The Magarshak Machine:
A Stream-Partitioned Model for Governed State Evolution

The SPACER Framework: Streams, Policy, Actions, Capabilities, Execution, and Relations in Reactive Distributed Systems

Gregory Magarshak
Qbix, Inc. & Intercoin, Inc.
IE New York City
gmagarshak@faculty.ienyc.ie.edu

Version 4.0 April 12, 2026

Abstract

Classical models of computation—the Turing Machine and the von Neumann architecture—formalize computation as transformations over centralized or implicitly shared state. They do not natively encode distributed ownership, append-only histories, explicit side effects, or policy-governed interactions, and so provide poor foundational abstractions for the distributed, reactive, multi-actor systems that constitute modern software infrastructure.

We introduce the *Magarshak Machine* ($\mathcal{MM}$), a formal computational model whose six components form the **SPACER** framework: **S**treams, **P**olicy, **A**ctions, **C**apabilities, **E**xecution, and **R**elations. Streams are governance and metadata layers: their messages may carry content-address hashes of underlying data that may be encrypted or stored off-chain, so that policy enforcement operates on verifiable metadata without requiring access to opaque content. Relations between streams are recorded in two bidirectional index tables—*relatedTo* and *relatedFrom*—updated atomically when streams post `relateTo`/`unrelateTo` messages, generalising vector clocks to dynamic, typed, weighted, retractable stream graphs while keeping relation data separate from stream content. Actors are cryptographic identities.

Computation is expressed as *Actions* governed by a formal five-phase execution calculus. VALIDATE performs structural pre-screening with no I/O. COMPUTE assembles all needed data including remote fetches via side-effect-free view protocols (HTTP GET, IMAP, file reads). REQUIRE enforces governance policy with retry-loop semantics, forming a negotiation cycle that terminates when the Policy accepts or is abandoned. EXECUTE applies effects exclusively to locally owned streams. CALL emits events to named subscriber streams, propagating through the distributed system entirely via the subscription mechanism—no direct cross-publisher writes.

We introduce *ripple deduplication*: hash-based ripple identifiers carried on all events in a causal chain, deduplicated against a bounded local log at each publisher, preventing event storms in cyclic or multi-path subscription graphs without any global coordination. Pre-set *interaction rails*—the subscription and ACL topology established in advance under governance—bound the rates of value and information flow across the system.

From these constructions we derive substantive theorems. The central structural result establishes *embarrassing parallelism*: any set of actions whose REQUIRE-declared write sets and read sets are mutually disjoint may execute simultaneously on independent nodes without coordination, and this disjointness is statically decidable at enqueue time. Parallelism scales linearly with the number of distinct publishers. We also prove a precise CAP classification (AP per-publisher, optional CP per action), causal consistency, minimal cache invalidation, and deterministic replay. We establish the formal connection to the Probabilistic Language Trie framework, importing quantitative caching advantage theorems. We further formalize AI inference calls (LLMs, diffusion models) as ε-*deterministic view capabilities* within COMPUTE, proving that content-addressable memoization makes cache-hit invocations exactly deterministic, and deriving a *Probabilistic Consensus* result: two nodes independently executing the same action agree on its effect with probability $(1-\varepsilon)^k$, enabling fork detection and coordination-free consensus for all but a $k\varepsilon$ fraction of invocations.

The Magarshak Machine defines *how distributed reactive systems evolve safely, transparently, and composably over time*—in parallel, without global consensus, with governance over metadata even when content is opaque, and with bounded, locally-deduplicated event propagation.

Contents

1 Introduction

1.1 Motivation

Five structural properties of modern distributed systems are absent from classical computational formalisms:

(i) **Distributed ownership.** State is produced and consumed by mutually distrusting actors across administrative domains and legal jurisdictions.

(ii) **Append-only persistence.** Reliable systems favor immutable, append-only logs because they are trivially replicated, auditable, and safe under partial failure.

(iii) **Policy-governed interaction.** In finance, healthcare, and federated platforms, every state transition must be authorized and auditable.

(iv) **Explicit side effects.** Network I/O, cryptographic operations, and external storage writes have observable consequences beyond state mutation; modeling them as implicit is a source of correctness and security failures.

(v) **Reactive event propagation.** State changes drive downstream computations; systems react to changes rather than polling for them.

None of these properties is captured intrinsically by the Turing Machine or von Neumann architecture. They are retrofitted through middleware, databases, message queues, and access-control frameworks, producing accidental complexity and impedance mismatches between the computational model and the deployed reality.

1.2 Our Approach and Contributions

We propose the *Magarshak Machine* ($\mathcal{MM}$), a formal model that elevates all five properties to first-class status through six named components—**Streams**, **Policy**, **Actions**, **Capabilities**, **Execution**, and **Relations**—whose initials form the acronym **SPACER**:

$$\mathcal{MM} \triangleq (\underbrace{S}_{\text{Streams}}, \underbrace{P}_{\text{Policy}}, \underbrace{A}_{\text{Actions}}, \underbrace{C}_{\text{Capabilities}}, \underbrace{E}_{\text{Execution}}, \underbrace{R}_{\text{Relations}})$$

Streams are append-only, publisher-owned state logs that serve as governance and metadata layers over (potentially encrypted) content. **P**olicy governs every state transition, enforced in the REQUIRE phase with retry-loop semantics. **A**ctions are the computation units, structured into the five-phase calculus VALIDATE→COMPUTE→REQUIRE→EXECUTE→CALL. **C**apabilities are the controlled interfaces for external I/O—both side-effect-free view capabilities (used in COMPUTE) and mutating capabilities (used in EXECUTE). **E**xecution is the push-only, reactive event engine: Rule TRIGGER and Rule EXECUTE drive the entire system without polling. **R**elations are stream-to-stream edges stored in two index tables—*relatedTo* and *relatedFrom*. A stream updates its relations by posting `relateTo`/`unrelateTo` messages; the infrastructure applies these atomically to both tables, enabling efficient bidirectional graph traversal and generalizing vector clocks.

