Gall's Law

Idea In Short

Do not begin with the full complexity you eventually hope to manage. Gall's Law states that a complex system that works is invariably found to have evolved from a simple system that worked. The implication is not aesthetic minimalism. It is an operational rule about learning, failure and adaptation. Systems become dependable when they accumulate validated components, tested interfaces and real feedback over time. Teams that attempt the finished complexity from the start usually inherit too many untested interactions at once. That makes failure harder to diagnose and harder to repair. Leaders should therefore treat working simplicity as the starting condition for reliable scale. Build something narrow that actually functions, expose it to reality, then add complexity in increments that preserve a working state

Gall's Law is one of the sharpest rules for building anything with interacting parts. It explains why so many ambitious systems look coherent in design documents and then fail in reality. The failure usually does not come from a lack of intelligence or effort. It comes from trying to assemble too many uncertain interactions at once. A working complex system is rarely born whole. It is usually grown.

What Gall's Law says

Gall's Law is commonly stated in a stark form: a complex system that works is invariably found to have evolved from a simple system that worked. John Gall presented the idea in his 1975 book on how systems behave and fail1. The fuller version is even more forceful. It continues by saying that a complex system designed from scratch never works and cannot be patched up to make it work, so teams must start over from a working simple system2.

The statement is memorable because it treats complexity not as a design target but as an outcome of evolution. In other words, dependable complexity is assembled through successive working states. A system earns its complexity by surviving contact with reality.

That is why Gall's Law has remained influential far beyond the book in which it appeared. It captures a pattern that engineers, managers, product teams and operators repeatedly encounter.

Why complexity fails from scratch

Complex systems fail from scratch because they contain too many unknown interactions to validate all at once. Every added component creates more interfaces, dependencies and paths to failure. When a team designs the whole structure in one pass, it often assumes that each part will behave as intended and that the parts will behave well together. Those assumptions are usually too optimistic.

The difficulty is not merely implementation. It is diagnosis. When a simple system fails, the cause is often easier to isolate. When a complex system fails, the cause may sit in an interaction between parts that each look correct in isolation. That makes debugging slow, confidence weak and repair expensive.

This is why Gall's Law is less about ideology than about testability. Systems become reliable when changes occur in units small enough to observe, verify and correct.

Why simple working systems matter

A simple working system does more than prove feasibility. It creates a stable base for learning. Once a narrow system functions in real conditions, teams gain information about user behavior, failure modes, throughput, constraints and operating assumptions. That evidence is far more valuable than confidence built entirely from planning.

A working simple system also gives teams something to preserve. New features, rules, or components can be added one layer at a time while maintaining a baseline that still works. If a change fails, the team knows what changed and what it changed from. This makes iteration legible.

That is the hidden strength of simplicity in Gall's Law. Simplicity is not praised because it is elegant. It is praised because it makes progress diagnosable.

Why the law matters in software and operations

Software teams understand Gall's Law intuitively because they have seen the difference between incremental change and big-bang rewrites. A narrow product that works can expand through tested modules, monitored releases and rollback paths. A fully specified system built all at once usually contains too many assumptions to validate before launch. The resulting failures are not just technical. They are organizational.

The same logic applies in operations. A process that works at small scale can be extended, automated and standardized over time. A process designed for full-scale complexity on day one may look complete on paper but collapse under real variability. The lesson is consistent: capability should be extended from demonstrated competence, not merely from intended architecture3.

That is why Gall's Law is useful in product design, platform engineering and operating model design. It rewards staged reliability over visionary completeness.

Why leaders ignore it

Leaders often ignore Gall's Law because comprehensive design feels efficient. It seems wasteful to begin with a limited system when the larger objective is already visible. Teams want to solve the whole problem, signal ambition and avoid revisiting architecture later. Large programs also create political pressure to define everything early, even when the system cannot yet support that certainty.

But this preference for complete design creates a trap. It shifts effort from validated learning to speculative coordination. Teams spend more time managing dependencies than proving they can operate reliably. The result is often a polished plan with little operational truth underneath it.

That is why Gall's Law is also a governance lesson. Overspecification can look like control while actually reducing the probability of a working result.

How to apply Gall's Law

The first step is to define the smallest version of the system that can deliver real value under real conditions. That version should be narrow enough to validate but substantial enough to reveal whether the underlying concept works. If it cannot function end to end, it is not yet

the right starting point.

The second step is to add complexity in increments that preserve a working state. Each addition should make the system more capable without making it opaque. If the team cannot explain what changed, how it is tested and what failure would look like, the increment is too large.

The third step is to let reality govern the pace of expansion. User behavior, defect patterns, operating cost and failure modes should shape what gets added next. Gall's Law works because it turns complexity into an earned outcome rather than an upfront declaration4.

The broader strategic lesson

Gall's Law matters because many failures that look technical are really failures of sequence. The system was not too ambitious in ultimate scope. It was too ambitious in initial form. It attempted to become complex before becoming dependable.

That lesson extends far beyond engineering. Organizations, workflows, transformation programs and policy systems all become fragile when they are designed as full-complexity solutions before they have a simple version that works. Leaders who respect Gall's Law do not reject scale. They respect the order in which scale becomes real.

A working complex system is therefore not just a larger design. It is the residue of many smaller validations. That is what makes it trustworthy.

- 1John Gall
- 2John Gall
- 3Gall's Law
- 4Complexity, By John Gall

Summary

Gall's Law endures because it explains why many ambitious systems fail despite intelligent design and good intentions. Working complexity is not simply designed correctly in one

pass. It is discovered through iteration, constraint and learning under real conditions. The law does not say simple systems always work, nor that complexity is undesirable. It says dependable complexity usually emerges from smaller working systems rather than from grand designs that try to solve every problem at once. That principle matters in software, organizations, products, policy and operations. When leaders respect it, they reduce hidden interdependence and keep failure legible. When they ignore it, they often create systems too complex to debug, too brittle to adapt and too incomplete to trust