Ontology of Architectural Regimes in Computing

Growth Fronts, Closure, and Architectural Disposal Beyond Evolutionary Narratives

Alexey A. Nekludoff

AstraVerge Research

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

6 February 2026

Abstract

Computing history is frequently described through an evolutionary narrative in which architectures are said to "develop," "mature," or "progress," as if technical objects possessed an internal trajectory of improvement. This paper rejects that framing on ontological grounds. Technical objects do not evolve. They are created, stabilized, reproduced, and ultimately disposed through externally governed architectural regime changes.

Adopting an architectural ontology in which architecture is prior to dynamics and observation, the paper introduces the notion of *architectural regimes* in computing as modes of structural unity maintained under specific constraints. Three regime operators are employed: *growth fronts*, marking zones of architectural indeterminacy; *closure points*, fixing regimes for reproduction through stabilization and standardization; and *odd remainders*, representing unresolved structural tensions that drive regime degradation and replacement.

On this basis, the history of computing is reconstructed as a sequence of discontinuous architectural regimes rather than as continuous technological progress. The analysis spans pre–von Neumann architectural plurality, the von Neumann closure and its long reproduction phase, the transition to parallel and multi-core regimes under integration limits, and contemporary cloud and distributed architectures as a partially closed regime with active growth fronts. Architectural disposal is formalized as an ontological event: the loss of a regime of unity, not the appearance of a superior technology.

The contribution of the paper is neither a new chronology of computing nor a theory of innovation, but a structural language for distinguishing architectural existence from dynamic persistence and regime replacement from optimization. This language supports diagnostic analysis of contemporary computing without recourse to evolutionary metaphors or technological futurism.

Keywords

architectural ontology; architectural regimes; computing history; standardization; closure; growth fronts; unresolved remainder; disposal; regime transitions

Contents

1 Introduction **1**

2 Definitions and Notation **1**
2.1 Architecture . . . 2
2.2 Architectural Regime . . . 2
2.3 Growth Front . . . 2
2.4 Closure Point . . . 2
2.5 Odd Remainder . . . 3
2.6 Architectural Invariants . . . 3
2.7 Architectural Degradation . . . 3
2.8 Architectural Disposal . . . 3
2.9 Notation Summary . . . 3

3 Architectural Regimes in Computing: A Structural Map **4**

4 Ontological Premises: Architecture Prior to Dynamics **4**

5 Regime Operators in Computing **5**
5.1 Growth Front as Architectural Indeterminacy . . . 5
5.2 Closure Point as Regime Fixation . . . 6
5.3 Odd Remainders as Drivers of Regime Breakdown . . . 6

6 Computing History as a Sequence of Architectural Regimes **6**
6.1 Regime I: Pre–von Neumann Plurality (c. 1930–1945) . . . 6
6.2 Regime II: Von Neumann Reproduction and Its Integration Limits . . . 7
6.3 Regime III: Parallelism, Multi-core, and GPUs (c. 1970–2000) . . . 8
6.4 Regime IV: Cloud and Distributed Architectures (c. 2005–present) . . . 8
6.5 Why This Periodization Is Not Optional . . . 9

7 Why "Evolution" Misdescribes Computing **10**

8 Architectural Degradation and Disposal **10**
8.1 Architectural Degradation . . . 11
8.2 From Degradation to Disposal . . . 11
8.3 Disposal as Regime Replacement . . . 11

9 Implications: Diagnostics Without Futurism **12**

10 Conclusion **12**

1 Introduction

Accounts of computing history are commonly framed in evolutionary terms. Architectures are said to "develop," "mature," or "progress," as if technical systems possessed an intrinsic trajectory of improvement. Such language compresses heterogeneous factors—engineering design, standardization, industrial organization, institutional adoption—into a quasi-natural process. The result is not merely rhetorical imprecision but an ontological displacement: agency is shifted from architectural governance to the technical objects themselves.

This paper argues that such accounts misdescribe the nature of architectural change in computing. Technical objects do not evolve. What changes are the architectural regimes under which objects are created, stabilized, reproduced, and eventually replaced. These regimes persist only so long as they maintain structural coherence: a condition in which architectural invariants remain internally integrable.

