Why causal coding is faster than chaos

How invariants generate speed, creativity, and stability in software systems

Author
Alexander Wolfgang Strüver

Date
28.12.2025

Location
Thailand

License: CC BY-NC-ND 4.0

1. Introduction: The Misconception of Speed and Creativity

Current debates surrounding software development and AI-powered programming repeatedly reveal a fundamental misunderstanding: freedom is equated with speed, structure with restriction. The implicit assumption is that the fewer rules, boundaries, or fixed specifications a system has, the faster, more creatively, and more scalably it can develop.

However, this equation is incorrect.

She overlooks a crucial systemic connection: speed, creativity, and long-term scalability arise not from the absence of boundaries, but from clearly defined limits. Where no invariants exist, there is no orientation. Where no limits of impossibility are defined, the search space explodes. And where the search space is unlimited, effective speed decreases, even though activity increases.

The problem lies not in the coding style, personal preferences, or methodological schools, but in the underlying system logic. A system without fixed reference points can try many things, but cannot develop anything systematically. It produces possibilities, but not robust structures.

The central question, therefore, is not how much freedom a system should have, but rather why limitations are precisely the prerequisite for speed, creativity, and scalability to emerge in the first place. This question cannot be answered aesthetically or ideologically, but only functionally.

The following text explores this question from a technical perspective and shows why stable software and AI systems become powerful not despite, but because of clearly defined invariants, limits, and causal structures.

2. Chaos is not creativity – a technical clarification

To correctly answer the question of speed and creativity in technical systems, a precise definition of terms is necessary. In many debates, chaos and creativity are implicitly equated, even though they describe fundamentally different systemic states.

Chaos refers to an unfiltered space of possibilities. All states, transitions, and combinations are, in principle, permitted without any prior exclusion criteria. From a technical perspective, this means a maximal search space in which every new option is equally valid alongside all others.

Creativity, on the other hand, is not unlimited experimentation, but selective variation within stable boundaries. It presupposes that a large proportion of possible states are excluded from the outset. Only in this way is it possible to systematically evaluate, develop, or discard variations.

This difference can be described mathematically. In an unlimited search space, the number of possible states grows exponentially. With each additional variable, the combinatorics increase faster than they can be meaningfully searched or evaluated. The result is an increase in activity coupled with a decrease in effective speed. A lot is generated, but little of it is usable or applicable.

In a limited search space, however, exclusion criteria act as a reduction mechanism. They narrow down the number of possible states before any calculations, programming, or testing even takes place. This doesn't reduce the variety, but rather the arbitrariness. Relevant variants become visible more quickly, while irrelevant ones are immediately eliminated.

The crucial point, therefore, is not whether a system can generate many possibilities, but whether it is capable of eliminating possibilities early on. Without these exclusion criteria, complexity explodes. The perceived dynamism increases, but the actual speed of development decreases.

In this sense, chaos is not a creative state, but a computationally inefficient one. Creativity arises where variation is guided by stable boundaries – not where everything is equally possible.

3. Classical programming: Stable, but often causally underdefined

Classical programming remains the backbone of stable software systems. It is deterministic, follows clear control flows, and allows programs to be executed logically correctly and reproducibly. For many applications, this form of programming is sufficient and proven.

Its strength lies in the explicit control of processes. Control structures, state transitions, and data flows are clearly defined. Errors can be located, tests can be targeted, and the system's behavior is fundamentally explainable.

At the same time, however, a structural limitation becomes apparent: In many classic codebases, the actual invariants are not formalized. Assumptions about valid states,

permissible transitions, or system-wide boundary conditions often exist only implicitly – in the minds of the developers, in comments, or in historically developed conventions.

System boundaries are often not clearly defined. Which states should be impossible, which paths are explicitly excluded, and where hard terminations are necessary often remains imprecise. As a result, architectural discipline becomes a matter of individual experience and diligence, rather than an inherent property of the system itself.