Our concrete contributions are:

1. **Unified stream and relation model (§3).** All state is stored in append-only streams. Relations between streams are maintained in two bidirectional index tables (*relatedTo*, *relatedFrom*), updated atomically when streams post `relateTo`/`unrelateTo` messages. The message log provides the auditable event record; the tables provide efficient $O(\log n)$ graph traversal. This generalizes vector clocks to dynamic, typed, weighted, retractable stream graphs.

2. **Formal Actor definition (§3.3).** Actors are cryptographic identities (public keys), enabling formal treatment of delegation and ownership.

3. **Operational semantics (§4).** We define a small-step reduction relation for $\mathcal{MM}$ from which all claimed properties are formally derived.

4. **Phase calculus (§5).** Actions are structured into five phases—VALIDATE → COMPUTE → REQUIRE → EXECUTE → CALL—with precisely stated semantics for each: COMPUTE may fetch remote data via idempotent view protocols; REQUIRE enforces governance policy with retry-loop semantics; EXECUTE writes only to locally owned streams; CALL emits events to named subscriber streams.

5. **Ripple deduplication (§8).** We introduce hash-based ripple identifiers that prevent event storms in cyclic or multi-path subscription graphs, using only local state at each publisher.

6. **Governance on metadata (§9).** We show that $\mathcal{MM}$'s Policy function enforces access control, rate limits, and value caps entirely on stream metadata, without requiring access to (potentially encrypted) stream content. Pre-set interaction rails establish the governance topology in advance, amortizing coordination costs across all future events.

7. **Embarrassing Parallelism theorem (§6).** We prove that $\mathcal{MM}$'s design structurally enables embarrassing parallelism for conflict-free action sets, that disjointness is statically decidable from REQUIRE declarations, and that parallelism scales linearly with the number of publishers (Corollary 6.4).

8. **CAP classification (§13).** We prove that $\mathcal{MM}$ is AP per-publisher and supports optional per-action CP, with the consistency tradeoff scoped to individual capability invocations.

9. **Push-only execution (§7).** We formalize purely reactive (push-based) execution and prove that polling is unnecessary and absent from the model.

10. **Derived theorems (§§10–11).** Minimal cache invalidation, causal consistency without consensus, and deterministic replay are derived from the operational semantics, not asserted as axioms.

11. **AI capabilities and probabilistic consensus (§12).** We formalize AI inference calls (LLMs, diffusion models, etc.) as ε-deterministic view capabilities in COMPUTE, prove that content-addressable memoization converts cache-hit invocations to exactly

deterministic ones, and derive a Probabilistic Consensus theorem showing that fingerprint comparison achieves agreement with probability $(1 - \varepsilon)^k$—making coordination costs proportional to $k\varepsilon$ per invocation rather than constant.

1.3 Paper Organisation

Section 2 reviews related models. Section 3 defines the unified stream and actor model, including the metadata-vs-content separation. Section 4 gives the operational semantics including a worked formal trace (§4.4). Section 5 formalizes the phase calculus. Section 6 proves the structural enabling theorem for embarrassing parallelism. Section 7 develops push-only reactive semantics. Section 8 introduces ripple deduplication. Section 9 addresses governance on metadata and pre-set interaction rails. Sections 10 and 11 derive caching and replay. Section 12 develops AI view capabilities, ε-determinism, probabilistic consensus, and fork detection. Section 13 addresses distributed execution and the CAP classification. Section 14 covers HTTP deployment. Section 15 establishes the connection to Probabilistic Language Tries. Section 16 gives a comparative analysis. Section 17 concludes.

Notation Summary

Symbol	Meaning
$\mathcal{MM} = (S, P, A, C, E, R)$	The Magarshak Machine: SPACER six-tuple
$\mathcal{MM}$	A Magarshak Machine instance
$\mathbb{S}$	Universe of stream identifiers
$\mathcal{S}^T$	Active stream set at time T
S_i	Stream with identifier i; triple $(M_i, Meta_i, ACL_i)$
M_i	Message log of stream i (append-only sequence)
ACL_i	Access-control function of stream i
$\mathbb{M}$	Universe of typed messages
$m = (t, \tau, d)$	Message: timestamp, type, payload
$S_i +\!\!+ m$	Append operation (extends M_i)
$id(a)$	Cryptographic identity (public key) of actor a
$\Sigma = (\mathcal{S}, R, pending)$	Global machine state
α	Action type
$(\alpha, \mathbf{x})$	Action invocation with input $\mathbf{x}$
Validate, Compute, Require, Execute, Call	Five execution phases
$ReadSet(\alpha)$, $WriteSet(\alpha)$	Declared read/write sets from Require
Δ	Delta: set of message sequences to append to outpu
$\Sigma \xrightarrow{e} \Sigma'$	Small-step reduction with label e
G^T	Induced relation graph at time T
$D = (\mathcal{S}, F_D)$	Stream dependency graph (for cache invalidation)
$W = (V_W, E_W)$	Write-conflict graph (for parallelism analysis)
$\mathcal{P}$	Policy function: actor × action × streams → verdic
rid	Ripple identifier: hash of originating event
RL_i	Ripple log of stream i (bounded set of processed rip
ε	Failure probability of an ε-deterministic view capab
$\kappa(mid, i, h)$	Compute cache key: hash of model ID, input, hyper
$\phi_\nu(\alpha, \mathbf{x})$	Compute fingerprint of invocation on node ν
$\delta_p = p_K - p_{K+1}$	Probability gap at cache boundary (PLT caching th

2 Background and Related Work

2.1 The Turing Machine

A Turing Machine [25] is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ operating on a single infinite tape with transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. It defines computability precisely but makes no provision for ownership, concurrency, policy, or observable side effects.