The central claim of this work is therefore simple but consequential: *architectures exist as unified entities only while a regime of structural coherence can be maintained*. When unresolved structural tensions exceed the integrative capacity of a regime, architectural unity is lost. Replacement follows not as improvement or optimization, but as an ontological transition to a different regime. Such transitions are discontinuous, externally governed, and irreducible to evolutionary narratives.

To make this claim operational, the paper introduces a regime ontology for computing. Rather than describing technological change through performance curves or innovation cycles, it analyzes computing history in terms of regimes of unity and the structural operators that act upon them. Growth fronts mark zones of architectural indeterminacy, closure points fix regimes for reproduction, and odd remainders accumulate as unresolved tensions that ultimately force regime termination and replacement.

Contributions.

- A regime-level vocabulary for computing architectures, grounded in architectural coherence rather than behavioral continuity.
- A structural reconstruction of computing history as a sequence of discontinuous regime closures, reproductions, and replacements.
- An ontological account of architectural degradation and disposal, distinguishing dynamic persistence from architectural existence.
- A diagnostic framework for identifying unresolved structural tensions in contemporary computing without recourse to technological futurism.

By shifting the analytical focus from objects to regimes, and from narratives of progress to conditions of unity, the paper aims to provide a structurally grounded account of computing architecture that remains intelligible across historical periods without animating technical artifacts or projecting developmental agency onto them.

2 Definitions and Notation

This section fixes the technical meaning of the core terms used throughout the paper. All definitions are structural rather than descriptive. They do not refer to observable behavior, performance metrics, or implementation details, but to conditions of architectural existence and persistence.

2.1 Architecture

Architecture denotes the relational structure that defines the conditions under which a system exists as a unified entity. Architecture specifies admissible states, relations, and invariants. It is ontologically prior to both dynamics and observation.

Architecture is not:

- a diagram or representation,
- a description of behavior,
- a collection of components.

Architecture is the condition that makes components, behaviors, and observations meaningful as parts of a system.

2.2 Architectural Regime

An **architectural regime** is a mode of structural unity under which an architecture maintains coherence. A regime is defined by:

- a set of architectural invariants,
- a bounded space of admissible configurations,
- specific mechanisms of integration and compensation of structural tensions.

Architectural regimes are not gradual states. A system either exists within a regime or has exited it through loss of coherence.

2.3 Growth Front

A **growth front** is the minimal locus within an architectural regime where continuation remains possible. It marks the boundary between stabilized structure and open architectural possibility.

Formally, a growth front exists if and only if the architecture contains at least one structural degree of freedom that cannot be internally compensated within the current regime.

In computing systems, growth fronts typically appear as:

- unresolved architectural alternatives,
- non-finalized standards or interfaces,
- competing paradigms that cannot be reduced to parameter tuning.

2.4 Closure Point

A **closure point** is a locally complete architectural configuration that suppresses continuation at a given level of description. Closure occurs when all degrees of freedom relevant at that level are internally compensated.

Closure does not imply:

- optimality,
- permanence,
- absence of future change.

Closure marks a transition from development to reproduction: the architecture persists by replication rather than by structural extension.

2.5 Odd Remainder

An **odd remainder** is a minimal non-compensable structural tension within an architectural regime. It is not numerical and not quantitative; it is defined by the absence of an internal architectural compensator.

The persistence of an odd remainder is sufficient to:

- prevent architectural closure,
- sustain a growth front,
- force continuation or regime transition.

Elimination of all odd remainders at a given level results in local closure.

2.6 Architectural Invariants

Architectural invariants are relational conditions that must hold across all admissible configurations of a regime. Violation of an invariant constitutes architectural degradation, even if dynamic behavior remains observable.

Invariants are not performance constraints; they are conditions of unity.

2.7 Architectural Degradation

Architectural degradation is the loss of coherence within a regime due to invariant violation or externalization of integration. Degradation may remain latent: systems can continue to operate dynamically after architectural unity has been lost.

2.8 Architectural Disposal

Architectural disposal denotes the termination of a regime as a unified architectural entity. Disposal occurs when degradation reaches a point at which the regime can no longer integrate its own unresolved remainders.

