

The Complexity Paradox

Why Engineers Admire Efficiency but Design Complexity

Alexey A. Nekludoff

AstraVerge Research

E-mail: an@astraverge.org
ORCID: 0009-0002-7724-5762

September 15, 2026

Abstract

Modern engineers often admire systems that achieve comparable results with fewer computational resources, lower memory requirements, and simpler infrastructure. At the same time, when designing their own systems, they frequently associate professional maturity with technological complexity: distributed databases instead of conventional relational systems, clusters instead of single servers, specialized storage instead of ordinary file or object storage, and complex orchestration platforms even where simpler solutions may be sufficient.

This creates a paradox. Efficiency is admired as a property of successful products, while minimization is often undervalued as an engineering method. This note argues that the contradiction is not primarily technical. It emerges from different incentive structures in technical education, engineering careers, large organizations, and the businesses that ultimately pay for technological systems.

Contents

1	Introduction	1
2	The Attraction of Complexity	1
3	The Three-Cent Business Requirement	2
4	The Chinese Model Paradox	3
5	When Complexity Becomes an Organizational Asset	3
6	Why Does the Business Allow Complexity?	4
7	A Concrete Example: The “AI Data Platform”	6
8	Simplicity Can Become a Fashion Too	8
9	Engineering as Reduction	8
10	Conclusion	9

1 Introduction

Software engineering has developed a substantial body of principles advocating simplicity: avoid unnecessary functionality, postpone architectural decisions until they are justified by requirements, and prefer systems whose complexity corresponds to the complexity of the problem being solved. The distinction between essential and accidental complexity has been recognized for decades [1], while evolutionary approaches to software design similarly discourage introducing distributed structures before they become necessary [2], [3].

Yet contemporary engineering practice reveals an apparent contradiction. Technological complexity is not merely tolerated; it is frequently associated with professional maturity. Distributed databases, orchestration platforms, specialized storage systems, streaming infrastructures, and large multi-component architectures often carry more professional prestige than simpler alternatives capable of satisfying the same requirements.

This creates what we call the *complexity paradox*: engineers admire efficiency as an outcome while often treating complexity as evidence of engineering sophistication.

The paradox becomes particularly visible in artificial intelligence. Models capable of achieving competitive results with fewer parameters, lower memory consumption, or reduced computational requirements are widely praised for their efficiency. At the infrastructure level, however, the same principle is not always applied. A simple database, object store, file server, or sufficiently powerful single machine may be regarded as architecturally unsophisticated even when it satisfies the actual workload.

This paper argues that the contradiction cannot be explained solely by technical considerations. Different actors optimize different objectives. A business benefits from obtaining the required result at lower cost. An engineer may benefit from experience with fashionable or sophisticated technologies. An organizational unit may benefit from larger budgets, larger teams, or strategically visible technology programs. Consequently, technical complexity can simultaneously represent a cost to the system and an asset to some of the actors involved in creating it.

The purpose of this paper is therefore not to argue that simple architectures are universally superior. Rather, it proposes a stricter criterion for introducing complexity: every significant increase in architectural complexity should correspond to an identifiable requirement that cannot be adequately satisfied by the simpler alternative.

A concrete machine-learning data architecture is used to illustrate this principle across three stages of scale. The example demonstrates that the same problem may rationally evolve from a conventional database and file or object storage, through a moderately distributed architecture, to a specialized distributed data platform—but that none of these later stages follows merely from the fact that the system is ‘large’, ‘modern’, or related to artificial intelligence.

2 The Attraction of Complexity

A recurring observation in technical education is that students are often more interested in learning sophisticated technologies than in understanding when those technologies are actually necessary.

A conventional relational database may appear uninteresting compared with a distributed analytical database. NFS may appear primitive compared with a specialized distributed storage system. A large server may seem architecturally unsophisticated compared with a Kubernetes cluster.