2.2 The von Neumann Architecture

The von Neumann model [27] executes programs over a shared mutable address space. Shared mutable memory is the root cause of data races, cache-coherence costs, and memory-safety vulnerabilities. Mitigation strategies (locks, transactions, immutable data structures) are retrofitted, not intrinsic.

2.3 The Actor Model

The Actor Model [1, 11] is the closest classical precursor to $\mathcal{MM}$: actors communicate via asynchronous message passing and hold private state. However, actor state is mutable; relations between actors are implicit; there is no policy primitive; and execution is not phase-structured.

2.4 Event Sourcing and CQRS

Event sourcing [9, 26] models aggregate state as a sequence of immutable domain events. It is the architectural practice most aligned with $\mathcal{MM}$'s stream model, but it is a pattern, not a formal computational model. It lacks a formal actor identity model, a capability model, a policy function, and a phase-structured execution calculus. The Magarshak Machine provides the formal substrate that event sourcing practitioners implicitly assume.

2.5 Vector Clocks

Vector clocks [8, 14, 21] track causal dependencies between processes by associating each message with a vector of per-process logical timestamps. They answer the question: *did event e happen before event f?* [14] The $\mathcal{MM}$ relation model generalizes this: by recording, for each message posted to a stream, which other streams it is related to (in the *relatedTo* and *relatedFrom* tables), the causal graph between any two streams is queryable without scanning full message logs.

2.6 CRDTs

Conflict-free Replicated Data Types [24] achieve eventual consistency by requiring operations to be commutative and idempotent. CRDTs are a special case of $\mathcal{MM}$'s stream model: a CRDT state is a stream whose append operation satisfies the CRDT merge laws. $\mathcal{MM}$ does not require this of all streams; it is an optional specialization.

2.7 Blockchains

Permissionless blockchains [4, 23] achieve global consensus on an append-only ledger via proof-of-work or proof-of-stake. The append-only commitment is shared with $\mathcal{MM}$, but the requirement for global replication and total ordering is not. $\mathcal{MM}$ replaces global consensus with per-stream sequential consistency and per-publisher ownership, achieving auditability without the throughput and latency costs of consensus. We formalize this comparison in §6.

2.8 Internet Computer Protocol (ICP) Canisters

ICP canisters [7] are WebAssembly modules with orthogonal persistence, message queues, and a global execution model enforced by subnet consensus. Like $\mathcal{MM}$, canisters have explicit messaging. Unlike $\mathcal{MM}$, canister state mutations within a subnet require subnet-level consensus, and cross-subnet calls involve asynchronous inter-canister messaging with rollback semantics. We compare throughput and coordination costs in §6.

2.9 Dataflow and Stream Processing

Apache Flink [5] and Apache Kafka [13] and similar dataflow and log-streaming systems operate on unbounded event streams with stateful operators and durable message logs. $\mathcal{MM}$ generalizes these by reifying ownership, actor identity, policy, and relational structure that such systems treat as infrastructure concerns.

2.10 Probabilistic Language Tries

Magarshak [20] introduces *probabilistic language tries* (PLTs): rooted prefix trees in which each outgoing edge carries the conditional probability of the corresponding token or action under a generative model $\mathcal{M}$. PLTs simultaneously serve as optimal lossless compressors (via frequency-weighted interval encoding), policy representations for sequential decision problems, and memoization indexes for inference reuse. The central theorem of that work establishes that a prior-guided cache initialized from $P_{\mathcal{M}}$ strictly dominates any empirical-frequency cache during the initial phase of a system's operation.

The present paper and the PLT framework are deeply complementary. $\mathcal{MM}$ provides the *governance and ownership model* that PLTs lack: who owns an artifact, what policy governs its creation, and which actors may read it. Conversely, PLTs provide the *quantitative* caching theory that $\mathcal{MM}$'s structural caching theorem (Theorem 10.6) does not: how much better a prior-guided cache is relative to empirical frequency caching, as a function of the distribution's entropy. Section 15 develops these connections formally.

3 Streams, Actors, and Relations

3.1 The SPACER Framework

Definition 3.1 (Magarshak Machine / SPACER)**.** *A* Magarshak Machine *is a six-tuple*

$$\mathcal{MM} \triangleq (S,\ P,\ A,\ C,\ E,\ R)$$

whose components, forming the acronym ***SPACER****, are:*

- S — *a universe of append-only, publisher-owned* Streams *(§3), each a governance and metadata layer that may carry content-address hashes of underlying encrypted or off-chain data;*
- P — *a* Policy *function* $\mathcal{P} : \mathit{Actor} \times \alpha \times \mathbf{x} \times \mathbf{y} \to \{\textsc{allow}, \textsc{deny}(r), \textsc{constrain}(\sigma)\}$ *where* $\alpha \in A$ *is the specific action type being evaluated and* $\mathbf{x}, \mathbf{y}$ *are its input and computed records; enforced in the* Require *phase (§5);*
- A — *a set of* Action *types, each structured into the five-phase calculus* Validate→Compute→Require→Execute→Call *(§5);*
- C — *a set of* Capability *interfaces, partitioned into view capabilities* $\mathcal{C}_{view}$ *(idempotent, used in* Compute*) and mutating capabilities* $\mathcal{C}_{mut}$ *(used in* Execute*);*
- E — *the* Execution *engine: the push-only operational semantics (Rules* Create*,* Trigger*,* Execute*,* Retry*) and the subscription registry mapping streams to subscribed action types (§4); and*
- R — Relations*: two append-only index tables, relatedTo and relatedFrom, that record which messages posted to streams are related to which other streams. Both endpoints of every relation are stored, enabling efficient traversal in either direction (§3.4).*