Disposal is an ontological event, not a market signal and not mere obsolescence. It marks the replacement of one architectural regime by another.

2.9 Notation Summary

For reference, the following notation is used throughout the paper:

Symbol / Term	Meaning
Architecture	Relational condition of system existence
Regime	Mode of maintained architectural unity
Growth front	Locus of unresolved architectural continuation
Closure point	Locally complete configuration suppressing continuation
Odd remainder	Non-compensable structural tension
Invariant	Condition required for regime coherence
Degradation	Loss of architectural coherence
Disposal	Ontological termination of a regime

This notation is intentionally minimal. It is sufficient to distinguish architectural existence from dynamic behavior and regime transitions from continuous variation.

3 Architectural Regimes in Computing: A Structural Map

To make the regime-based interpretation explicit, Table 1 summarizes the major architectural regimes in the history of computing. The table should be read structurally rather than chronologically: it identifies regimes by their mode of unity, closure conditions, and unresolved remainders, not by technological milestones alone.

Table 1: Architectural regimes in computing

Regime	**Growth Front**	**Odd Remainders**	**Closure Mechanism**	**Disposal Trigger**
Pre–von Neumann (c. 1930–1945)	Plural machine models; competing memory and programming concepts	Universality; program storage; synchronization; operational control	Von Neumann architecture as a unifying program–memory model and reproducible abstraction	Inability to scale plurality into an industrially reproducible unity
Von Neumann (Reproduction regime)	Incremental optimization; microarchitectural tuning	Memory bottleneck; sequential execution; coordination limits	Standardized CPU/memory separation; stable programming model; industrial ecosystem	Accumulation of integration limits exceeding internal compensation
Parallel / Multi-core (c. 1970–2000)	Vectorization; multiprocessing; pipeline execution; cache hierarchies	Thermal limits; frequency scaling ceiling; memory latency and bandwidth	Multi-core CPUs and GPU-style accelerators as stabilized parallel regimes	Energy and coordination costs dominate gains from further scaling
Cloud / Distributed (c. 2005–present)	Distributed systems; microservices; container orchestration; serverless paradigms	Latency; data consistency; coordination overhead; energy consumption	Loss of integrability of distribution costs; emergence of incompatible scaling constraints	

4 Ontological Premises: Architecture Prior to Dynamics

This work adopts an explicitly architectural stance. Dynamics are treated as descriptions of state transitions *within* a space of admissibility, whereas architecture defines the conditions under which such a space exists at all. In this sense, dynamics presuppose architecture, but architecture is not derivable from dynamics.

Observable behavior, execution traces, and performance metrics describe how a system behaves once an architectural regime is in place. They do not specify the relational constraints, invariants, or boundaries that constitute the system as a unified entity. Consequently, functional equivalence or behavioral similarity cannot be taken as evidence of architectural identity. Distinct architectures may induce indistinguishable dynamics under observation.

This asymmetry establishes a fundamental ontological boundary: architectural structure is not reconstructible from dynamics alone. No accumulation of observational data suffices to recover architecture once its relational coherence has been lost. The present paper does not re-argue this claim in full; it adopts it as a fixed ontological ground developed elsewhere [1].

Within this framework, architectural unity is not a static property of objects but a condition that must be actively maintained. A system exists as a coherent architectural entity only so long as its defining relations remain compatible and internally integrated. Persistence of behavior does not guarantee persistence of unity: systems may continue to operate dynamically after architectural coherence has already degraded.

This distinction is crucial for understanding technological change. Architectural regimes do not transition because performance metrics gradually improve or degrade, nor because behavior becomes suboptimal under some external criterion. Regime transitions occur when unresolved structural tensions exceed the capacity of the current architecture to integrate them without violating its invariants. At that point, unity is lost and replacement becomes unavoidable.

Two consequences follow directly from this stance.

First, architectural unity must be treated as a maintained condition rather than as an inherent attribute of technical objects. Objects do not carry unity intrinsically; unity is conferred by the architectural regime that governs their creation, integration, and reproduction.

Second, regime transitions are not reducible to behavioral descriptions or optimization narratives. Performance regressions, bottlenecks, or inefficiencies may signal deeper structural tensions, but they do not constitute the transition itself. The transition is ontological: it is the loss of a regime of coherence and the emergence of a new architectural condition of existence.