The implicit hierarchy often looks approximately like this:

more components → more specialized technologies → more complex architecture → more advanced engineering.

Yet software engineering has long distinguished complexity inherent in a problem from complexity introduced by its implementation [1]. The distinction matters because the latter requires justification: implementation complexity is not evidence of problem complexity.

The attraction of sophisticated technology is nevertheless understandable. Complex technologies have substantial career value. They appear in job descriptions, conference presentations, architecture diagrams, and technical interviews. Experience with Kubernetes, distributed databases, streaming platforms, and large-scale data infrastructure can be converted into professional status and market value.

This effect is not merely anecdotal. An empirical study of 591 software professionals found that 82% believed that using trending technologies made them more attractive to prospective employers, while 60% of hiring professionals reported that technology trends influenced their job offerings [4].

An engineer can therefore have a rational incentive to acquire technological complexity. A business, however, has a different incentive.

3 The Three-Cent Business Requirement

A business rarely needs Kubernetes, PostgreSQL, YTsaurus, Apache Doris, or any other particular technology. It needs a problem solved and, other things being equal, wants that problem solved as cheaply as possible.

The business requirement can therefore be expressed somewhat provocatively as:

required result → minimum acceptable cost.

This creates a fundamental conflict. The engineer may benefit professionally from a system containing ten technologies. The organization benefits economically if the same result can be obtained with three.

Suppose a machine-learning dataset must be stored and accessed by a number of training processes. It is tempting to begin by selecting a specialized distributed data platform. But the first engineering questions should be much less exciting:

Why is ordinary object storage insufficient?

Why is NFS insufficient?

If structured metadata must be queried and transformed, why is a conventional relational database insufficient?

These questions do not imply that simple technologies are always preferable. At sufficiently large scales, or under particular consistency, throughput, availability, geographical, or computational requirements, specialized distributed systems may be entirely justified.

The important question is different:

Which requirement makes the simpler solution insufficient?

If this question cannot be answered, technological complexity has not yet been justified. This is consistent with the YAGNI principle: anticipated capabilities impose not only an implementation cost but also continuing costs of maintenance, testing, understanding, and adaptation [2].

A similar argument has become influential in industry practice through McKinley’s “Choose Boring Technology” essay [5].

4 The Chinese Model Paradox

The contradiction becomes particularly visible in contemporary discussions of artificial intelligence.

Engineers frequently admire models that provide competitive capabilities while requiring less memory, less computation, or less expensive hardware. Recent Chinese AI systems have attracted considerable attention precisely because efficiency has become one of the important dimensions on which models are compared.

The admired direction is therefore:

fewer resources $\rightarrow$ lower cost $\rightarrow$ comparable result.

Yet the same engineers may approach ordinary infrastructure design from almost the opposite direction:

more components $\rightarrow$ more specialized infrastructure $\rightarrow$ more sophisticated architecture.

This is the complexity paradox. We admire efficiency when it appears in a finished product, but frequently fail to recognize minimization as an engineering method that produces efficiency.

An efficient system is not interesting merely because it eventually became small or fast. Someone had to ask repeatedly: What do we actually need? What can be removed? What can be replaced by something simpler? Which complexity is fundamental to the problem, and which complexity did we introduce ourselves?

Adding another component is comparatively easy: every new component can usually be associated with some desirable property. Removing a component is harder because it requires demonstrating that the required properties of the complete system remain intact without it. Reduction can therefore require deeper understanding than construction.

This observation matters because it rules out a trivial explanation of the complexity paradox. The preference for sophisticated infrastructure cannot simply be attributed to an inability to recognize the value of efficiency. The same engineers readily recognize reduction in computation, memory, and hardware requirements as an engineering achievement when evaluating systems designed by others. The asymmetry appears when the same criterion must be applied reflexively to one’s own architectural decisions. The question is therefore not whether engineers value efficiency, but why that value can become secondary when they themselves choose the architecture.