This has a clear consequence: Classical programming is logically correct, but often not systemically closed. Stability arises from convention, reviews, and discipline, not from enforceable causal structures. As long as this discipline exists, such systems function reliably. If it is lost, or if the system grows beyond its original assumptions, the risk of drift, inconsistent extensions, and hard-to-locate errors increases.

This observation leads to the question of what programming must look like if stability is to be borne not by people, but by the architecture itself.

4. Probabilistic / Vibe Coding: Speed without orientation

The use of AI-powered tools has led to the establishment of a new form of programming, often referred to as probabilistic or vibe coding. This approach differs fundamentally from classical programming, not primarily in the tools used, but in the underlying logic.

Probabilistic coding works with patterns rather than causes. Decisions do not arise from explicit causal models, but from statistical probabilities and similarities to known structures. The development process is iterative: results are generated, evaluated, adapted, and generated again. Architecture is not created beforehand, but—if at all—subsequently from the emerging behavior.

A key feature of this approach is its strong context dependency. The quality and consistency of the results depend significantly on the available context. Changes in the prompt, the model, or the surrounding system state can lead to qualitatively similar but structurally different solutions.

In the short term, this approach generates a high perceived speed. Functional code is created quickly, prototypes can be built in a short time, and visible results convey an impression of high productivity.

In the long run, however, structural effects become apparent. Without fixed invariants, the meaning of code and architecture drifts across iterations. Inconsistencies accumulate, interfaces lose their clarity, and extensions increasingly require corrections to existing code. The system is not further developed, but continuously repaired.

A repair economy emerges: progress consists less of building stable structures than of constantly correcting deviations. The initial speed reverses. Changes become more expensive, errors harder to locate, and stability becomes an emergent by-product.

The central diagnosis is clear: The problem does not lie in the AI itself. It lies in the lack of explicit, enforceable invariants. Without fixed reference points, even the most powerful tool

cannot provide orientation. Speed without orientation remains short-term – and leads to systemic drift.

5. Causal Coding: The Closed Core

Causal coding addresses a point that is missing in many modern development approaches: the explicit construction of aclosed causal coreThis core forms the foundation of the entire system and differs fundamentally from modular or iterative architectures, which only seek their stability in the course of development.

A causal kernel is invariant. Its fundamental rules, state spaces, and transitions are clearly defined and do not change through subsequent extensions. It is deliberately kept small to ensure its complete penetration, verifiability, and stability. At the same time, it is closed: all permissible states and transitions are explicitly described; everything else is excluded. Finally, it is non-negotiable. Its rules are not adapted situationally to "enable" new requirements but serve as a fixed reference for all further development.

The underlying principle is clear: The causal core is constructed correctly once. This phase requires precision, discipline, and formal clarity. However, once the core is complete, it is no longer modified. Further development takes place exclusively in the layers above it, which must adhere to the invariants of the core.

This approach is not unique to software engineering, but rather corresponds to proven structures in other stable systems. In the cosmos, fundamental constants like gravity or the speed of light are not adjusted to allow for new phenomena. Precisely because they are invariant, diversity and complexity can unfold. The kernel of an operating system behaves similarly: it defines basic rules and interfaces and remains stable, while a multitude of complex applications can be built upon it.

The crucial factor here is the perspective: The closed causal core does not create limitations, but rather provides orientation. It does not reduce possibilities, but structures them. By defining what is impossible, it creates a clear framework for what can be meaningfully developed further. On this basis, stability is not imposed, but systemically guaranteed.

6. Invariants as velocity amplifiers

The key practical benefit of causal coding lies in the effect of invariants as structural speed enhancers. While invariants are often misunderstood as constraints, they fulfill precisely the opposite role in functional systems.

The underlying mechanism is simple: Every new code snippet, every extension, and every modification is immediately checked against the invariants of the causal kernel. This check doesn't occur during operation or in later testing phases, but rather as a preliminary filter within the development process itself. Anything that contradicts the invariants is discarded.

