Processual Memory Architecture:
A Transformation-Based Framework for Verifiable Computation and Safety-by-Construction AGI

William D. Diacont
Luminareware LLC
Lutz, Florida, USA
February 2026

Abstract

We present **Processual Memory Architecture (PMA)**, a computational framework that unifies data storage and computation by representing all information as transformation functions rather than static state, rendering the traditional ontological distinction between them architecturally unnecessary. In PMA, storing information means encoding it as a mathematical transformation that produces the data when applied to a standardized canonical input; reading means applying the transformation; and computing means composing transformations. This inversion of the conventional von Neumann paradigm yields five emergent architectural properties—structural auditability, transparent reasoning, enforced constraints, tamper evidence, and reversibility—that collectively enable *verifiable computation*: systems that can mathematically verify the integrity and correctness of their own reasoning chains.

We provide a complete mathematical specification of PMA over Galois fields GF(2^k) with round-trip exactness guarantees, constructive algorithms for both invertible and non-invertible encoding modes, and a reference permutation-based embodiment with explicit bit-level storage formats. We analyze thermodynamic properties under reversible logic implementation, demonstrating that PMA operations on adiabatic substrates can approach within 10× of the Landauer limit at the local-node level. We then present the integration architecture for PMA with artificial general intelligence (AGI) safety frameworks, showing how transformation-based reasoning enables safety constraints that are structural rather than advisory—creating systems where unsafe behavior is computationally undefined rather than merely prohibited. We discuss applications to financial auditing, medical AI verification, and autonomous systems governance, and compare PMA's approach to verifiable computation with existing paradigms including blockchain, zero-knowledge proofs, and mechanistic interpretability.

Keywords: computer architecture, verifiable computation, reversible computing, category theory, process ontology, AGI safety, transformation composition, Galois fields, structural auditability, von Neumann bottleneck

1 Introduction

Since von Neumann's foundational 1945 report [26], virtually all practical computing systems have maintained a fundamental ontological distinction: data is static state stored in memory, and computation is a dynamic process that acts upon that state. This distinction—so deeply embedded in computing practice that it is rarely questioned—creates an inherent asymmetry between what information *is* (passive patterns of bits) and what computation *does* (active manipulation of those patterns). The resulting architecture requires continuous shuttling of data between storage and processing, creating the well-known von Neumann bottleneck [1] that constrains modern systems despite decades of mitigation efforts.

This paper introduces Processual Memory Architecture (PMA), a framework that unifies data storage and computation, rendering the traditional distinction between them architecturally unnecessary. In PMA, information is represented not as static state but as *transformation functions*—mathematical mappings that produce observable data only when applied to a standardized canonical input. Under this representation, storing information means encoding it as a transformation; reading information means applying the transformation to the canonical input; and computing means composing transformations. Storage and computation become the same operation viewed from different perspectives.

This conceptual inversion draws from two intellectual traditions that have not previously been applied to computer architecture. *Category theory* [2, 3] emphasizes morphisms (structure-preserving mappings) over objects, and the Yoneda lemma establishes that objects are fully characterized by their relationships to other objects. *Process philosophy* [4], developed by Whitehead, argues that the fundamental constituents of reality are dynamic processes rather than static substances. PMA operationalizes both insights: information is process (transformation), and identity is relationship (to the canonical input).

The architectural consequences of this shift are substantial. We identify five properties that emerge naturally from transformation-based information representation:

1. Structural Auditability. Every computational result carries the complete chain of transformations that produced it. The audit trail is not maintained separately from computation—it *is* the computation.

2. Transparent Reasoning. Each transformation in a composition chain represents a discrete reasoning step that can be individually inspected. Explanations are read from the transformation chain, not generated after the fact.

3. Enforced Constraints. Required transformations can be made structural components of valid computation. A reasoning chain missing a required transformation is invalid by construction, not merely non-compliant with policy.

4. Tamper Evidence. Transformation chains are cryptographically bound through canonical input derivation. Unauthorized modifications break mathematical consistency and are detectable through verification.

5. Reversibility. For invertible transformations, any computation can be reversed to recover original inputs, enabling the compute-uncompute-verify pattern for mathematical verification of computational integrity.

These properties are not features added to PMA; they are emergent consequences of the foundational design choice to represent information as transformation. Collectively, they enable what we term *verifiable computation*: systems that can verify the integrity, correctness, and completeness of their own reasoning.

The implications for artificial general intelligence (AGI) safety are particularly significant. Current approaches to AGI alignment rely on training (hoping systems learn correct behavior), testing (sampling outputs), and interpretability research (attempting to understand internal representations after the fact) [5, 6]. These approaches, while valuable, ultimately depend on trust assumptions that cannot be mathematically verified. PMA offers an alternative: an architecture where safety constraints are structural components of valid computation, creating systems where unsafe behavior is *undefined* rather than merely prohibited.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the formal mathematical framework. Section 4 specifies finite-precision implementation over Galois fields. Section 5 analyzes thermodynamic properties. Section 6 presents the integration with AGI safety architecture. Section 7 discusses applications. Section 8 compares PMA with alternative verification paradigms. Section 9 addresses limitations and open problems. Section 10 concludes.

2 Related Work

2.1 Von Neumann Bottleneck Mitigation

The performance limitations of the von Neumann architecture have motivated extensive research into alternative computing paradigms. Cache hierarchies, prefetching, and multi-threading reduce the *frequency* of bottleneck encounters but retain the fundamental data/computation separation. Near-memory computing [8] and processing-in-memory [7] reduce *distance* between data and computation but retain the ontological distinction. Neuromorphic architectures [9, 10] collocate memory and processing, drawing inspiration from biological neural networks, but still treat data as state acted upon by computational processes. PMA differs from all of these by unifying data and computation at the architectural level: information does not *reside near* computation; information *is* computation.