Remark 3.2. *The ordering* $S \to P \to A \to C \to E \to R$ *reflects a natural dependency chain: the state substrate (S) is governed (P), over which computation (A) operates via controlled interfaces (C), orchestrated by the execution engine (E), with the relational index (R) maintained as a derived, bidirectional view over stream content. Relations are listed last because they are not a primary storage component: they are two index tables populated as a side effect of posting messages, providing efficient graph traversal without altering the core append-only semantics of streams.*

3.2 Streams: The State Substrate (S)

Definition 3.3 (Message)**.** *Let* $\mathbb{M}$ *be a countable universe of* messages*. A message is a triple* $m = (t, \tau, d)$ *where* $t \in \mathbb{N}$ *is a strictly monotone logical timestamp,* τ *is a type drawn from a schema registry* $\mathcal{T}$*, and* d *is a byte payload conforming to* τ*.*

Definition 3.4 (Stream Universe and Active Set)**.** *Let* $\mathbb{S}$ *be a countably infinite universe of stream identifiers. At any global time* T*, the* active stream set $\mathcal{S}^T \subseteq \mathbb{S}$ *is a finite set of instantiated streams. Each stream* $S_i \in \mathcal{S}^T$ *is a triple:*

$$S_i \triangleq (M_i,\ \mathit{Meta}_i,\ \mathit{ACL}_i)$$

where $M_i \in \mathbb{M}^*$ *is a finite, strictly timestamp-ordered sequence of messages,* Meta_i *is a record* (`id`, `owner`, `schema`, `createdAt`)*, and* $\mathit{ACL}_i : \mathit{Actor} \times \{\textsc{read}, \textsc{append}, \textsc{admin}\} \to \mathbb{B}$ *is an access-control function.*

Remark 3.5 (Streams as Metadata Layers)**.** *A stream is a* governance and metadata layer*, not necessarily a content store. The payload field d of each message may contain the actual data, a content-addressed hash $H(data)$ pointing to data stored elsewhere (e.g. an encrypted blob, a file system object, or a distributed store), or a combination of both. This separation is deliberate: $\mathcal{MM}$'s governance mechanisms—access control, rate limiting, policy enforcement, ripple deduplication—operate on the stream's metadata regardless of whether the underlying content is encrypted, hashed, or stored off-chain. A stream therefore provides verifiable, auditable governance over data whose contents may be entirely opaque to the governance infrastructure. The latest content-address hash in M_i acts as a commitment: it allows any party with ACL read access to verify the current state of the underlying data without necessarily being able to decrypt it.*

Stream creation is an operation in the operational semantics (§4), not an implicit assumption. The active set grows monotonically: streams are never deleted.

Definition 3.6 (Append)**.** *The* append *operation extends a stream's message log:*

$$S_i +\!\!+ m \triangleq (M_i \cdot \langle m \rangle,\ Meta_i,\ ACL_i) \ \ \textit{iff}\ \ t(m) > t(\text{last}(M_i))$$

where $\cdot$ denotes sequence concatenation. Append is rejected if the timestamp condition fails. No other operation modifies M_i.

3.3 Actors: Cryptographic Identities Underlying P, A, and C

Definition 3.7 (Actor)**.** *An* actor *is a pair $a = (k_{pub}, k_{priv})$ where k_{pub} is a public key and k_{priv} is the corresponding private key, drawn from an asymmetric key scheme (e.g. Ed25519 or ECDSA secp256k1). The* actor identity *is $id(a) \triangleq k_{pub}$.*

Actor identity is therefore a content-addressed public key. This definition is not merely notational: it directly grounds ownership, delegation, and ACL entries in cryptographic verifiability.

Definition 3.8 (Publisher Ownership)**.** *Stream S_i is* owned *by actor a if $Meta_i.\texttt{owner} = id(a)$. The owner is the unique authority permitted to perform* ADMIN *operations on the stream (including ACL modification).*

Definition 3.9 (Delegation)**.** *Actor a* delegates *append rights on S_i to actor b by appending a signed delegation message $m_\delta = (t, \texttt{delegateAppend}, \{b, expiry, scope\})$ to a designated control stream S_i^{ctrl} owned by a. An actor c evaluating $ACL_i(b, \text{APPEND})$ checks the delegation chain terminating at the owner.*

3.4 Relations: The Relational Index (R)

Relations in $\mathcal{MM}$ are edges between streams. The relation component R maintains two index tables recording which streams are currently related to which other streams. The mechanism by which these tables are updated is the posting of specially-typed messages to streams: a stream S_i posts a `relateTo` message to record that it is now related to

another stream, and an `unrelateTo` message to retract that relation. The message is the auditable, append-only event record; the tables are the live queryable index derived from those messages.

