecturally often gets pushed aside. It's like slapping on a quick fix to get the car running, knowing it might cause bigger problems down the road.

The Revolving Door (High Staff Turnover): If team members are constantly leaving and new folks are coming in, it's tough for anyone to really understand the software's original design, the "why" behind certain decisions, or even just how things are supposed to be done. New people, trying their best, can unintentionally make changes that weaken the architecture.

When the Process Doesn't Care About the Blueprint (Process Issues): Sometimes, the way a company builds software, like some super-fast agile methods, might focus so much on churning out small pieces quickly that no one's really looking at the big picture or ensuring all those pieces fit together neatly according to an overall design. The "architecture" just kind of happens, and it's often messy.

The Wild West (No Clear Process): If there's no set way of doing things and everyone is just figuring it out as they go, it's almost guaranteed that there won't be a well-thought-out architecture. Development and fixes become a free-for-all, which is a recipe for erosion.

It's Just "How We Do Things Around Here" (Company Culture): Some companies really value quality and have strong processes and best practices that help prevent or fix architectural problems, especially if they're building software where mistakes can have serious consequences. Others might prioritize speed or cutting costs above all else, and in those places, erosion is much more likely to take hold.

3. Software Erosion Management: Strategies, Methods, and Costs

Managing software architecture erosion involves three main approaches: prevention, minimization, and recovery. While prevention aims to stop erosion before it starts, it is often considered a form of minimization since it reduces the risk and impact of future erosion. In practice, most organizations focus on minimization and recovery, as completely preventing erosion is rarely feasible in complex, evolving systems. There is also the “no-action” approach, where erosion is allowed to occur without intervention [9]. This is typically reserved for small, non-critical projects with short lifecycles or limited business value, where the cost and effort of addressing erosion would outweigh any potential benefits. In such cases, organizations or customers may accept reduced quality or performance, knowing that the consequences are minimal.

Recovery approaches are designed to repair the damage caused by erosion after it has occurred. This typically involves identifying degraded components or modules and refactoring them to restore alignment with the intended architecture. While recovery can be effective, it often requires significant effort to locate and address all instances of erosion, and the process can become increasingly complex as the system grows. Process-driven minimization, by contrast, focuses on reducing the likelihood and severity of erosion through disciplined development practices. These methods rely on well-defined processes, such as maintaining comprehensive architectural documentation, enforcing design standards, and regularly monitoring compliance with architectural constraints. The goal is to catch and correct deviations early, making maintenance and updates more predictable and less costly over time. However, these approaches demand a high level of organizational discipline, ongoing communication among team members, and a willingness to invest in process rigor and oversight.

At the highest level of rigor and cost are formal minimization methods, which utilize advanced tools like Architecture Description Languages (ADLs), domain-specific constraint languages, and formal frameworks or patterns [10]. These techniques enable detailed syntactic analysis of architecture descriptions, helping teams identify rule violations and suggest targeted repairs. While formal methods can be highly effective in maintaining architectural integrity, they require extensive training, specialized expertise, and significant time investment, making them less common in everyday practice. Regardless of the chosen approach, successful erosion management depends on a foundation of thorough architecture analysis, clear documentation, and strong coordination across development teams. As software systems grow in complexity and importance, organizations must be prepared to escalate their investment in governance,

monitoring, and process improvement to ensure that erosion is kept in check and long-term software quality is preserved.

3.1 The Hunt for Software Decay: What's Being Done About It

When software starts to drift away from its original design, it's like a house slowly developing structural problems. Researchers have been working on ways to spot these issues early, prevent them from happening, and fix them when they do occur. All these approaches share a common goal: catching those moments when someone changes the code in a way that breaks good design rules, because those small breaks add up to big problems over time.

Seeing the Problem: Making Erosion Visible
One popular approach is to create visual tools that show developers where things are going wrong. Imagine being able to see a map of your software where trouble spots light up in red. These tools look for specific warning signs like circular dependencies (where module A needs module B, which needs module C, which needs module A-creating a tangled mess). Other tools track when object-oriented design principles are being violated, like when code that should be independent starts becoming too intertwined. By making these problems visible, developers can spot trends and fix issues before they spread throughout the system.

Building Rules into Models: The Blueprint Approach
Another strategy is to create detailed models of how the software should be structured, with built-in rules that act like architectural guidelines. Think of it as having a blueprint with notes that say "never connect the bathroom plumbing directly to the electrical system." These models can automatically check if new code follows the rules, flagging violations before they make it into the final product. This approach is especially helpful for teams that need to maintain strict standards.

Mapping Dependencies: Who Needs Whom
Some teams use tools that track which parts of the software depend on other parts. These create relationship maps (sometimes called Dependency Structure Matrices) that show at a glance if modules are connecting in ways they shouldn't. Other tools use specialized query languages that can ask questions like "show me all places where low-level code is directly accessing the user interface"-which would typically be a design no-no. While powerful, these approaches sometimes miss the bigger architectural picture.