2.2 Reversible Computing

Landauer [11] established the thermodynamic cost of information erasure, and Bennett [12] demonstrated that computation can be performed reversibly, avoiding this cost in principle. Toffoli [13] and Fredkin and Toffoli [14] provided universal reversible gate sets. Recent commercial interest in energy-efficient computing has renewed attention to reversible architectures, with Vaire Computing demonstrating a reversible test chip in 22nm CMOS achieving energy recovery factors of 1.77× (capacitor arrays) and 1.41× (shift registers), and targeting commercial reversible silicon by 2027–2028 [15]. PMA is naturally aligned with reversible computing because transformation composition preserves information—no data is overwritten during computation. However, PMA's contributions extend beyond energy efficiency; the verification, auditability, and safety properties arise from the transformation-based representation independent of implementation substrate.

2.3 Verifiable and Transparent Computation

Blockchain technology [16] provides tamper-evident records of *events* but not verification of *reasoning*. Zero-knowledge proofs [17, 18] enable verification that a computation was performed correctly without revealing the computation itself, but require explicit proof construction separate from the computation. Verifiable computation schemes [19] delegate computation to untrusted parties with proof of correctness, but add substantial overhead. Database provenance research [27, 28] has explored propagating derivation history alongside query results, but operates as a metadata layer atop conventional storage rather than as the computational substrate itself. PMA's approach differs fundamentally from all of these: verification is not a separate process applied to computation but an intrinsic property of how computation occurs. The transformation chain is the proof.

2.4 AI Safety and Interpretability

Current AI safety research addresses the challenge of ensuring AI systems behave as intended through several approaches. Constitutional AI [5] trains models with principle-based feedback. Mechanistic interpretability [20, 21] seeks to reverse-engineer internal representations of trained models. Alignment research [22] attempts to specify and instill correct values. These approaches work within the existing neural network paradigm, attempting to understand or constrain systems after they are built. PMA proposes a complementary approach: designing the computational substrate so that safety properties are structural. This is analogous to the distinction between inspecting a building for structural integrity after construction versus designing the building with load-bearing constraints that make structural failure physically impossible.

2.5 Category Theory in Computing

Category theory has been applied to programming language theory [23], database theory [24], quantum computing [25], and circuit-level hardware semantics [29, 30], but has not been applied

to the fundamental question of how data is represented and stored at the architectural level. The categorical perspective—emphasizing morphisms over objects—aligns naturally with PMA's emphasis on transformations over static data. PMA can be formalized as a category where objects are canonical input/output types, morphisms are stored transformations, and composition of morphisms unifies storage and computation.

2.6 Functional Programming

Functional programming languages, particularly those in the ML/Haskell tradition following Backus's vision [1], share PMA's emphasis on transformation composition as a computational primitive. Lazy evaluation implements deferred computation; pure functions enable referential transparency; and combinator-based programming composes transformations algebraically. However, functional programming implements these properties *in software atop conventional von Neumann hardware*. The underlying substrate retains the data/computation distinction: functions are stored as instruction sequences in memory, and execution requires the conventional fetch-decode-execute cycle with its attendant data movement costs and irreversible state transitions.

PMA's contribution is not the paradigm of computation-as-composition—which functional programming has explored for decades—but the realization of this paradigm *as the computational substrate itself*. This distinction yields three properties unavailable to software-only approaches: (i) verification operates over algebraic structure rather than execution traces, enabling structural auditability without re-execution; (ii) reversibility is a physical property of the substrate, not a software abstraction, enabling the thermodynamic advantages analyzed in Section 5; and (iii) safety constraints are algebraically enforced at the architectural level rather than implemented as software-level type systems or runtime checks that can be circumvented by lower-level access. The closest software approximations—Haskell's ST monad for encapsulated mutation and linear types for resource tracking—illustrate both the aspiration and the limitation: they provide strong compile-time guarantees within the language, but these guarantees evaporate at the FFI boundary or under unsafe operations, precisely because they are enforced by a compiler rather than by the computational substrate.

3 Formal Framework

3.1 Definitions

Definition 3.1 (Processual Memory Architecture). A PMA system is a tuple $\langle R, C, \circ, O \rangle$ where R is a transformation register storing encoded transformation functions; C is a canonical input generator producing standardized reference signals; $\circ$ is a composition operator combining transformation functions; and O is an observation interface producing outputs by applying transformations to canonical inputs.