5 When Complexity Becomes an Organizational Asset

Technical architecture is not selected in an organizational vacuum. Organizations adopt management practices partly because those practices are perceived as progressive, rational, or legitimate, rather than solely because of demonstrated improvements in efficiency [6].

A related mechanism is mimetic isomorphism: under uncertainty, organizations tend to imitate other organizations perceived as successful or legitimate [7]. In technology, this creates an obvious path by which an architecture developed for the specific constraints of a large technology company can become a model for organizations operating under very different constraints.

Real software structures are therefore shaped not only by formal design principles, but also by organizational history, expediency, imitation, and incremental evolution [8].

A simple solution may solve a problem but create little organizational activity around itself. A larger architecture can create budgets, projects, teams, management positions, migration programs, consulting contracts, conference presentations, and valuable entries in professional CVs.

Thus the optimization function inside a large organization may differ from the optimization function of the business as a whole. The textbook model of engineering selection is:

requirements → alternatives → evaluation → technology.

The organizational process may instead resemble:

organizational incentives → budget → career opportunities → technically defensible architecture.

This phenomenon is neither specific to a particular country nor to a particular technology. The vocabulary changes: one decade produces service-oriented architectures, another microservices, cloud-native platforms, data meshes, zero-trust architectures, or AI-ready infrastructure. The underlying organizational incentive can remain remarkably stable.

Empirical research on IT fashion supports the broader point that technological adoption can provide organizational and managerial benefits that are not identical to improvements in operational performance. Association with fashionable information technologies may improve organizational reputation and executive compensation even without corresponding short-term performance improvements [9].

This also helps explain one of the most common arguments in engineering:

“This is how we do it here.”

A technology may have been entirely rational when introduced into a particular organization because of scale, existing infrastructure, internal expertise, or integration with other systems. Over time, however, the original constraints can disappear from the explanation:

solution to a local problem → organizational standard → best practice → assumed engineering principle.

Engineers then move between organizations and carry these assumed principles with them. A solution created for the infrastructure of a global technology company can eventually appear in a company whose entire workload would comfortably fit on one server.

6 Why Does the Business Allow Complexity?

This raises a more fundamental question. If the organization pays for the engineers, owns the infrastructure, and ultimately bears the cost of operating the resulting system, why does it allow technological complexity to grow at all?

Part of the answer appears before engineers enter any particular organization. Technologies such as Kubernetes, distributed databases, specialized data platforms, and large-scale infrastructure have value in the labor market. They appear in job descriptions, technical interviews, conference programs, and professional profiles. Empirical evidence on résumé-driven development confirms that developers perceive experience with trending technologies as valuable to prospective employers [4].

Technological complexity can therefore acquire value as *professional capital*, independently of whether that complexity is required by any particular system. For an engineer, learning and using a sophisticated technology may be rational even when a simpler technology would be sufficient for the immediate technical problem.

This explains why engineers may be attracted to complexity, but it does not yet explain why organizations accept and sometimes actively encourage it.

Complexity can also acquire *organizational value*. Fashionable technologies can signal technical maturity, facilitate recruitment, justify larger budgets and teams, strengthen organizational legitimacy, and associate a company with practices used by highly visible technology firms. Research on management fashion and institutional imitation provides a broader theoretical basis for this mechanism [6], [7], while empirical work on IT fashion demonstrates that adoption can produce reputational and managerial benefits independently of immediate improvements in operational performance [9].

An architectural decision may therefore be evaluated simultaneously against at least three different sources of value:

$$\text{technical value} \neq \text{professional value} \neq \text{organizational value}.$$

These values can coincide. A genuinely large system may require a distributed platform; engineers may gain valuable experience from building it; and the organization may benefit from the resulting capability.

But they need not coincide.

A technology may have high professional value while adding little technical value to a particular system. It may have organizational value as a signal of technological sophistication while increasing rather than reducing operational cost. Conversely, a technically mundane solution may satisfy the actual requirements extremely well while generating little professional or organizational visibility.