Definition 3.10 (Relation Tables)**.** *The* relation component R *of a Magarshak Machine consists of two index tables:*

- *relatedTo: a set of records* $(S_i,\ S_j,\ \tau,\ w)$ *asserting that stream* S_i *is currently related to stream* S_j *with relation type* $\tau \in \mathcal{T}$ *and weight* $w \in \mathbb{R}$*.*
- *relatedFrom: a set of records* $(S_j,\ S_i,\ \tau,\ w)$ *recording the same relationship from the perspective of the target stream* S_j*, enabling efficient reverse lookup.*

Both tables are updated atomically whenever a relation message is posted. The four message types that drive table updates are:

`relateTo`(j, τ, w)	*posted to* S_i*: insert* (S_i, S_j, τ, w) *into relatedTo and* (S_j, S_i, τ, w) *into relatedFrom*
`relateFrom`(i, τ, w)	*posted to* S_j*: symmetric acknowledgement (usually co-issued)*
`unrelateTo`(j, τ)	*posted to* S_i*: remove all records with matching* (S_i, S_j, τ) *from both tables (regardless of weight* w*)*
`unrelateFrom`(i, τ)	*posted to* S_j*: symmetric retraction*

where τ *is the relation type and* w *is a scalar weight.*

Remark 3.11 (Messages as audit log; tables as live index)**.** *The posted messages are the ground truth: they are appended to the stream's message log and are immutable and auditable like any other message. The relatedTo and relatedFrom tables are a derived index maintained by the infrastructure for efficient graph traversal—a stream can look up all streams it is related to, or all streams related to it, in* $O(\log n)$ *without scanning message logs. The current table state at any time* T *is fully reconstructible by replaying the relation messages up to* T*.*

Definition 3.12 (Induced Relation Graph)**.** *The* induced relation graph *at time* T *is a directed, typed, weighted multigraph* $G^T = (\mathcal{S}^T, Edges^T)$ *where* $(i, j, \tau, w) \in Edges^T$ *iff* $(S_i, S_j, \tau, w) \in relatedTo^T$*.*

3.5 Relation Events as a Generalization of Vector Clocks

Vector clocks [8, 14, 21] track causal dependencies between a fixed set of n processes using n-dimensional timestamp vectors. They answer: *did event* e *causally precede event* f*?*

$\mathcal{MM}$'s relation event model strictly subsumes this.

Proposition 3.13 (Vector Clock Embedding)**.** *Any vector clock system over* n *processes* $P_1, \ldots, P_n$ *can be faithfully* embedded *in a set of* $\mathcal{MM}$ *streams* $S_1, \ldots, S_n$*: there exists an* $\mathcal{MM}$ *configuration that records exactly the causal information a vector clock records, and from which the happened-before relation is recoverable. The embedding is strict:* $\mathcal{MM}$ *admits configurations (dynamic stream sets, typed weighted edges, edge retraction) that have no counterpart in any fixed-*n *vector clock system.*

Proof. Embedding. Map each process P_i to stream S_i. In a vector clock system, P_i maintains $VC_i[1..n]$ where $VC_i[j]$ is the last known logical timestamp of P_j observed by P_i. When P_i sends a message, it appends that message to S_i and, for each $j \neq i$ whose frontier has advanced since the last message, posts `unrelateTo`$(j, \texttt{observed})$ followed immediately by `relateTo`$(j, \texttt{observed}, VC_i[j])$ to S_i. The first message removes the stale entry from both tables; the second inserts the updated frontier weight. The happened-before relation $e \to f$ (event e at P_i happened before event f at P_j) is recoverable from the table state at the time f was posted: $e \to f$ iff the weight in *relatedTo* for entry $(S_j, S_i, \texttt{observed})$ at time $t(f)$ is $\geq t(e)$. This faithfully reproduces the vector clock.

Strictness. A standard vector clock requires a fixed set of n processes at initialization. $\mathcal{MM}$ permits streams to be created dynamically (Rule CREATE), relations to be retracted via `unrelateTo` messages, and relation records to carry arbitrary typed weights. No fixed-n vector clock can represent a dynamic stream set or retractable causal edges. □ □

$\mathcal{MM}$ extends vector clocks in three ways: (i) the set of streams is dynamic, not fixed; (ii) relation records carry typed, weighted payloads rather than scalar timestamps; (iii) retraction (`unrelateTo`) is supported, enabling modeling of transient associations (e.g. session membership, temporary permissions).

4 Operational Semantics

4.1 Global State

Definition 4.1 (Global State)**.** *A* global state Σ *of a Magarshak Machine is a triple* $\Sigma = (\mathcal{S}, R, \mathit{pending})$ *where* $\mathcal{S}$ *is the current active stream set,* $R = (\mathit{relatedTo}, \mathit{relatedFrom})$ *is the current relation table state (Definition 3.10), and pending is a finite multiset of* triggered action invocations *awaiting execution.*

4.2 Actions

Definition 4.2 (Action Type)**.** *An* action type α *is a tuple:*

$$\alpha = (\mathit{name},\ \textsc{Validate}_\alpha,\ \textsc{Compute}_\alpha,\ \textsc{Require}_\alpha,\ \textsc{Execute}_\alpha,\ \textsc{Call}_\alpha)$$

where each phase is a function defined in §5. An action invocation *is a pair* $(\alpha, \mathbf{x})$ *where* $\mathbf{x}$ *is the input record.*

4.3 Small-Step Reduction Rules

We define the small-step reduction relation $\Sigma \xrightarrow{e} \Sigma'$ where e is either a *stream event* (message appended) or an *action invocation*. This follows the tradition of action-based operational semantics [16] in which system behaviour is defined as a sequence of labelled state transitions.

Rule Create. A new stream may be created by its future owner a:

$$\frac{id \notin \mathcal{S} \qquad \mathcal{P}(a,\ \texttt{createStream},\ (id, \tau_s),\ \varnothing) \neq \textsc{deny}}{(\mathcal{S},\ R,\ \mathit{pending}) \xrightarrow{\textsc{create}(a, id, \tau_s)} (\mathcal{S} \cup \{S_{id}\},\ R,\ \mathit{pending})}$$

where S_{id} is initialized with empty log, owner a, and schema τ_s, and (id, τ_s) plays the role of the input record $\mathbf{x}$ while $\varnothing$ indicates no computed record $\mathbf{y}$ is required for stream creation.