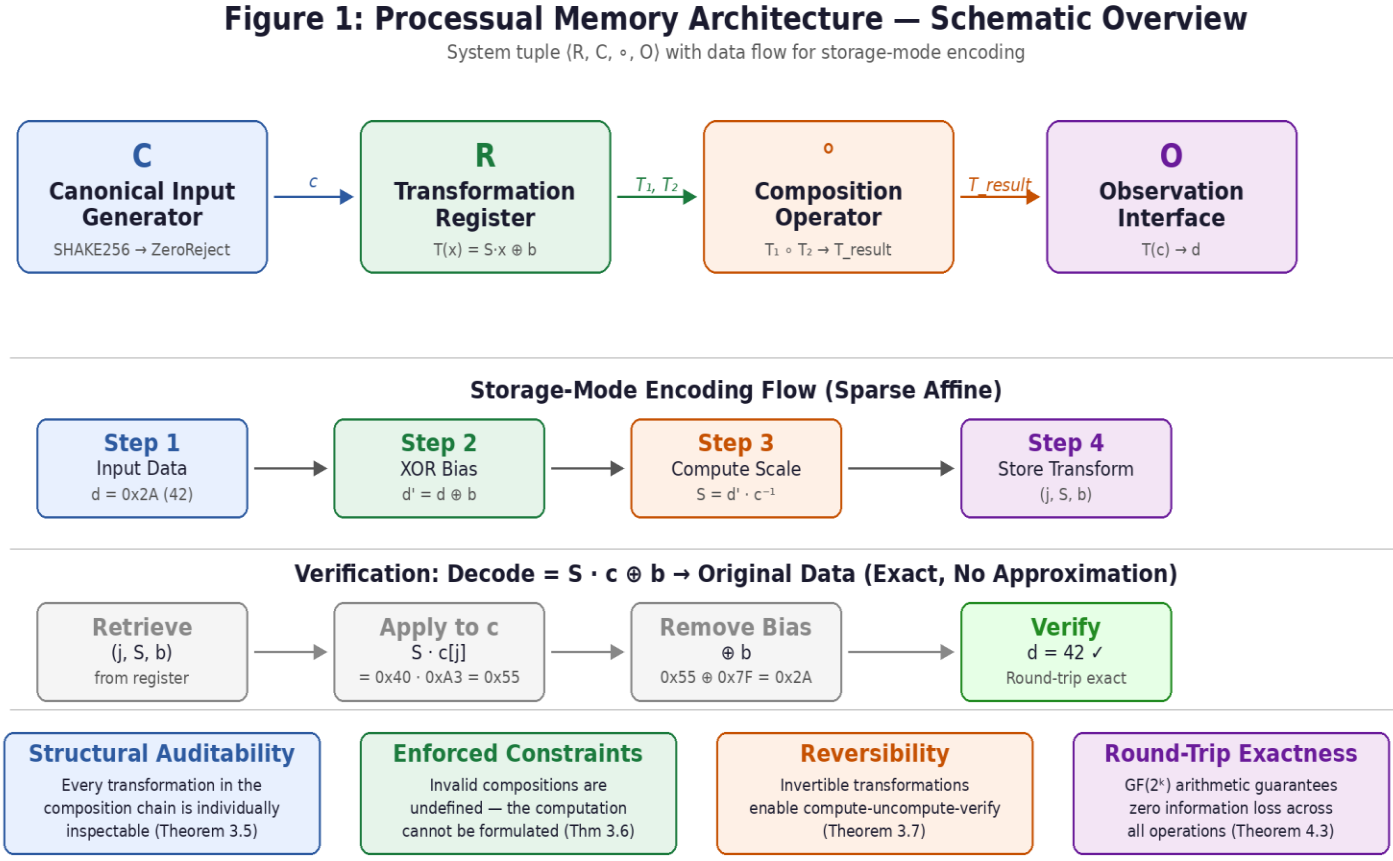

Figure 1. Schematic of the PMA system tuple (R, C, ∘, O) with storage-mode encoding and verification data flow. Top row: the four architectural components. Middle rows: step-by-step encoding and decoding using sparse affine transformations over GF(2⁸). Bottom row: the four structural properties guaranteed by the architecture.

Definition 3.2 (Transformation Encoding). For data value d and canonical input c, a transformation encoding is a function T such that T(c) = d. Storing d means computing and storing T; reading d means computing T(c).

Definition 3.3 (Computation as Composition). For transformations T_1 and T_2 representing operands, computation is defined as composition: $T_{result} = \Phi(T_1, T_2)$ where Φ is a composition rule corresponding to the desired operation. The result transformation T_{result} produces the computational result when applied to the canonical input.

Definition 3.4 (Canonical Input). The canonical input c is generated deterministically from a cryptographic seed: c = ZeroReject(SHAKE256(seed ‖ context, output_length)), where ZeroReject ensures $c \neq 0$ by iterating with an incremented salt until a nonzero output is obtained. For SHAKE256 producing an output of length matching the data vector dimension, the probability of an all-zero result is 2^{-L} where L is the output length in bits, so the expected number of iterations is essentially 1 for any practical L (e.g., 2^{-128} for L = 128 bits, corresponding to a single element of $GF(2^{128})$).

The canonical input generator must satisfy four properties: (i) *determinism* (reproducibility for verification); (ii) *collision resistance* (preventing cross-context transformation chain forgery); (iii) *pseudorandomness* (preventing adversarial selection of pathological data/canonical-input interactions); and (iv) *variable-length output* (matching the data vector dimension). SHAKE256 is an extendable-output function (XOF) from the SHA-3 family (NIST FIPS 202) satisfying all four requirements with 256-bit security; any XOF with equivalent security properties may be substituted. The tamper-evidence property (Property 4 in Section 1) depends on collision resistance and pseudorandomness: forging a transformation chain requires finding T' such that $T'(c) = T(c)$ for the target canonical input c, which is computationally infeasible when c is pseudorandom and the transformation space is exponential in k.

3.2 Encoding Modes

PMA distinguishes two encoding modes with distinct algebraic properties:

Storage Encoding (Non-Invertible). A storage transformation T_s is any function satisfying $T_s(c) = d$. Invertibility is not required. Multiple transformations may encode the same data. This mode is optimized for space efficiency.

Computation Encoding (Invertible). A computation transformation T_c is a bijection satisfying $T_c^{-1} \circ T_c = T_c \circ T_c^{-1} = I$. Computation transformations form a group under composition, enabling logical reversibility, energy-efficient implementation on reversible substrates, and the compute-uncompute-verify pattern.

3.3 Categorical Formalization

PMA admits a natural categorical formalization. Define category **PMA** where: objects are data types (the canonical input/output spaces); morphisms are stored transformations; composition of morphisms is both computation and storage update; and identity morphisms represent null transformations (data equal to its canonical representation). The Yoneda perspective applied to **PMA** conceptually motivates the insight that data values can be characterized by the transformations that produce them—providing categorical grounding for PMA's core principle that information is transformation. (We note that the formal Yoneda lemma characterizes an object by the totality of morphisms from all other objects; PMA's application is analogical rather than a direct instantiation, as each datum is characterized by a specific transformation from the canonical input rather than by all possible morphisms.)