This distinction explains why the phenomenon is already visible in technical education. Students have not yet entered the organizations whose systems they will eventually design, yet the professional environment has already taught them which technologies constitute valuable career capital. Their preference is therefore rational from the perspective of their future position in the labor market: learning a widely demanded distributed platform may offer greater professional value than learning that a particular problem requires no distributed platform at all.

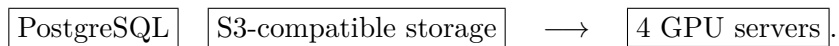
The resulting preference for complexity does not originate in the requirements of a particular system. It is formed earlier in the professional environment and can later be reinforced by organizational incentives. By the time an actual architectural decision is made, complexity may therefore carry technical, professional, and organizational value simultaneously—even when only the first of these is required to solve the engineering problem.

7 A Concrete Example: The “AI Data Platform”

Consider a company building a computer-vision product. It has 20 TB of training images, several million metadata records, four GPU servers, and a team that retrains its models a few times per week. New images arrive daily, but training datasets are effectively immutable once a particular experiment starts.

The simple architecture

A straightforward architecture might consist of S3-compatible object storage for images and model artifacts, PostgreSQL for metadata, labels, provenance, experiment information, and dataset manifests, and ordinary Python jobs for dataset preparation and training:

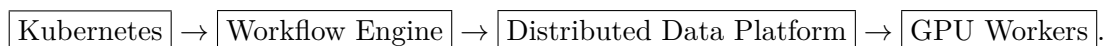


A training job receives a dataset manifest, reads the required objects, trains the model, and writes the resulting artifacts back to object storage. If the aggregate read bandwidth is sufficient and ordinary SQL is adequate for constructing and validating datasets, the architecture satisfies the stated requirements.

The attractive architecture

Now imagine an architecture review. The system is criticized for not being sufficiently scalable. A proposal appears for an “enterprise AI data platform” consisting of Kubernetes, a distributed data system, a message broker, a distributed analytical database, centralized observability, a workflow orchestrator, and an internal abstraction layer hiding the storage technologies from ML engineers.

The new diagram looks considerably more impressive:



Every component can be technically justified. Kubernetes provides orchestration. Distributed storage provides scalability. A message broker decouples producers and consumers. An analytical database accelerates analytical queries. A workflow system manages dependencies. None of these statements is false.

But they do not answer the relevant architectural question:

Which requirement of the original system could not be satisfied by PostgreSQL, object storage, and four GPU servers?

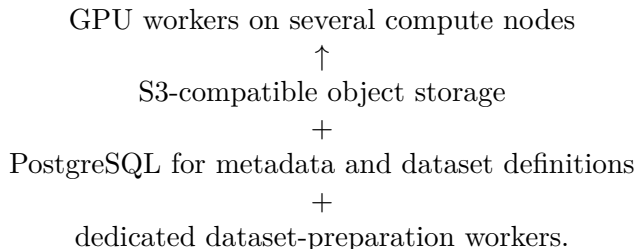
If the answer is “we may need to scale in the future,” the organization is paying today for a hypothetical future requirement. If the answer is “large technology companies use this architecture,” an organizational precedent has been substituted for a system requirement. If no measurable requirement can be identified, the second architecture is not more advanced. It is simply more complex.

The middle ground

Real systems need not jump directly from a simple architecture to a fully integrated platform. Suppose the dataset grows to 100–200 TB, several training teams now share the infrastructure,

dozens of GPU workers run concurrent experiments, and a single storage server becomes an undesirable bottleneck or failure domain.

A realistic intermediate architecture might be:



Large immutable objects are stored in object storage, while PostgreSQL continues to hold structured metadata, provenance, labels, quality information, dataset versions, and experiment references. Dataset-preparation workers perform expensive transformations and produce immutable manifests consumed by training jobs.