Reverse Engineering: Working Backward
For more complex or older systems, some teams use sophisticated tools that work backward from the existing code to create models of what's actually happening, then compare that to what was intended. This is like examining a finished building and creating blueprints from what you see, then comparing those to the original plans to spot differences. These methods take more effort but can be invaluable for understanding systems that have evolved over many years.

Finding What Works for Your Team
There's no one-size-fits-all solution here. The right approach depends on what you're building, how experienced your team is, and how mature your organization is when it comes to thinking about architecture. A small startup might benefit most from simple visualization tools, while a company building medical devices might need the rigor of formal models with strict rules. As we'll see later, your organization's "architectural maturity" plays a big role in determining which methods will work best for keeping your software healthy over the long haul.

3.2 Gauging an Organization's Architectural Strength: The Maturity Factor

The way a company approaches the development and ongoing care of its software architecture reveals what we call its "Architectural Maturity Level." Businesses that are "architecturally-aware" and truly "architecture-centric" are naturally better equipped to handle the wear and tear (erosion and drift) that software designs can experience over time. However, to be formally recognized as having this advanced architectural focus, an organization needs to embrace several key principles. The more of these principles they successfully implement, the higher their architectural maturity. These principles aren't just abstract ideas; they form the practical steps a company must establish, enforce, and support to effectively control erosion and drift, especially in software systems where reliability is critical [11].

To achieve a high level of architectural maturity and effectively manage erosion, organizations should strive to meet the following foundational requirements:

- **Cultivating Architectural Awareness:** The entire organization must recognize the critical role of software architecture. This means having a clear, well-defined process for designing and modifying architecture that everyone understands and is expected to follow.
- **Explicit and Traceable Architectural Requirements:** Architectural needs must be clearly spelled out, documented, and directly linked to the software's quality goals and major functions. The company's process should detail how these requirements are recorded and maintained, including specific formats and any necessary systems or models.
- **Rigorous Conformance Checking:** All architectural designs must be verified for compliance before any coding begins and continually throughout the software's evolution. The steps for this checking, and what comes out of each step, must be clearly laid out in the organization's development process.
- **Architecture-First Updates:** Before any changes are made to the code to accommodate new requirements, the architecture itself must be updated first to reflect these planned changes. This ensures that implementation always follows a deliberate architectural adjustment.
- **Ensuring Implementation Compliance:** Every piece of new code or any update must be thoroughly checked to ensure it aligns with the established architectural design before it's released. The specific steps, tools, and procedures for these checks must be an integral part of the organization's adopted development process.

An organization that embeds strong support for these five principles within its defined software development process is demonstrably more architecturally mature than one that falls short in one or more areas. Adopting these practices directly

and positively impacts the challenges of architecture erosion and drift. It ensures that the causes, impacts, and management strategies discussed earlier are systematically taken into account through specific process steps and the documented outputs these steps produce.

4. Conclusions: Navigating the Legacy Software Landscape

As software development costs, timeline pressures, and complexity continue to rise, organizations increasingly opt to extend existing systems rather than build new ones from scratch. This economic reality has led to software lifecycles spanning decades, characterized by extended maintenance periods punctuated by intensive upgrade phases. Legacy systems require continuous adaptation to address technological shifts, emerging user needs, interface modernization, deployment changes, and security vulnerabilities. However, this ongoing evolution introduces a significant risk: architectural erosion.

This erosion-the gradual deviation from intended design-accumulates throughout a system's lifespan, progressively undermining performance, maintainability, reliability, and overall quality. The consequences range in severity from minor operational inefficiencies to catastrophic system failures where software becomes unstable, unusable, or unable to fulfill its core functions [12]. Similarly, maintenance impacts escalate from simple refactoring requirements to completely brittle, unmaintainable code bases that resist modification.

While conventional wisdom suggests erosion occurs gradually during maintenance phases, research reveals it can begin during initial architecture definition and accelerate rapidly under certain conditions. Contributing factors include schedule pressures, architectural misalignment with requirements, inadequate development processes, high staff turnover, and organizational culture that prioritizes short-term delivery over long-term sustainability. Organizations seeking to mitigate these risks must implement architecture-centric governance frameworks that address prevention, correction, and ongoing management of erosion. This requires establishing clear processes for architectural definition, maintenance, and evolution that explicitly account for the causes and impacts outlined in this research, supported by organizational commitment to architectural integrity throughout the software lifecycle.