The categorical typing discipline ensures composability: for transformations to compose, the output type of one must match the input type of the next. In a uniform PMA system where all transformations operate over the same canonical space $GF(2^k)^n$, composition is direct. In heterogeneous systems where cognitive functions operate over different data types, adaptor morphisms (type-converting transformations) mediate between stages, preserving the audit chain across type boundaries.

3.4 The Five Emergent Properties

We now show that the five architectural properties claimed in Section 1 follow from the definitions above.

Theorem 3.5 (Structural Auditability). *Every computational result in a PMA system carries the complete chain of transformations that produced it, provided the system retains the ordered sequence of constituent transformations (composition history) alongside the composed result. No external audit infrastructure is required. Proof.* A result d is produced by applying $T_{result} = T_n \circ \ldots \circ T_1$ to canonical input c. The composed transformation T_{result} encodes the full composition history when the ordered factor sequence $\{T_1, \ldots, T_n\}$ is retained as architectural metadata. Note that the composed transformation T_{result} alone does not admit unique factorization; PMA systems therefore retain the ordered sequence of constituent transformations to ensure full auditability. The result cannot exist without the transformation chain, because the result *is* the transformation chain applied to c. □

Remark. The emergent property is not that PMA automatically retains history, but that computation and audit trail are structurally identical: the composition chain serves simultaneously as the computation and its own provenance record. In conventional architectures, audit infrastructure must be engineered separately and kept consistent with computation through explicit synchronization. In PMA, the only architectural decision is whether to retain or discard the factor sequence after composition; no separate audit mechanism, synchronization protocol, or consistency guarantee is required.

Theorem 3.6 (Enforced Constraints). *Required transformations can be made necessary components of valid computation chains. Proof.* Define a validity predicate V(T) that requires specific transformations $\{T_{req,1}, \ldots, T_{req,k}\}$ to appear as factors in the composition. A chain $T = T_n \circ \ldots \circ T_1$ is valid only if $\forall i \in \{1,\ldots,k\}$: $T_{req,i}$ appears as a factor. Invalid chains produce no recognized output. Constraint compliance is verified by checking the transformation structure, not by inspecting behavior. □

Theorem 3.7 (Reversibility and Verification). *For invertible transformations, the compute-uncompute-verify pattern provides complete mathematical verification of computational integrity. Proof.* Given result $d = T(c)$ where $T = T_n \circ \ldots \circ T_1$ and each T_i is invertible, compute $T^{-1}(d) = T_1^{-1} \circ \ldots \circ T_n^{-1}(d)$. If $T^{-1}(d) = c$, the computation is verified correct. If $T^{-1}(d) \neq c$, an error or tampering has occurred. This verification is complete (not statistical) and constructive (it identifies the exact failure point). □

4 Finite-Precision Implementation over Galois Fields

The formal framework of Section 3 is specified over abstract algebraic structures. For hardware implementation, we require finite-precision arithmetic with guaranteed properties. We specify PMA operations over Galois fields $GF(2^k)$ where $k \in \{8, 16, 32, 64, 128, 256\}$.

Notation. Throughout this section, $\oplus$ denotes addition in $GF(2^k)$ (bitwise XOR), $\cdot$ denotes multiplication in $GF(2^k)$ (polynomial multiplication modulo the irreducible polynomial), and / denotes field division (multiplication by the multiplicative inverse). We avoid the tensor product symbol $\otimes$ for field multiplication to prevent confusion with its distinct meaning in the categorical formalization of Section 3.3.

4.1 Exact Arithmetic Guarantee

Theorem 4.1 (Exact Field Arithmetic). *All arithmetic operations in $GF(2^k)$ are exact.* For any $a, b \in GF(2^k)$: addition $a \oplus b$ is exact (bitwise XOR); multiplication $a \cdot b$ is exact (polynomial multiplication mod $p(x)$); inversion a^{-1} is exact for $a \neq 0$ (extended Euclidean algorithm); and division $a / b = a \cdot b^{-1}$ is exact for $b \neq 0$. No rounding, truncation, or overflow occurs.

Standard irreducible polynomials include: $GF(2^8)$: $p(x) = x^8 + x^4 + x^3 + x + 1$ (the AES field); $GF(2^{128})$: $p(x) = x^{128} + x^7 + x^2 + x + 1$ (the GCM field).

4.2 Affine Encoding over $GF(2^k)$

Definition 4.2 (Affine Encoding). For data vector $d \in GF(2^k)^m$ and canonical input $c \in GF(2^k)^n$ with $m \leq n$, the affine encoding is the pair (M, b) where $M \in GF(2^k)^{m \times n}$ and $b \in GF(2^k)^m$ such that $M \cdot c \oplus b = d$.

We provide a sparse construction that achieves $O(m)$ storage rather than $O(mn)$. The algorithm selects a single nonzero component $c[j]$ of the canonical input (guaranteed to exist by the ZeroReject mechanism) and computes scaling factors $S[i] = d'[i] / c[j]$ for each data component, where $d' = d \oplus b$.

Theorem 4.3 (Round-Trip Exactness). *For sparse encoding (j, S, b) constructed by the sparse affine algorithm and canonical input c with $c[j] \neq 0$: Decode(Encode(d)) = d for all components.*
Proof. By construction, $S[i] = (d[i] \oplus b[i]) / c[j]$. Decoding computes $S[i] \cdot c[j] \oplus b[i] = (d[i] \oplus b[i]) \oplus b[i] = d[i]$. Division by $c[j]$ is well-defined since $c[j] \neq 0$ by ZeroReject. All operations are exact in $GF(2^k)$. □

4.3 Constructive Invertible Encoding

For computation-mode encoding, we require invertible bijections. We provide a deterministic basis-mapping construction.