By fixing architecture as ontologically prior to dynamics, this paper establishes the conceptual basis for a regime-level analysis of computing. Growth fronts, closure points, odd remainders, and architectural disposal are introduced not as metaphors of technological change, but as structural operators that act on regimes of unity rather than on objects or behaviors.

5 Regime Operators in Computing

The purpose of this section is not to redefine the regime operators introduced earlier, but to specify how they *act* within computing architectures. Growth fronts, closure points, and odd remainders are treated here as operators on architectural regimes rather than as descriptive labels. Their role is to distinguish phases of architectural continuation, reproduction, and breakdown that cannot be captured by performance metrics or incremental design narratives [2].

5.1 Growth Front as Architectural Indeterminacy

In computing, a growth front does not correspond to novelty in the sense of innovation or experimentation. It corresponds to *architectural indeterminacy*: a regime in which the system cannot yet decide, within its own constraints, how it must be structured.

A growth front is present when:

- multiple architectural decompositions remain admissible,
- key interfaces have not yet been fixed,
- architectural decisions cannot be reduced to parameter tuning within an existing design.

Under these conditions, architectural change is not incremental. Competing solutions are not variants of the same architecture but candidates for closure. Growth fronts therefore mark periods in which

computing systems are architecturally unfinished, even when they are functionally operational.

5.2 Closure Point as Regime Fixation

A closure point in computing is not the adoption of a "better" solution but the fixation of architectural invariants that suppress further indeterminacy. Closure occurs when an architecture becomes sufficiently specified that alternative decompositions are no longer structurally admissible.

In practice, closure manifests as:

- the stabilization of instruction sets, execution models, or coordination mechanisms,
- the fixation of interfaces that constrain future extensions,
- the emergence of an ecosystem oriented toward reproduction rather than redesign.

Closure transforms architectural change into optimization. Once a closure point is reached, variation continues, but it does so within a regime whose fundamental structure is no longer open to revision without rupture. Computing architectures persist in this mode not because they are optimal, but because they are *internally integrable*.

5.3 Odd Remainders as Drivers of Regime Breakdown

Odd remainders are the structural tensions that resist integration within a closed regime. They are not anomalies or implementation defects; they are indicators that the architecture has exhausted its capacity to compensate its own constraints.

In computing systems, odd remainders typically emerge as:

- bottlenecks that cannot be removed without violating architectural invariants,
- scaling limits that force external coordination or ad-hoc extensions,
- conflicts between performance, correctness, and integrability.

As long as odd remainders remain integrable, a regime persists. When they accumulate beyond the capacity of internal compensation, architectural degradation begins. At this point, continuation within the same regime becomes impossible: further change requires a transition to a different architectural regime.

Together, these operators define a structural cycle characteristic of computing history: growth fronts open architectural possibility, closure points stabilize regimes for reproduction, and odd remainders accumulate until the regime can no longer sustain unity. This cycle is discontinuous, externally governed, and irreducible to evolutionary narratives of gradual improvement.

6 Computing History as a Sequence of Architectural Regimes

This section reconstructs the history of computing as a sequence of architectural regimes rather than as a continuous evolution of machines. Each regime is characterized by a specific mode of unity, a bounded space of admissible architectures, and a characteristic pattern of unresolved structural tensions. Transitions between regimes are discontinuous and externally governed; they correspond to closure events rather than gradual refinement.

6.1 Regime I: Pre–von Neumann Plurality (c. 1930–1945)

Early computing did not begin with a single architectural paradigm but with a heterogeneous field of machine concepts. Mechanical, electromechanical, and early electronic devices were developed inde-

pendently, often tailored to narrow problem classes. Notable examples include differential analyzers, relay-based calculators, and early electronic machines such as the Atanasoff–Berry Computer and Colossus.

Growth front. The defining feature of this period was architectural plurality. Memory, control, and programming were not yet unified concepts. Programs were realized through physical rewiring, plugboards, or fixed control mechanisms. Competing notions of computation coexisted without a dominant decomposition of machine structure [3, 4].