The architecture is distributed where distribution provides a concrete benefit, but not every component has become distributed. The data path remains straightforward:

$$\text{metadata} \rightarrow \text{dataset manifest} \rightarrow \text{objects} \rightarrow \text{training workers.}$$

This illustrates a broader property of complex systems: they can emerge through the composition and evolution of simpler subsystems rather than appearing fully formed [10]. In software architecture, Fowler makes a closely related practical argument for beginning with a simpler monolithic structure and introducing distribution when sufficient reasons emerge [3].

The evolution may therefore be incremental:

$$\text{NFS/S3} + \text{PostgreSQL} \rightarrow \text{object storage} + \text{PostgreSQL} + \text{workers} \rightarrow \text{integrated distributed data platform.}$$

Each transition should correspond to a measurable constraint: storage capacity, aggregate bandwidth, concurrency, failure isolation, dataset-preparation time, multi-tenancy, or another explicit requirement. The middle architecture may survive for years; it may also remain the final architecture.

When the complex architecture becomes necessary

Suppose the company eventually operates thousands of workers, continuously produces derived datasets, executes petabyte-scale distributed transformations, requires strict multi-tenant resource isolation, and must schedule computation close to distributed data. The simple and intermediate architectures may now genuinely become insufficient, and a system such as YTSaurus or another integrated distributed data-and-compute platform may be entirely reasonable.

But notice what changed. The technology did not become appropriate because the data were a machine-learning training dataset or because the company was “doing AI.” It became appropriate because the requirements changed:

$$\text{ML dataset} \not\Rightarrow \text{distributed data platform,}$$

while

specific scale and operational requirements $\Rightarrow$ possibly distributed architecture.

The local architectural consequence should not be mistaken for a general engineering principle.

8 Simplicity Can Become a Fashion Too

The argument above does not imply that simpler architectures are intrinsically superior. Simplicity itself can become a source of professional identity and status. “Boring technology,” monolithic architectures, minimal infrastructure, and rejection of fashionable platforms can become cultural signals just as easily as technological complexity can.

This produces a symmetric failure mode. An engineer may reject a distributed architecture not because it is unnecessary, but because minimalism itself has become the preferred engineering identity.

The distinction therefore cannot be made by counting components or classifying technologies as simple or complex. Both approaches must be evaluated against the same external criterion:

requirements $\rightarrow$ constraints $\rightarrow$ adequate architecture.

A twenty-component architecture is not wrong because it contains twenty components. A three-component architecture is not correct because it contains only three. The relevant question is whether each architectural decision can be traced to a requirement or constraint whose satisfaction justifies its cost.

Thus, reduction is a method rather than an objective. The goal is not the smallest possible system, but the smallest system that adequately satisfies the actual problem.

9 Engineering as Reduction

None of this is an argument against complex systems. Some problems are intrinsically large, distributed, concurrent, geographically separated, or subject to requirements that cannot be satisfied by a simple architecture. Complexity is sometimes unavoidable.

But unavoidable complexity and prestigious complexity are different things.

A useful engineering principle is therefore:

A more complex technology should be introduced when a concrete requirement cannot be adequately satisfied by the simpler existing solution.

This changes the role of architecture. The purpose of an architect is not to maximize the number of technologies in a diagram, nor is professional maturity demonstrated by knowing the largest possible number of fashionable systems.

Sometimes the strongest architectural decision is to look at thirty components and say:

“Keep five.”

This principle also exposes an important mismatch between technical education and industrial reality. Students reasonably seek technologies that increase their value in the labor market. Employers advertise complex technology stacks and reward experience with them. Educational programs respond by teaching those technologies. Yet the economic system that ultimately finances engineering asks a much simpler question:

Can you produce the required result for less?

Perhaps one of the most important skills technical education can provide is therefore not the ability to add another technology to an architecture. It is the ability to justify why that technology needs to be there at all.