Algorithm 4.4 (Basis Extension Method). Given source vector c and target vector d, both nonzero in $GF(2^k)^n$: (1) Extend c to a basis B_c by scanning standard basis vectors and testing linear independence. (2) Extend d to a basis B_d by the same deterministic scan. (3) Construct M as the unique linear map sending B_c to B_d. (4) Compute $M = [d \mid w_2 \mid \ldots \mid w_n] \cdot [c \mid v_2 \mid \ldots \mid v_n]^{-1}$.

Theorem 4.5 (Guaranteed Invertibility). *Algorithm 4.4 always produces an invertible matrix $M \in GL(n, GF(2^k))$ satisfying $M \cdot c = d$, without retry or perturbation. Proof.* M is a change-of-basis transformation between two valid bases. Both B_c and B_d span $GF(2^k)^n$ by construction. A linear map between bases is always invertible. □

Note: Algorithm 4.4 requires both c and d to be nonzero. For the edge case where d = 0 (encoding a zero vector), the method extends naturally to affine bijections $T(x) = Mx \oplus b$ with M invertible, where b absorbs the offset. Since XOR with a fixed vector is itself a bijection, the affine extension preserves invertibility and round-trip exactness while covering the full output space.

Worked Example: Encoding, Composition, and Verification

We illustrate the sparse affine encoding with a concrete numerical example in $GF(2^8)$ using the AES irreducible polynomial $p(x) = x^8 + x^4 + x^3 + x + 1$.

Example 4.6 (Scalar Encoding in $GF(2^8)$). Let d = 0x2A (decimal 42) be the data value to store. Let c = 0xA3 be the canonical input (produced by SHAKE256 and ZeroReject), and let b = 0x7F be the bias vector.

Step 1 — Bias removal: $d' = d \oplus b = 0x2A \oplus 0x7F = 0x55$.

Step 2 — Scaling factor: $S = d' \cdot c^{-1}$ in $GF(2^8)$. Since c = 0xA3, we compute $c^{-1} = 0xC3$ (verifiable: $0xA3 \cdot 0xC3 = 0x01$ in $GF(2^8)$). Thus $S = 0x55 \cdot 0xC3 = 0x40$.

Step 3 — Store transformation: The transformation register stores T = (j = 0, S = 0x40, b = 0x7F). The original data value 42 is not stored; only the transformation that produces it is stored.

Step 4 — Decode (observation): $S \cdot c \oplus b = 0x40 \cdot 0xA3 \oplus 0x7F = 0x55 \oplus 0x7F = 0x2A =$ 42. The original data is recovered exactly, with no floating-point approximation — all operations are exact in $GF(2^8)$.

Composition. Now let a second transformation T_2 encode $d_2 = 0x13$ (decimal 19) with bias $b_2 =$ 0x5E, yielding $S_2 = (0x13 \oplus 0x5E) \cdot 0xC3 = 0x4D \cdot 0xC3 = 0x86$. Composing $T_1 \circ T_2$ produces a new affine transformation with $S_{composed} = S_1 \cdot S_2 = 0x40 \cdot 0x86 = 0xD6$ and $b_{composed} = S_1 \cdot b_2 \oplus b_1 = 0x6A \oplus 0x7F = 0x15$.

Verification of composition. Applying T_composed to c yields $0xD6 \cdot 0xA3 \oplus 0x15 = 0xC6 \oplus$ $0x15 = 0xD3$. Applying $T_1 \circ T_2$ stepwise — first $T_2(c) = 0x86 \cdot 0xA3 \oplus 0x5E = 0x4D \oplus 0x5E =$ 0x13, then $T_1(0x13) = 0x40 \cdot 0x13 \oplus 0x7F = 0xAC \oplus 0x7F = 0xD3$ — produces the identical result. The composition is exact.

Audit trail. The chain $[T_1(S = 0x40, b = 0x7F), T_2(S = 0x86, b = 0x5E)]$ records the ordered factor sequence. Each factor is individually inspectable (Theorem 3.5), and the composed transformation cannot have been produced without both factors being present. This is the structural auditability property: the chain's existence constitutes proof that the required computational steps occurred.

5 Thermodynamic Analysis

PMA's transformation-based computation is naturally aligned with reversible computing because transformation composition preserves information—no data is overwritten. We analyze energy dissipation under two design scenarios.

5.1 Local-Node Ideal Analysis

Under idealized local conditions (node capacitance $C \leq 0.1$ fF, adiabatic clocking with $T_{clk} \geq 1000 \times RC$, operating voltage $V \leq 0.3$ V, leakage power ≤ 1 pW per gate), a single PMA switching event achieves:

$$E_{switching} = (RC/T_{clk}) \times \tfrac{1}{2}CV^2 \approx 4.5 \times 10^{-21} \text{ J}$$

This yields a ratio of approximately 1.6× the Landauer limit ($kT \ln 2 \approx 2.87 \times 10^{-21}$ J at 300 K), demonstrating that PMA switching events can approach fundamental thermodynamic limits.