Odd remainders. Several unresolved tensions accumulated across these designs:

- lack of universality and reuse,
- absence of a unified notion of stored programs,
- difficulty of synchronization and control in increasingly complex machines,
- tight coupling between problem specification and machine configuration.

These tensions could not be resolved through incremental improvement of any single design. They pointed to the absence of a coherent architectural regime rather than to local engineering deficiencies.

Closure. The von Neumann architecture functioned as a closure point by introducing a reproducible unity: separation of processing and memory, symbolic programs stored alongside data, and a control structure independent of physical rewiring. Importantly, this closure was not merely technical but pedagogical and industrial: it produced an architecture that could be taught, replicated, and scaled institutionally [5].

6.2 Regime II: Von Neumann Reproduction and Its Integration Limits

Once established, the von Neumann architecture entered a long reproduction phase. For decades, architectural variation occurred primarily within a fixed decomposition: single-threaded execution, hierarchical memory, and centralized control. Innovation focused on speed, reliability, and cost rather than on architectural redefinition.

Reproduction regime. During this phase, architectural change took the form of microarchitectural refinement: instruction pipelining, caching, branch prediction, and incremental increases in clock frequency. These developments preserved the core invariants of the regime while extending its operational envelope [6].

Odd remainders accumulating under reproduction. Over time, structural tensions accumulated that could not be resolved internally:

- the memory wall, where latency and bandwidth failed to scale with processing speed,
- diminishing returns from instruction-level parallelism,
- coordination and coherence costs under increasing transistor counts.

These limits were not accidental. They marked the exhaustion of the regime's capacity to integrate further performance gains without violating its architectural invariants.

6.3 Regime III: Parallelism, Multi-core, and GPUs (c. 1970–2000)

The response to these integration limits was not a smooth extension but a regime shift. Parallelism emerged as a new growth front, reopening architectural indeterminacy at multiple levels simultaneously.

Growth front. Vector processors, shared-memory multiprocessing, message-passing systems, and deeply pipelined designs explored incompatible decompositions of computation. GPUs, initially developed as fixed-function accelerators, gradually evolved into programmable parallel devices, further expanding the architectural search space [7, 8].

Odd remainders. Despite gains, new tensions emerged:

- thermal and power-density limits preventing further frequency scaling,
- synchronization and coherence overheads in parallel execution,
- persistent imbalance between compute throughput and memory access.

Closure. Multi-core CPUs and GPU-style accelerators stabilized as dominant forms. Parallelism ceased to be experimental and became normalized within toolchains, programming models, and hardware ecosystems. Closure did not eliminate tension but fixed a new regime of unity in which parallel execution was treated as the default rather than as an exception.

6.4 Regime IV: Cloud and Distributed Architectures (c. 2005–present)

The most recent regime shift occurred not at the level of individual machines but at the level of system composition. Cloud and distributed architectures displaced the single-system boundary as the primary locus of architectural coherence.

Growth front. Service decomposition, containerization, orchestration platforms, and serverless execution models reopened fundamental questions about state, coordination, and responsibility boundaries. Architectural decisions increasingly concern deployment topology and operational control rather than instruction execution [9, 10].

Odd remainders. Unresolved tensions are now dominated by:

- latency and partial failure in distributed environments,
- trade-offs between consistency, availability, and coordination,
- escalating operational complexity and energy consumption at scale.

Consistency as an Odd Remainder (The CAP Case). A paradigmatic example of an unresolved structural tension in distributed architectures is captured by the CAP formulation. Within the present framework, CAP is not interpreted as a theorem about system behavior, but as a manifestation of an architectural odd remainder.

Consistency, availability, and partition tolerance are not competing implementation goals. They correspond to incompatible architectural invariants that cannot be jointly integrated within a single closed regime. The persistence of this incompatibility indicates not a design trade-off but the absence of architectural closure.

From a regime perspective, the significance of CAP lies precisely in its resistance to resolution. Decades of system design have not eliminated the underlying tension; they have only produced localized compensations, context-dependent relaxations, and operational workarounds. This persistence is diagnostic: it signals that the current distributed regime cannot internally compensate its own coordination constraints.

CAP therefore functions as an odd remainder in the strict sense. It is a minimal, non-compensable structural tension that forces architectural continuation. Attempts to "solve" CAP invariably result in either externalizing coherence (through operational policy, human intervention, or economic constraint) or fragmenting the architecture into partially incompatible sub-regimes.