The consequence is clear. Entropic states – that is, variants that are not structurally compatible – are immediately discarded. This prevents an explosion of variants before it can even occur. The development process narrows early on to a relevant, consistent search space.

This reduced search space has direct technical effects. Debugging becomes faster because errors are no longer diffusely distributed across the system, but rather become visible as clear violations of defined invariants. Causes can be located more precisely, and corrections can be targeted more effectively. At the same time, the need for subsequent rework decreases significantly, as flawed developments are not carried over through multiple iterations.

The crucial factor here is timing. Speed doesn't come from quickly generating many options, but from eliminating irrelevant or contradictory states early on. Every discarded path not only saves development time, but also prevents subsequent costs in maintenance, debugging, and stabilization.

The core message can therefore be clearly formulated: Speed in system design is not achieved through late repairs, but through early elimination. Invariants do not act as brakes, but as structural accelerators, because they provide orientation and consistently focus the development space.

7. Entropy zones: Why boundaries are not mistakes

In stable technical systems, there are always areas that are deliberately left untouched. These areas are often misunderstood or interpreted as deficiencies. In fact, they represent a key stability mechanism.

Entropy zones denote states or transitions that the system intentionally excludes. They are not unintended edge cases, but explicitly defined impossibilities. Their purpose is not to prevent functionality, but to protect the system from states in which order, reproducibility, or safety would be lost.

Technically, such entropy zones manifest themselves in clear constructs: hard terminations, forbidden states, failsafes, or irreversible exit paths. These mechanisms ensure that the system does not attempt to "repair" or further process inconsistent or dangerous states. Instead, the execution path is consistently terminated or redirected.

A common misconception is to view these boundaries as flaws or limitations. From a functional perspective, they are quite the opposite. They define orientation. A system that knows no forbidden states also knows no safe direction. Only through clearly defined impossibilities does behavior become controllable and assessable.

The analogy to physical systems clarifies this point. Black holes are not a defect in the universe, but rather define gravitational boundaries. They mark regions that cannot be entered without destroying the system. It is precisely through this boundary that the overall structure remains stable. Orientation arises not from the fact that everything is accessible, but from the clarity of what must be avoided.

Applied to software and AI systems, this means: "No entry" is an active protective mechanism. Entropy zones prevent the system from entering states from which a controlled return is no longer possible. They reduce risks, limit the spread of damage, and create reliable safety assumptions.

This aspect is crucial for governance and safety. Systems can only be operated, regulated, or audited responsibly if it is clearly defined which states are excluded and why. Limits are not a weakness of a system, but rather a prerequisite for it to remain controllable.

The key message is therefore: Entropy zones are not an expression of scarcity, but of order. They create orientation, and it is precisely through their conscious avoidance that they protect the system as a whole.

8. Layered Complexity: Where Creativity Really Arises

A common category error in discussions about complexity, fractality, and system design is to locate complexity at the foundation. In fact, the exact opposite is true in stable systems:The deeper the level, the simpler and more invariant it is. Complexity arises layer by layer as you move upwards.

The fundamental principle is this: The core of a system is simple, invariant, and closed. It defines a few, strict rules and state spaces that are neither extended nor relativized. Each layer above it completely adopts these invariants and extends them with additional degrees of freedom. This creates new complexity without altering the foundation.

This principle is universally observable. In technical systems, the kernel forms the lowest level. It provides elementary mechanisms and deliberately remains minimal. Building upon this, services are created that bundle specific functions, and above these, in turn, user interfaces, interaction logics, and dynamic behavior are developed. The highest complexity always lies where the most degrees of freedom exist—not in the kernel.

The same pattern is evident in natural systems. The fundamental laws of physics are simple and invariant. Chemistry extends these rules by adding new combinations without altering them. Biology, in turn, builds upon chemical processes and generates highly complex, adaptive structures. None of these levels relativizes the one below it; each presupposes it.