10 Conclusion

The complexity paradox arises because architectural complexity has different values for different participants in technological development. For a business, complexity is primarily a cost that must eventually be paid through infrastructure, engineering effort, maintenance, and operational risk. For engineers and organizational units, however, the same complexity may provide professional, organizational, or strategic value.

This difference in incentives helps explain why the widely accepted principle of engineering simplicity does not necessarily produce simple systems. The problem is not that engineers are unaware of the cost of complexity. Rather, complexity itself may provide benefits that are external to the technical system being designed.

The practical implication is not that distributed databases, Kubernetes, specialized data platforms, or other sophisticated technologies should be avoided. At sufficient scale and under appropriate requirements, they may be exactly the right solution. The important distinction is between complexity that follows from the problem and complexity that follows from the environment in which the problem is being solved.

Engineering maturity should therefore not be measured by the number or sophistication of technologies employed. A stronger criterion is the ability to identify the minimum architecture that satisfies the actual requirements and to introduce additional complexity only when a specific constraint makes it necessary.

This leads to a simple question that can be applied to almost every architectural decision:

What requirement prevents us from using the simpler solution?

If this question has a precise answer, additional complexity may be justified. If it does not, the architecture may be optimizing something other than the technical problem.

In this sense, reduction is not the opposite of advanced engineering. It is one of its strongest forms.

References

- [1] F. P. Brooks, “No silver bullet: Essence and accidents of software engineering,” *Computer*, vol. 20, no. 4, pp. 10–19, 1987. DOI: 10.1109/MC.1987.1663532 [Online]. Available: <https://doi.org/10.1109/MC.1987.1663532>
- [2] M. Fowler. “Yagni,” Accessed: Sep. 15, 2026. [Online]. Available: <https://martinfowler.com/bliki/Yagni.html>
- [3] M. Fowler. “Monolith first,” Accessed: Sep. 15, 2026. [Online]. Available: <https://martinfowler.com/bliki/MonolithFirst.html>
- [4] J. Fritzsch, M. Wyrich, J. Bogner, and S. Wagner, “Résumé-driven development: A definition and empirical characterization,” 2021. arXiv: 2101.12703 [cs.SE]. Accessed: Sep. 15, 2026. [Online]. Available: <https://arxiv.org/abs/2101.12703>
- [5] D. McKinley. “Choose boring technology,” Accessed: Sep. 15, 2026. [Online]. Available: <https://mcfunley.com/choose-boring-technology>

- [6] E. Abrahamson, "Management fashion," *Academy of Management Review*, vol. 21, no. 1, pp. 254–285, 1996. DOI: 10.5465/amr.1996.9602161572 [Online]. Available: <https://doi.org/10.5465/amr.1996.9602161572>
- [7] P. J. DiMaggio and W. W. Powell, "The iron cage revisited: Institutional isomorphism and collective rationality in organizational fields," *American Sociological Review*, vol. 48, no. 2, pp. 147–160, 1983. DOI: 10.2307/2095101 [Online]. Available: <https://doi.org/10.2307/2095101>
- [8] B. Foote and J. W. Yoder, "Big ball of mud," in *Proceedings of the Fourth Conference on Pattern Languages of Programs (PLoP '97)*, Technical Report WUCS-97-34, Monticello, Illinois, 1997. [Online]. Available: <https://www.joeyoder.com/research/papers>
- [9] P. Wang, "Chasing the hottest it: Effects of information technology fashion on organizations," *MIS Quarterly*, vol. 34, no. 1, pp. 63–85, 2010. Accessed: Sep. 15, 2026. [Online]. Available: <https://aisel.aisnet.org/misq/vol34/iss1/6/>
- [10] H. A. Simon, "The architecture of complexity," *Proceedings of the American Philosophical Society*, vol. 106, no. 6, pp. 467–482, 1962. [Online]. Available: <https://www.jstor.org/stable/985254>