Rule Trigger. Appending message m to stream S_i enqueues all actions subscribed to S_i:

$$\frac{m \text{ is well-typed for } S_i \qquad \mathit{ACL}_i(a, \textsc{append}) = \mathit{true} \qquad t(m) > t(\text{last}(M_i))}{(\mathcal{S},\ R,\ \mathit{pending}) \xrightarrow{\textsc{append}(a, i, m)} (\mathcal{S} +\!\!+_i m,\ \mathit{updateR}(R, i, m),\ \mathit{pending} \uplus \mathit{triggers}(i, m))}$$

where $\mathit{triggers}(i, m)$ is the set of action invocations subscribed to stream S_i filtered by message type $\tau(m)$, $\mathcal{S} +\!\!+_i m$ denotes the state with S_i replaced by $S_i +\!\!+ m$, and $\mathit{updateR}(R, i, m)$ updates the relation tables if $\tau(m) \in \{\texttt{relateTo}, \texttt{relateFrom}, \texttt{unrelateTo}, \texttt{unrelateFrom}\}$ (Definition 3.10), leaving R unchanged for all other message types.

Rule Execute. An invocation $(\alpha, \mathbf{x}) \in \mathit{pending}$ executes its five phases in order. Define the intermediate values:

$$\begin{aligned}
\mathbf{y} &= \textsc{Compute}_\alpha(\mathbf{x}, \mathcal{S}) \\
\mathit{decl} &= \textsc{Require}_\alpha(\mathbf{x}, \mathbf{y}, \Sigma_{\mathit{read}}) \quad \text{(proceeds; see Rule } \textsc{Retry} \text{ if rejected)} \\
\Delta &= \textsc{Execute}_\alpha(\mathbf{x}, \mathbf{y}, \mathit{decl}, \mathcal{S}) \\
\mathcal{E}_{\text{out}} &= \textsc{Call}_\alpha(\mathbf{x}, \Delta)
\end{aligned}$$

The reduction rule is then:

$$\frac{(\alpha, \mathbf{x}) \in \mathit{pending} \qquad \mathit{decl} \text{ proceeds} \qquad \Delta,\ \mathcal{E}_{\text{out}} \text{ as above}}{(\mathcal{S},\ R,\ \mathit{pending}) \xrightarrow{(\alpha, \mathbf{x})} (\mathit{applyDeltas}(\mathcal{S}, \Delta),\ R,\ (\mathit{pending} \setminus \{(\alpha, \mathbf{x})\}) \uplus \mathit{triggerAll}(\mathcal{E}_{\text{out}}))}$$

where $\mathit{applyDeltas}(\mathcal{S}, \Delta)$ appends each local delta Δ_i to the corresponding stream S_i, and $\mathit{triggerAll}(\mathcal{E}_{\text{out}})$ applies Rule Trigger to each event in $\mathcal{E}_{\text{out}}$, enqueuing all subscribers to the targeted streams.

Rule Retry. If Require rejects with reason r, the action may be re-entered at Validate with a modified input $\mathbf{x}'$ derived from $(\mathbf{x}, r)$:

$$\frac{(\alpha, \mathbf{x}) \in \mathit{pending} \quad \textsc{Require}_\alpha \text{ rejects with } r \quad \mathbf{x}' = \mathit{negotiate}(\mathbf{x}, r) \quad |\mathit{retries}(\alpha, \mathbf{x})| < N_{\max}}{(\mathcal{S},\ R,\ \mathit{pending}) \xrightarrow{\textsc{retry}(\alpha, r)} (\mathcal{S},\ R,\ (\mathit{pending} \setminus \{(\alpha, \mathbf{x})\}) \uplus \{(\alpha, \mathbf{x}')\})}$$

Here $\mathit{negotiate}(\mathbf{x}, r)$ is an application-supplied function that produces a modified input given the rejection reason r: for example, reducing a transfer amount to satisfy a rate-limit rejection, or selecting an alternative target stream. Its specification is part of the action definition, not the $\mathcal{MM}$ model itself. The predicate $|\mathit{retries}(\alpha, \mathbf{x})| < N_{\max}$ counts how many times the

original invocation $(\alpha, \mathbf{x})$ has already been retried (tracked locally by the executing node); $N_{\max} \in \mathbb{N}$ is a per-action-type retry limit declared in α's definition. If $N_{\max}$ retries are exhausted, the invocation is removed from *pending* and a rejection event is emitted to the action's error stream.

These four rules (CREATE, TRIGGER, EXECUTE, RETRY) constitute the complete operational semantics of $\mathcal{MM}$. All claimed properties are derived from them.

4.4 Worked Formal Trace

We trace a minimal concrete example: Alice posts a message, which triggers a notification to Bob because S_{Bob} is related to S_{Alice} via the *relatedTo* table.

Setup. Let the initial state be $\Sigma_0 = (\mathcal{S}_0, R_0, \emptyset)$ where:

$$\begin{aligned}
\mathcal{S}_0 &= \{S_{\text{Alice}},\ S_{\text{Bob}},\ S_{\text{Notif}},\ S_{\text{Inbox/Bob}}\} \\
S_{\text{Alice}} &= (\langle\rangle,\ \mathit{Meta}_{\mathit{Alice}},\ \mathit{ACL}_{\mathit{Alice}}) \\
S_{\text{Bob}} &= (\langle m_0\rangle,\ \mathit{Meta}_{\mathit{Bob}},\ \mathit{ACL}_{\mathit{Bob}}) \quad \text{where } m_0 = (0,\ \texttt{relateTo},\ (S_{\text{Alice}},\ \texttt{follows},\ 1.0)) \\
S_{\text{Notif}} &= (\langle\rangle,\ \mathit{Meta}_{\mathit{Notif}},\ \mathit{ACL}_{\mathit{Notif}}) \\
S_{\text{Inbox/Bob}} &= (\langle\rangle,\ \mathit{Meta}_{\mathit{Inbox/Bob}},\ \mathit{ACL}_{\mathit{Inbox/Bob}})
\end{aligned}$$