Seen in this light, CAP is not a limitation of distributed systems but an indicator of regime status. Its continued relevance confirms that cloud and distributed computing remain within an active growth front rather than having reached a stable architectural closure.

Partial closure. Unlike earlier regimes, cloud computing exhibits only partial closure. Standards and dominant platforms exist, yet no stable and universally integrable set of invariants has emerged. Architectural unity remains externally enforced through orchestration, policy, and economic constraints rather than through internal coherence alone. This indicates an active growth front rather than a fully stabilized regime.

6.5 Why This Periodization Is Not Optional

The regime-based periodization proposed here is not an interpretative preference and not a historiographic convenience. It follows directly from the ontological premises adopted in this work. Once architecture is treated as ontologically prior to dynamics, periodization cannot be based on performance curves, technological milestones, or continuity of usage. It must track changes in the conditions of architectural unity.

Any alternative periodization implicitly commits to at least one of the following assumptions: (i) that architectural identity can be inferred from behavioral continuity; (ii) that gradual performance improvement constitutes structural continuity; or (iii) that technological artifacts possess an internal trajectory of development. All three assumptions contradict the architectural stance adopted here.

Under an architectural ontology, regimes are individuated by their invariants and by the mechanisms through which unresolved structural tensions are integrated or suppressed. A regime change therefore occurs precisely when such integration becomes impossible within the existing architectural constraints. This criterion is neither subjective nor historical; it is structural.

The proposed periodization is minimal in this sense. It introduces no additional regimes beyond those required to account for observed architectural breaks: the transition from plurality to reproducible unity (pre–von Neumann to von Neumann), the exhaustion of sequential integration (von Neumann to parallel), and the displacement of the system boundary itself (single-machine to distributed regimes).

Importantly, these transitions are not optional reinterpretations of history. They correspond to points at which architectural invariants were either fixed or abandoned. Where invariants persist, the regime persists regardless of surface-level change. Where invariants fail, no amount of dynamic continuity suffices to preserve regime identity.

This explains why the present periodization is coarse rather than granular. Architectural regimes do not admit fine subdivision without reintroducing behavioral or technological criteria that are ontologically secondary. The aim is not to catalogue innovations, but to identify the minimal sequence of regimes required for architectural coherence to be maintained over time.

7 Why "Evolution" Misdescribes Computing

The evolutionary metaphor [1] misdescribes computing not because it is imprecise, but because it introduces an ontological error. In biological contexts, evolution refers to a process grounded in internal mechanisms of variation, selection, and inheritance. These mechanisms operate within the entities that evolve. None of these conditions hold for technical objects.

Technical objects do not reproduce themselves, do not generate inheritable variation, and do not participate in selection processes internal to their own existence. They are created, modified, replicated, and discarded through externally governed architectural decisions. To speak of their "evolution" is therefore to ascribe to them a form of agency they do not possess.

The persistence of the evolutionary metaphor in computing discourse has a specific consequence: it masks the role of architectural governance. By presenting historical change as a quasi-natural process, responsibility is displaced from design, standardization, institutional choice, and operational constraint onto the objects themselves. Architectural breaks are reinterpreted as "natural progress," and discontinuities are smoothed into narratives of gradual improvement.

This masking effect becomes particularly problematic at regime boundaries. Architectural closure points, where invariants are fixed and reproduction begins, are misread as stages of maturation. Conversely, architectural disposal, where a regime loses its capacity for coherence, is described as obsolescence rather than as the termination of a mode of unity. In both cases, the decisive structural events disappear behind descriptive continuity.

The regime vocabulary introduced in this paper avoids this confusion by preserving change without animating objects. Growth fronts, closure points, and odd remainders are not metaphors of development; they are operators acting on architectural regimes. Change is attributed explicitly to the mechanisms that govern architectural unity: design choices, standardization processes, integration constraints, and institutional reproduction.

Under this vocabulary, continuity is no longer taken as evidence of identity. A regime may persist dynamically while having already lost architectural coherence, just as a new regime may appear abruptly without genealogical continuity of objects. What matters is not whether a system looks similar to its predecessor, but whether it operates under the same conditions of architectural unity.