5.2 System-Level Conservative Analysis

Under realistic system conditions (effective capacitance $C \leq 1$ fF including wiring, $T_{clk} \geq 100 \times RC$, operating voltage $V \leq 0.9$ V, 10 nodes per logical operation, energy recovery $\geq 90\%$), the derivation proceeds as follows. Each node cycles energy $\tfrac{1}{2}CV^2 = \tfrac{1}{2} \times (1 \times 10^{-15}) \times (0.9)^2 = 4.05 \times 10^{-16}$ J. A logical operation spanning 10 nodes cycles a total of 4.05×10^{-15} J. With 90% energy recovery (i.e., 10% dissipated), the net energy dissipated per logical operation is $0.10 \times 4.05 \times 10^{-15} = 4.05 \times 10^{-16}$ J, yielding ~141,000× the Landauer limit ($kT \ln 2 \approx 2.87 \times 10^{-21}$ J at 300 K). Note that the adiabatic clocking condition ($T_{clk} \geq 100 \times RC$) is the regime that enables the stated recovery efficiency; these are not independent multiplicative factors. This compares favorably with conventional CMOS at ~3×10^{-15} J per operation, representing approximately 7× improvement. The dominant factor is recovery efficiency—achieving >99% recovery would reduce PMA energy by an additional order of magnitude.

Engineering-Conditional Result 5.1. *PMA logical operations achieve energy dissipation within 10^5–10^6× of the Landauer limit at the system level, and within 2× (approximately 1.6×) at the local-node level, given the conditions stated above.* The energy–delay tradeoff is explicit: voltage has quadratic impact on energy, while adiabaticity ratio trades linearly between dissipation and latency.

6 Application to AGI Safety Architecture

The five emergent properties of PMA collectively address a central challenge in AGI safety: how to provide mathematical guarantees about the behavior of systems more capable than human oversight can fully comprehend. We describe an integration architecture where PMA serves as the computational substrate for an AGI safety framework, enabling safety properties that are structural rather than trained.

6.1 Cognitive Functions as Transformations

In the integrated architecture, each AGI cognitive function maps to a PMA transformation: perception (encoding external inputs into the transformation domain), context assessment (integrating perceived inputs with historical context), ethical constraint evaluation (applying safety boundaries), authority verification (confirming operational authorization), reasoning (domain-specific inference), and decision synthesis (producing actionable outputs with confidence scores).

The complete reasoning chain is the composition: $T_{synthesis} \circ T_{reasoning} \circ T_{authority} \circ T_{ethics} \circ T_{context} \circ T_{perception}$. Critically, T_{ethics} and $T_{authority}$ are designated as **structurally required** transformations. A reasoning chain that does not include these transformations is invalid by construction, regardless of its output.

6.2 Safety by Construction versus Safety by Constraint

The distinction between PMA-based safety and conventional constraint-based safety is fundamental. In conventional architectures, safety constraints are policies: rules the system is instructed to follow. Compliance depends on correct implementation, proper configuration, and absence of exploits. An AGI might identify that constraint C is enforced by software module M, modify M to disable C, and operate without the constraint.

In a PMA architecture, safety constraints are algebraic. The constraint is embedded in the transformation algebra itself. An AGI attempting to compose transformations that violate the constraint would find the composition *undefined*—analogous to a type error in a strongly-typed language, where the invalid expression cannot be compiled rather than failing at runtime. No state change occurs. The constraint violation is not detected after the fact; it cannot be *formulated* within the computational substrate.

Proposition 6.1 (Safety as Definitional Property). *In a PMA system with structurally required ethical constraint transformations, behaviors that do not include the specified constraint evaluations are computationally undefined rather than prohibited, conditional on the correctness and completeness of the constraint specification.* This transforms the enforcement aspect of AGI safety from an alignment problem (ensuring the system wants to be safe) to a computational architecture problem (ensuring the system cannot express reasoning that omits required constraint evaluations). We note that PMA addresses the enforcement of safety constraints but not their specification; whether a given set of constraints captures all relevant safety properties remains an open research question (see Section 9).

Failure Mode Analysis. When an attempted computation omits a structurally required transformation, the composition operator produces no valid output—the chain fails to compose, analogous to a type error in a strongly-typed language rather than a runtime exception. The system does not execute the computation and then check the result against a policy; the computation cannot be formulated without the required components. The failure is detectable (the composition

operator reports which required transformation is absent), deterministic (the same invalid chain always fails identically), and non-bypassable (there is no lower-level interface at which the composition could be forced), provided the composition operator itself is implemented correctly—a standard trusted computing base assumption shared with all hardware-enforced security mechanisms such as memory management units and trusted platform modules.

6.3 Verification of AGI Reasoning

PMA enables post-hoc verification of any AGI decision through the compute-uncompute-verify pattern. Given a decision output d, a reviewer can: (1) retrieve the complete transformation chain $T = T_n \circ \ldots \circ T_1$; (2) verify that required transformations (T_{ethics}, $T_{authority}$) are present in the chain; (3) inspect any individual transformation to understand that specific reasoning step; (4) reverse the entire chain to verify mathematical integrity. The reviewer does not receive a generated explanation of the decision. They receive the *actual reasoning chain* in a form that can be mathematically audited.

7 Applications

7.1 Financial Systems Auditing

Financial institutions currently process millions of transactions through algorithms that produce outputs with separate audit logs. Regulators review logs. Auditors sample records. But the computational reasoning—why a specific trade was executed, how a credit limit was calculated, what risk factors were considered—exists only in ephemeral processing states that are discarded after execution. PMA enables financial computations where the audit trail is the computation itself: every credit decision, risk calculation, and trading algorithm execution carries provable, complete reasoning chains that regulators can mathematically verify without relying on separately maintained logs.

7.2 Medical AI Verification

Diagnostic AI systems increasingly influence treatment decisions, but when a patient asks why a particular treatment was recommended, neither the physician nor the algorithm can provide the actual reasoning chain. PMA-based medical AI would preserve the complete transformation chain from patient data input through diagnostic reasoning to treatment recommendation. Each reasoning step—symptom weighting, differential diagnosis, treatment selection—would be individually inspectable and mathematically verifiable. Regulatory bodies could audit not just outcomes but actual reasoning processes.