The message m_0 in S_{Bob} is a `relateTo` message posted by Bob to record that his stream is related to Alice's stream. Posting m_0 caused the infrastructure to write the entries:

$$R_0: \quad \mathit{relatedTo} = \{(S_{\text{Bob}},\ S_{\text{Alice}},\ \texttt{follows},\ 1.0)\}, \quad \mathit{relatedFrom} = \{(S_{\text{Alice}},\ S_{\text{Bob}},\ \texttt{follows},\ 1.0)\}$$

The induced relation graph G^0 has a single edge $(S_{\text{Bob}},\ S_{\text{Alice}},\ \texttt{follows},\ 1.0)$. $S_{\text{Inbox/Bob}}$ is Bob's inbox; no actions subscribe to it in this minimal example, so events emitted there produce no further invocations.

Let α_{notify} be an action type subscribed to S_{Alice} on message type `post`, with:

- REQUIRE: $\mathit{ReadSet} = \{S_{\text{Alice}}\},\ \mathit{WriteSet} = \{S_{\text{Notif}}\},\ \mathit{CapSet} = \emptyset$
- EXECUTE: appends a `notification` message to S_{Notif} referencing the triggering post.
- CALL: emits the notification to $S_{\text{Inbox/Bob}}$.

Step 1: Alice posts. Actor Alice appends $m_1 = (1, \texttt{post}, \text{“Hello world”})$ to S_{Alice}. We verify the TRIGGER preconditions: (i) m_1 is well-typed for S_{Alice} (type `post` is in S_{Alice}'s schema); (ii) $\mathit{ACL}_{\mathit{Alice}}(\text{Alice}, \text{APPEND}) = \mathit{true}$ (Alice owns her stream); (iii) $t(m_1) = 1 > 0$, where we adopt the convention $t(\text{last}(\langle\rangle)) = 0$ for an empty log (i.e. any positive timestamp satisfies the ordering condition on an empty stream). All preconditions hold. Applying TRIGGER:

$$\frac{m_1 \text{ well-typed} \quad \mathit{ACL}_{\mathit{Alice}}(\text{Alice}, \text{APPEND}) = \mathit{true} \quad 1 > 0}{(\mathcal{S}_0,\ R_0,\ \emptyset) \xrightarrow{\text{APPEND}(\text{Alice}, \text{Alice}, m_1)} (\mathcal{S}_1,\ R_0,\ \{(\alpha_{\text{notify}}, m_1)\})}$$

where $\mathcal{S}_1 = \mathcal{S}_0$ with S_{Alice} updated to $(\langle m_1\rangle, \mathit{Meta}_{\mathit{Alice}}, \mathit{ACL}_{\mathit{Alice}})$. The invocation $(\alpha_{\text{notify}}, m_1)$ is enqueued because α_{notify} subscribes to S_{Alice} on type `post` and $\tau(m_1) = \texttt{post}$.

Step 2: Notification fires. Apply EXECUTE to $(\alpha_{\text{notify}}, m_1)$. The phase sequence runs:

1. VALIDATE: $m_1.\texttt{type} = \texttt{post}$, m_1 is non-empty. Returns *ok*.
2. COMPUTE: fetches (or uses locally cached) S_{Alice}; derives $\mathbf{y} = \{\texttt{postRef} : m_1,\ \texttt{authorHash} : H(\text{Alice})\}$.
3. REQUIRE: evaluates Policy — no rate limit violation, no ACL conflict. Returns *proceed*($\mathit{WriteSet} = \{S_{\text{Notif}}\}$, $\mathit{CapSet} = \emptyset$).
4. EXECUTE: produces $\Delta = \{S_{\text{Notif}} \mapsto \langle m_2 \rangle\}$ where $m_2 = (2, \texttt{notification}, \{\texttt{from} : \text{Alice}, \texttt{postRef} : m_1,\ \texttt{rippleId} : \mathit{rid}(m_1)\})$. Note: m_2 carries the same ripple ID as m_1, propagating the causal chain.
5. CALL: emits event $\{\texttt{stream} : \texttt{"Inbox/Bob"},\ \texttt{event} : m_2\}$ to Bob's inbox stream. Any actor subscribed to `Inbox/Bob` is triggered by Rule TRIGGER. α_{notify} does not know whether Bob has such subscribers; it only knows which stream to emit to.

Applying EXECUTE (with $\mathbf{y} = \{\texttt{postRef} : m_1,\ \texttt{authorHash} : H(\text{Alice})\}$, *decl* as above):

$$\frac{(\alpha_{\text{notify}}, m_1) \in \mathit{pending} \quad \text{all phases complete} \quad \Delta = \text{EXECUTE}_{\alpha_{\text{notify}}}(m_1, \mathbf{y}, \mathit{decl}, \mathcal{S}_1)}{(\mathcal{S}_1,\ R_0,\ \{(\alpha_{\text{notify}}, m_1)\}) \xrightarrow{(\alpha_{\text{notify}}, m_1)} (\mathcal{S}_2,\ R_0,\ \emptyset)}$$

where $\mathcal{S}_2 = \mathcal{S}_1$ with S_{Notif} updated to $(\langle m_2 \rangle, \mathit{Meta}_{\mathit{Notif}}, \mathit{ACL}_{\mathit{Notif}})$, $m_2 = (2, \texttt{notification}, \{\texttt{from} : \text{Alice}, \texttt{postRef} : m_1,\ \texttt{rippleId} : \mathit{rid}(m_1)\})$, and *pending* is empty because CALL's emitted event to `Inbox/Bob` is processed by *triggerAll*, which would enqueue any subscribers to that stream (none in this minimal example).