Abandoning the evolutionary metaphor is therefore not a matter of terminology but of analytical necessity. Without this shift, computing history cannot be described without conflating architectural replacement with development and structural rupture with optimization. The regime framework restores this distinction and makes architectural change intelligible without recourse to naturalized narratives.

8 Architectural Degradation and Disposal

Architectural disposal is not a market event, a lifecycle phase, or a consequence of technological obsolescence. It is an ontological event: the loss of a regime of architectural unity. Disposal marks the point at which an architecture can no longer exist as a coherent regime, regardless of whether its artifacts remain operational or economically viable.

[1] The framework developed here is not a variant of process philosophy applied to technology. It rejects, at the ontological level, the attribution of immanent processes of individuation, development, or evolution to technical objects. Processuality is not located in artifacts but in architectural regimes: in the externally governed conditions that create, stabilize, reproduce, and replace technical objects. This position explicitly diverges from process-oriented philosophies of technology, most notably those associated with Gilbert Simondon, in which technical artifacts are treated as bearers of internal developmental trajectories. Any account that assigns processuality to technical objects rather than to the architectures that govern their existence commits a categorical error by animating artifacts and systematically misidentifying architectural replacement as internal development, thereby obscuring the locus of architectural agency.

A critical distinction must therefore be drawn between *architectural persistence* and *dynamic persistence*. Systems may continue to execute, deliver services, and even scale in limited ways after architectural unity has already been compromised. Observable behavior alone is insufficient to determine whether a regime still exists as an architectural whole.

8.1 Architectural Degradation

Architectural degradation denotes the progressive loss of coherence within a regime. It occurs when architectural invariants are violated, bypassed, or externalized in order to maintain dynamic operation. Degradation is often latent: it may remain invisible to users and even to designers while accumulating structural contradiction.

Typical indicators of architectural degradation include:

- increasing reliance on ad-hoc constraints and compensatory mechanisms,
- externalization of coherence into operational policy, human intervention, or economic pressure,
- proliferation of patches, wrappers, and exception-handling structures that preserve behavior while undermining unity,
- divergence between nominal architectural descriptions and actual system operation.

Crucially, degradation is not synonymous with failure. On the contrary, highly degraded architectures often appear robust precisely because coherence is maintained through suppression rather than integration. This appearance of stability delays recognition of regime exhaustion.

8.2 From Degradation to Disposal

Disposal occurs when architectural degradation reaches a point at which the regime can no longer reproduce itself without expanding contradiction. At this stage, internal integration mechanisms are exhausted, and any further extension requires violating additional invariants or introducing new external dependencies.

Formally, architectural disposal is triggered when:

- unresolved odd remainders exceed the regime's capacity for internal compensation,
- coherence becomes dependent on external, non-architectural constraints,
- reproduction of the regime amplifies rather than resolves structural tension.

Disposal does not imply immediate disappearance. Legacy systems may persist for extended periods after disposal, supported by institutional inertia, regulatory constraint, or sunk cost. However, such persistence is post-architectural: the regime no longer governs its own unity.

8.3 Disposal as Regime Replacement

Architectural disposal is inseparable from regime replacement. A disposed regime does not transform into a new one; it is superseded. Replacement introduces a new set of architectural invariants and a new space of admissibility, often incompatible with the disposed regime's assumptions.

This perspective clarifies why architectural transitions are experienced as disruptive rather than evolutionary. Disposal is not improvement and not optimization; it is termination. The continuity that follows is institutional or operational, not architectural.

By treating disposal as an ontological event, rather than as a market outcome or technological fashion, this framework distinguishes between systems that merely continue to function and architectures that

continue to exist. The distinction is essential for understanding why computing regimes persist long after their coherence has been lost and why architectural change occurs through replacement rather than gradual refinement.

9 Implications: Diagnostics Without Futurism

The framework developed in this paper is intentionally non-predictive. It does not aim to forecast technological outcomes, identify winning paradigms, or extrapolate performance curves. Its practical value is diagnostic rather than prognostic: it provides structural criteria for determining the current ontological status of an architectural regime.