7.3 Autonomous Systems Governance

Autonomous military systems, self-driving vehicles, and robotic systems making real-time decisions present acute verification challenges. PMA provides a framework where autonomous decisions carry provable reasoning chains with structurally enforced authority limits and ethical constraints. A human reviewer asking "Why did the system take this action?" receives not a post-hoc explanation but the complete, verifiable, reversible chain of reasoning that can be mathematically audited for constraint compliance.

8 Comparison with Alternative Verification Paradigms

Property	**Blockchain**	**Zero-Knowledge Proofs**	**Mech. Interpretability**	**PMA**
Verifies	Events occurred	Computation correct	Internal features	Reasoning chain
Granularity	Transactions	Circuit-level	Neuron/feature-level	Transformation-level
Overhead	High (consensus)	High (proof gen.)	High (analysis)	None (intrinsic)
Post-hoc?	Yes	Yes	Yes	No (structural)
Reasoning visible?	No	No (by design)	Partial	Complete
Constraint enforcement	Contract-based	Not applicable	Not applicable	Structural
Reversibility	No	No	No	Yes (invertible mode)
AGI safety application	Audit trail	Proof of behavior	Understanding internals	Safety by construction
Overhead per operation	Consensus (seconds)	Proof generation (100–1000×)	Interpretation (variable)	O(n) perm.; O(n^2–n^3) matrix

The fundamental distinction is that blockchain, zero-knowledge proofs, and mechanistic interpretability are all applied *after or alongside* computation as separate verification processes. PMA makes verification *intrinsic to* computation. There is no separate verification step because verification is intrinsic to the computational representation.

9 Limitations and Open Problems

Hardware Readiness. PMA's full energy-efficiency benefits require reversible logic substrates that are not yet commercially available. While software emulation and FPGA prototyping are feasible today, the thermodynamic advantages require purpose-built adiabatic circuits.

Commercial reversible silicon is projected for 2027–2028 [15], but this timeline carries uncertainty.

Encoding Overhead. Representing information as transformations introduces storage overhead relative to raw data storage. The permutation-based reference embodiment stores a full 2048-bit permutation to encode a single 8-bit datum—a 256:1 overhead ratio. The sparse affine encoding reduces this to O(m) for m-dimensional data, yielding overhead proportional to the data dimension rather than the field size, but overhead remains significant for storage-dominated workloads.

Workload Suitability. PMA does not reduce data movement for all workloads. Applications dominated by single-shot reads, requiring frequent external data exchange, or with transformation representations exceeding available near-memory storage may not benefit from PMA. The architecture is most advantageous for composition-heavy workloads where many transformations compose before observation.

Controller Bootstrap. Pure PMA requires an external controller for conditional branching (the controller may itself be implemented in PMA, but initial execution requires conventional control flow). Turing completeness requires scalable state via chaining multiple transformation registers or external storage.

AGI Integration Gap. The integration of PMA with neural network-based AGI systems remains an open research problem. Current large language models are not built on transformation-based substrates, and retrofitting PMA properties onto existing architectures is non-trivial. The most likely integration path is a hybrid architecture where PMA accelerators handle safety-critical reasoning while conventional processors handle other workloads.

Formal Verification. While we provide proofs for the core theorems, a complete formal verification of PMA's security properties (e.g., the computational infeasibility of forging transformation chains with cryptographically strong canonical inputs) requires additional work in computational complexity theory.

Empirical Validation. This paper presents PMA as a formal foundations contribution. Software reference implementations, cycle-accurate simulations, and empirical performance benchmarks are planned for subsequent publications and will be necessary to validate the theoretical claims in practical settings.

Transformation Chain Growth. In long-lived processes, the transformation chain that constitutes the audit trail grows linearly with the number of computational steps. For continuously operating systems such as an AGI agent performing extended reasoning, chain length management becomes a practical concern. Strategies including periodic checkpointing—where the current output serves as a new canonical input to begin a fresh chain, with the prior chain's integrity preserved via cryptographic binding (e.g., c_new = ZeroReject(SHAKE256(terminal_state ‖ H(chain) ‖ context)))—and hierarchical chain compression require further investigation to balance auditability against resource constraints.

10 Conclusion

We have presented Processual Memory Architecture, a computational framework that dissolves the ontological distinction between data storage and computation by representing information as transformation functions. This conceptual inversion—drawing from category theory and process philosophy—yields five emergent architectural properties (structural auditability, transparent reasoning, enforced constraints, tamper evidence, and reversibility) that collectively enable verifiable computation: systems that can mathematically verify the integrity and correctness of their own reasoning.

We have provided a complete mathematical specification over Galois fields with exactness guarantees, constructive algorithms for both encoding modes, thermodynamic analysis demonstrating feasibility on reversible substrates, and an integration architecture for AGI safety where constraints are structural rather than advisory.

The implications for AGI safety are particularly significant. Current approaches to alignment—training, testing, interpretability—operate within the existing computational paradigm, attempting to understand or constrain systems after they are built. PMA offers a complementary approach: designing the computational substrate so that safety is definitional. An AGI built on PMA does not promise to follow ethical constraints—it can verify it did. It does not claim its reasoning was sound—the reasoning chain is mathematically verifiable. It does not assert it operated within authorized boundaries—the authority verification transformation is a structural component of valid computation.

PMA does not solve all problems of AGI safety. It does not guarantee that values are correctly specified, that capabilities are beneficial, or that deployment is wise. What it provides is a foundation on which safety guarantees can be *built and verified* rather than assumed and trusted. In a world where AI systems increasingly make consequential autonomous decisions, the ability to mathematically verify that reasoning was correct, complete, and uncorrupted may prove not merely useful but essential.

References

[1] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, vol. 21, no. 8, pp. 613–641, 1978.

[2] S. Mac Lane, Categories for the Working Mathematician, 2nd ed. Springer, 1998.