In this context, fractality does not describe the absence of structure, but rather its consistent repetition across levels. Local self-similarity is not a sign of autonomy without order, but rather an expression of a global, stable causal structure. Fractal patterns arise because invariants remain effective at all levels—not because they are absent.

The central point is therefore clear:Complexity grows upwards, never at the base.Creativity does not arise from the resolution of invariants, but from their consistent adoption and extension at higher levels. Those who use fractality as an argument against a stable core are confusing form with cause.

Properly understood, fractality is not an alternative to causal design, but rather its natural effect. It demonstrates that stable invariants enable diversity – not prevent it.

9. Why causal coding is superior in the long run

The comparison between chaotic and causally structured development approaches cannot be reduced to matters of style or personal preference. It is a question of the long-term functionality of systems.

Chaotic approaches generate a lot of output in the short term. Code is created quickly, numerous variants are produced, and activity conveys progress. However, the actual value of this output remains limited. Without stable reference points, it is unclear which results are viable, which represent only random intermediate states, and which will incur long-term costs. A high output is no substitute for directed development.

In comparison, causal coding produces less immediate output. The search space is deliberately limited, and many possibilities are excluded from the outset. The crucial difference lies in the value of the remaining results. Every generated building block is structurally embedded, connectable, and verifiable. Development is slower in breadth but faster in depth.

This difference has a long-term impact on several key system properties.

Reproducibility arises because behavior does not depend on situational decisions or implicit assumptions, but on explicit invariants. Identical initial conditions lead to identical results, regardless of context or person.

Maintainability improves because changes do not occur in a diffuse space of possibilities, but along clearly defined boundaries. Errors can be traced back to violations of invariants, and extensions remain locally confined.

Scalability becomes possible because additional complexity doesn't arise at the core, but rather in higher layers that build upon stable foundations. Growth means expansion, not restructuring.

Governance capability ultimately arises from the explicit definition of what is permitted and what is excluded. Only systems with clearly defined state spaces, boundaries, and termination conditions can be reliably controlled, audited, and accounted for.

The crucial point is the perspective: Causal coding is not an ideal or a theoretical luxury. It is a functional necessity as soon as systems reach a certain size, lifespan, or level of responsibility. Where stability, liability, and long-term viability become relevant, exploratory or probabilistic approaches are no longer sufficient.

The superiority of causal coding therefore lies not in elegance or dogmatism, but in its ability to preserve value as complexity increases.

10. Conclusion: Orientation trumps freedom

Considering the preceding connections, a consistent picture emerges. Systems do not fail because they lack sufficient freedom, but because they lack orientation. Where no clear invariants, limits, and impossibilities are defined, behavior can neither emerge in a stable nor reproducible manner.

Freedom without orientation expands the realm of possibilities without making it controllable. It generates activity, but not directed development. Only through clear reference points does freedom become effective, because it can unfold within a structured framework.

Causal coding addresses precisely this point. It doesn't replace creativity with rules, but rather creates the conditions for creativity to become sustainable. By defining what is impossible, it reveals what can be meaningfully developed further. Orientation arises not from retrospective correction, but from an upfront structure.

This makes it clear: speed, creativity, and stability are not opposites that need to be weighed against each other. They are the result of a system that knows and respects its limits.

Or, to put it more precisely:

Speed, creativity, and stability do not originate there,
where everything is possible,
but where it is clearly defined what is impossible.

License: CC BY-NC-ND 4.0
A notice: Definitions and terms according to the Strüver model (functional logic).
The content may be freely shared and used for non-commercial purposes, provided that the author is clearly credited:

"Based on the Strüver model (functional logic) – © Alexander W. Strüver"

When deriving or quoting individual terms, the marking must be made:

“Definition according to Strüver”

Any commercial use or distribution for a fee, especially in the context of teaching, therapy, or consulting services, is prohibited without a separate license agreement. Details and full license terms: https://alexstruever.com/lizenzen/