From a regime perspective, the relevant questions are not "what comes next?" but "what kind of unity is currently being maintained?" and "under what conditions does this unity fail?" The framework distinguishes three diagnostically significant states: (a) reproduction within a closed regime, (b) partial closure under accumulating tension, and (c) approach to architectural break driven by unresolved odd remainders.

A regime is in reproduction when variation remains internally integrable and architectural invariants are preserved. Partial closure is characterized by increasing reliance on external constraints to maintain coherence, while the core invariants remain nominally intact. An approaching break is indicated when unresolved remainders proliferate and further reproduction amplifies contradiction rather than resolving it.

This diagnostic stance avoids a common error in discussions of technological futures: the conflation of architectural pressure with imminent replacement. High tension does not imply that a new regime is ready or inevitable; it only indicates that the existing regime is losing its capacity for internal integration. Replacement remains contingent, externally governed, and often delayed by institutional, economic, or operational factors.

Within contemporary computing, several structural pressure zones can be identified without invoking futurist narratives:

- energy and thermal limits that constrain further scaling and shift coherence into economic and infrastructural domains,
- persistent imbalance between memory and computation across scales, indicating limits of integrability rather than temporary inefficiency,
- correctness under distribution, including consistency, coordination, and verification, where architectural coherence is increasingly externalized into policy and tooling,
- alternative paradigms such as quantum or neuromorphic computing, not as predicted successors, but as competing growth fronts that remain architecturally indeterminate.

The framework deliberately refrains from ranking these pressures or assigning them future trajectories. Its contribution lies in making visible the difference between regimes that continue to exist architecturally and those that persist only dynamically. By replacing prediction with diagnosis, it allows architectural analysis to remain structurally grounded without lapsing into speculative futurism.

10 Conclusion

This paper has argued that computing history is not the evolution of technical objects but a sequence of architectural regimes. These regimes are created, stabilized, closed, reproduced, degraded, and ultimately

disposed through externally governed transitions. Change is not immanent to artifacts; it is structural and architectural in origin.

By introducing regime operators—growth fronts, closure points, odd remainders, and architectural disposal—the paper provides a language in which computing can be described without recourse to evolutionary metaphors or narratives of inevitable progress. This language makes it possible to distinguish architectural unity from dynamic persistence, reproduction from development, and replacement from improvement.

Reconstructing computing history through regimes reveals why architectural breaks are discontinuous, why legacy systems persist after coherence has been lost, and why certain structural tensions resist resolution despite decades of optimization. What appears as gradual technological change is shown instead to be a sequence of regime closures and terminations, separated by periods of reproduction and latent degradation.

The contribution of this work is therefore not a new history of computing, nor a theory of innovation, but an ontological clarification. It fixes architecture as prior to dynamics, regimes as prior to objects, and disposal as a condition of architectural replacement rather than market obsolescence. With these distinctions in place, architectural change becomes intelligible without animation of artifacts or projection of developmental agency onto technical systems.

In doing so, the framework establishes a basis for diagnostic analysis of contemporary computing without futurism. It allows regimes to be identified by their conditions of unity and by the tensions that threaten them, rather than by predictions of what will come next. This shift—from narrative progress to structural diagnosis—marks the central outcome of the regime-based ontology developed here.

References

[1] A. A. Nekludoff, "Architecture of complex systems," Jan. 17, 2026, Preprint.

[2] A. A. Nekludoff, "The ontology of continuation: Growth fronts, closure points, and the stabilizing role of atomic hydrogen," Feb. 3, 2026, Preprint.

[3] P. E. Ceruzzi, *A History of Modern Computing*. MIT Press, 2003.

[4] T. Haigh, M. Priestley, and C. Rope, *Colossus: The Secrets of Bletchley Park's Codebreaking Computers*. Oxford University Press, 2014.

[5] J. von Neumann, "First draft of a report on the edvac," 1945, Moore School of Electrical Engineering.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[7] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, 1972.

[8] D. A. Patterson, "The trouble with multicore," *IEEE Spectrum*, 2010.

[9] M. e. a. Armbrust, "A view of cloud computing," *Communications of the ACM*, 2010.

[10] S. Newman, *Building Microservices*. O'Reilly Media, 2015.