5. Advancing Architectural Excellence: The Case for a Dedicated Maturity Model

The capability of organizations to effectively manage software architecture design, let alone address erosion and drift, varies dramatically. For critical software projects, particularly those involving life-or-death stakes, an organization's architectural maturity can be the paramount factor in selecting a vendor from competing bids [14]. It is widely accepted that when choosing a contractor for a large, critical project, decision-makers would favor one demonstrating a high level of architectural competence. Consequently, a standardized method for assessing an organization's architectural maturity is essential. Such a model would allow organizations to be formally recognized at specific competence levels, akin to the well-established CMMI-DEV model, which evaluates organizations based on the maturity of their overall product development processes and methods.

What is needed is a multi-dimensional framework that captures an organization's architectural proficiency through an evaluation of its specific architectural processes and the steps within those processes. An "Architecture Maturity Model Integration (AMMI)" built on this foundation could provide a robust means to assess an organization's architectural maturity, where higher maturity levels would ideally correlate with the production of more reliable, higher-quality software products. While software architecture is touched upon within the CMMI framework, it is not addressed with the specificity or depth required for an effective AMMI-style assessment [13]. Therefore, future research must focus on defining a dedicated architectural maturity model. This model should not only provide a means to assess organizations but also offer process guidance to help them enhance their architectural capabilities. Indeed, the foundational work presented in this paper begins to lay the groundwork necessary for developing such a comprehensive model.

References

[1] M. Fowler. "Technical Debt Quadrant". Internet: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html, October 14, 2009 [Dec. 28, 2019].
[2] B. Algaze. "Software is Increasingly Complex. That Can Be Dangerous." Internet: https://www.extremetech.com/computing/259977-software-increasingly-complex-thats-dangerous, Dec. 7, 2017 [Dec. 20, 2019].
[3] S.A. White. "Software Architecture Design Domain." In Proc. of the Second World Conference on Integrated Design and Process Technology, Vol. 1, Austin, TX. Dec. 1-4, 1996, pp. 283-290.
[4] S. A. White. "The Repository Based Software Engineering Program". in Proc. of the 1996 workshop, A NASA Focus on Software Reuse. George Mason University, Fairfax, Virginia pp. 53-62, September 24-27, 1996.
[5] H. Koziolek, D. Domis, T. Goldschmidt and P. Vorst. "Measuring Architecture Sustainability," IEEE Software, vol. 30, no. 6, pp. 54-62, Nov.-Dec. 2013.
[6] D. L. Parnas. "Designing software for ease of extension and contraction," IEEE transactions on software engineering, vol. SE-5, no. 2, pp. 128-138, Mar. 1979.
[7] B. Algaze. "Software is Increasingly Complex. That Can Be Dangerous." Internet: https://www.extremetech.com/computing/259977-software-increasingly-complex-thats-dangerous, Dec. 7, 2017 [Dec. 20, 2019].
[8] M. Dalgarno. (2009, Spring). "When Good Architecture Goes Bad," Methods and Tools [On line] Available http://www.methodsandtools.com/archive/archive.php?id=85, [Dec. 28, 2019].
[9] E. Whiting and S. Andrews. "Drift and Erosion in Software Architecture: Summary and Prevention

Strategies," to appear in Proceedings ACM 4th International Conference on Information System and Data Mining (ICISDM). Hilo, Hawaii, May 15-27. 2020.

[10] S.A. White. "A Framework for the development of Domain Specific Design Support Systems". in Proc. First World Conference on Integrated Design & Process Technology, Austin, TX. IDPT- Vol 1, Dec. 6-9, 1995, pp. 37-42.

[11] S. Schröder and M. Riebisch. "Architecture Conformance Checking with Description Logics," ECSA '17: in Proc. 11th European Conference on Software Architecture. Sep. 11–15, pp. 166-172, 2017.

[12] M. De Silva and I. Perera. "Preventing Software Architecture Erosion Through Static Architecture Conformance Checking," in Proc. IEEE 10th International Conference on Industrial and Information Systems (ICIIS0), Peradeniya, 2015, pp. 43-48.

[13] G. Murphy, K. Sullivan, D. Notkin. "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," ACM Software Engineering Notes. vol 20, issue 4, pp. 18-28, Oct. 1995.

[14] Z. Naboulski. "Code Metrics – Depth of Inheritance (DIT)," Internet: https://blogs.msdn.microsoft.com/zainnab/2011/05/19/code-metrics-depth-of-inheritance-dit/, May, 19, 2011 [Dec 20 2019].

[15] R. Shatnaw. "A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems," IEEE Transactions on Software Engineering. Vol: 36, Issue: 2 pp. 216–225, 2010.

[16] L. De Silva, & D. Balasubramaniam. "Controlling software architecture erosion: A survey," Journal of Systems and Software. Vol 85, Issue 1, pp. 132-151. Jan. 2012.

[17] R. Terra, M. T. Valente, K. Czarnecki and R. S. Bigonha, "Recommending Refactorings to Reverse Software Architecture Erosion," in Proc. 16th European