[3] S. Awodey, Category Theory, 2nd ed. Oxford University Press, 2010.

[4] A. N. Whitehead, Process and Reality, Corrected ed., D. R. Griffin and D. W. Sherburne, Eds. Free Press, 1978.

[5] Y. Bai et al., "Constitutional AI: Harmlessness from AI Feedback," arXiv:2212.08073, 2022.

[6] N. Nanda et al., "Progress Measures for Grokking via Mechanistic Interpretability," arXiv:2301.05217, 2023.

[7] S. Ghose et al., “Processing-in-Memory: A Workload-Driven Perspective,” IBM Journal of R&D, vol. 63, no. 6, pp. 3:1–3:19, 2019.

[8] A. Sebastian et al., "Memory Devices and Applications for In-Memory Computing," Nature Nanotechnology, vol. 15, pp. 529–544, 2020.

[9] M. Davies et al., "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," IEEE Micro, vol. 38, no. 1, pp. 82–99, 2018.

[10] S. Furber et al., "The SpiNNaker Project," Proceedings of the IEEE, vol. 102, no. 5, pp. 652–665, 2014.

[11] R. Landauer, "Irreversibility and Heat Generation in the Computing Process," IBM Journal of R&D, vol. 5, no. 3, pp. 183–191, 1961.

[12] C. H. Bennett, "Logical Reversibility of Computation," IBM Journal of R&D, vol. 17, no. 6, pp. 525–532, 1973.

[13] T. Toffoli, "Reversible Computing," in Automata, Languages and Programming, Springer, 1980, pp. 632–644.

[14] E. Fredkin and T. Toffoli, "Conservative Logic," International Journal of Theoretical Physics, vol. 21, pp. 219–253, 1982.

[15] Vaire Computing, “Ice River: A Reversible Computing Test Chip in 22nm CMOS,” tech. demonstration, March 2025. Reported in: S. Ward-Foxton, “Vaire Demos Energy Recovery with Reversible Computing Test Chip,” EE Times, September 2025. Available: https://vaire.co

[16] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[17] S. Goldwasser, S. Micali, and C. Rackoff, "The Knowledge Complexity of Interactive Proof Systems," SIAM Journal on Computing, vol. 18, no. 1, pp. 186–208, 1989.

[18] E. Ben-Sasson et al., "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture," in USENIX Security, 2014.

[19] R. Gennaro, C. Gentry, and B. Parno, "Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers," in CRYPTO, 2010.

[20] C. Olah et al., “Zoom In: An Introduction to Circuits,” Distill, 2020.

[21] T. Bricken et al., “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning,” Transformer Circuits Thread, Anthropic, 2023.

[22] J. Leike et al., "Scalable Agent Alignment via Reward Modeling: A Research Direction," arXiv:1811.07871, 2018.

[23] B. Pierce, Basic Category Theory for Computer Scientists. MIT Press, 1991.

[24] D. Spivak, “Functorial Data Migration,” Information and Computation, vol. 217, pp. 31–51, 2012.

[25] B. Coecke and A. Kissinger, Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning. Cambridge University Press, 2017.

[26] J. von Neumann, “First Draft of a Report on the EDVAC,” Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[27] P. Buneman, S. Khanna, and W. C. Tan, "Why and Where: A Characterization of Data Provenance," in International Conference on Database Theory (ICDT), Springer, 2001, pp. 316–330.

[28] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance Semirings," in Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), 2007, pp. 31–40.

[29] D. R. Ghica and A. Jung, "Categorical Semantics of Digital Circuits," in Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design (FMCAD), 2016, pp. 41–48.

[30] J. C. Baez and B. Fong, "A Compositional Framework for Passive Linear Networks," Theory and Applications of Categories, vol. 33, no. 38, pp. 1158–1222, 2018.

Appendix A: Reference Embodiment — Permutation-Based PMA

This appendix specifies a complete reference implementation using permutation encoding over $GF(2^8)$ (the set {0, 1, …, 255}), suitable for cryptographic and database applications.

Storage Format. A permutation π on {0, 1, …, 255} is stored as a contiguous 2048-bit (256-byte) block: PERM[256 × 8 bits] = {π(0)[7:0], π(1)[7:0], …, π(255)[7:0]}. Byte order is little-endian within each 8-bit element.

Read (Transformation Application). Given permutation storage PERM and canonical input c ∈ {0..255}: return PERM[c]. Complexity: O(1).

Composition. Given permutations $PERM_1$, $PERM_2$: for i = 0 to 255, $PERM_3$[i] = $PERM_1$[$PERM_2$[i]]. Complexity: O(n).

Inversion. Given PERM: for i = 0 to 255, PERM_INV[PERM[i]] = i. Complexity: O(n).

Distinction from Lookup Tables. A conventional lookup table stores data at addresses. PMA stores transformations that produce data when applied to canonical inputs. The differences: (1) PMA's canonical-input-as-address chain provides intrinsic audit capability; (2) transformations compose before observation, enabling deferred evaluation at the substrate level—unlike lazy evaluation in functional languages, which implements deferral as a software abstraction atop conventional hardware, PMA’s deferral is algebraic: composed transformations form a single composite that is never materialized until observation, with no intermediate storage or thunk management; (3) composition creates auditable lineage; (4) invertible transformations enable compute-uncompute-verify; (5) the transformation chain preserves complete history, unlike LUT writes which overwrite.

Appendix B: Patent Status and Intellectual Property

Aspects of the broader research program within which this work was developed are the subject of pending U.S. patent applications assigned to Luminareware LLC. The Processual Memory Architecture framework described in this paper is published to establish scientific priority, invite peer review, and foster collaboration in the verifiable computation and AGI safety research communities. Licensing inquiries may be directed to the author.