Final state. $\Sigma_2 = (\mathcal{S}_2,\ R_0,\ \emptyset)$ with: $M_{\text{Alice}} = \langle m_1 \rangle$, $M_{\text{Bob}} = \langle m_0 \rangle$, $M_{\text{Notif}} = \langle m_2 \rangle$, $M_{\text{Inbox/Bob}} = \langle m_2 \rangle$ (relation tables unchanged: R_0). The complete two-step execution is captured in two reduction steps with no global coordination: Alice's publisher node executed Step 1; the notification node executed Step 2, reading only S_{Alice} and writing only S_{Notif}. By Theorem 6.3 (after the second invocation is enqueued), if Bob had also posted simultaneously, his post action would be conflict-free with α_{notify} and could have executed in parallel.

5 The Phase Calculus

5.1 Overview

Every action in $\mathcal{MM}$ is structured into exactly five phases executed in strict order:

$$\text{VALIDATE} \rightarrow \text{COMPUTE} \rightarrow \text{REQUIRE} \rightarrow \text{EXECUTE} \rightarrow \text{CALL}$$

No phase may be skipped or reordered. Each phase has a precise type, a set of permitted operations, and a monotonicity property. Together they constitute the *phase calculus* of $\mathcal{MM}$—the execution language of the machine.

This design generalizes and formalizes the CEI pattern (Checks–Effects– Interactions) [2] from smart-contract engineering, extending it with an explicit pure-computation phase and a dependency- declaration phase.

5.2 Phase Definitions

Definition 5.1 (VALIDATE Phase)**.**

$$\textsc{Validate}_\alpha : \mathbf{x} \times \Sigma_{local} \to \{ok,\ reject(r)\}$$

Here Σ_{local} denotes a read-only snapshot of the locally cached *stream set—streams already held by the executing node, with no remote fetches permitted. This is a strict subset of the full materialization environment Σ_{read} available to* COMPUTE *and* REQUIRE*: $\Sigma_{local} \subseteq \Sigma_{read}$. The distinction is intentional:* VALIDATE *is cheap and local; remote I/O is deferred to* COMPUTE*.*
Input: *proposed input record $\mathbf{x}$ and a read-only view of locally cached stream state.*
Output: *acceptance or rejection with reason r.*
Purpose: *structural pre-screening before any I/O is attempted. Checks that $\mathbf{x}$ is well-typed, that required fields are present, and that locally-known invariants are satisfied (e.g. a user ID is syntactically valid; an amount is positive).*
Permitted: *reads of locally held stream state (no remote fetches); schema validation; structural constraint checks.*
Forbidden: *all state mutation; all network or protocol I/O; all stream appends.*
Note: *Rejection at this phase is cheap and local. It occurs before any external resource is consulted.*

Definition 5.2 (COMPUTE Phase)**.**

$$\textsc{Compute}_\alpha : \mathbf{x} \times \Sigma_{read} \to \mathbf{y}$$

Here Σ_{read} denotes the full stream materialization environment*: the active local stream set plus all streams reachable via remote fetch (HTTP GET, database SELECT, or any registered view capability). This is a strict superset of Σ_{local} from* VALIDATE*: $\Sigma_{local} \subseteq \Sigma_{read}$.*
Output: *derived computation record $\mathbf{y}$ containing all data needed by subsequent phases.*
Purpose: *assemble all information required to decide and execute the action. This includes fetching remote data and invoking generative models via read-only protocols.*
Permitted:

- *Reads from locally held streams (by stream URI).*
- *Remote fetches via idempotent, side-effect-free protocol operations: HTTP* `GET`*, IMAP* `FETCH`*, file system reads, database* `SELECT`*, or any capability registered as a* view capability $C \in \mathcal{C}_{view}$*. A view capability is one whose invocation produces no observable side effect at the remote system and whose result is deterministic for a given resource state.*
- Generative model inference*: invocations of large language models, diffusion models, embedding models, or any other AI inference capability registered as $C_{AI} \in \mathcal{C}_{view}$. These calls are view capabilities because they do not mutate remote state. Their outputs are* mostly deterministic *under fixed inputs and hyperparameters (see §12).*
- fetchOrCreate *operations: if the stream addressed by a given URI does not yet exist locally, it is materialized from its remote source and registered in the active stream set $\mathcal{S}$.*
- *Arbitrary pure computation over fetched data.*

Forbidden: *writes, appends, or any operation that mutates remote or local state; non-idempotent protocol operations (HTTP* `POST/PUT`*, SMTP* `SEND`*, etc.).*
Determinism note: *COMPUTE is* mostly deterministic *— exactly deterministic for pure computations and cached AI results, and approximately deterministic (with negligible failure probability ε) for uncached generative model calls. All invocation results are recorded in the action's audit trail to enable replay (§11) and content-addressable memoization (§12).*

Definition 5.3 (REQUIRE Phase)**.**

$$\textsc{Require}_\alpha : \mathbf{x} \times \mathbf{y} \times \Sigma_{read} \to \{\mathit{proceed}(\mathit{ReadSet}, \mathit{WriteSet}, \mathit{CapSet}),\ \mathit{reject}(r)\}$$

Input: *validated input record* $\mathbf{x}$*, the computation record* $\mathbf{y}$ *produced by COMPUTE, and read-only access to stream metadata in* Σ_{read} *(for rate-limit checking, ACL verification, etc.).*
Output: *either a* proceed *verdict with an explicit declaration of (a) all streams to be read (ReadSet), (b) all streams to be appended to (WriteSet), and (c) all mutating capabilities to be invoked (CapSet); or a* reject *verdict with reason* r*.*
Purpose: *REQUIRE is the* governance gate*. It evaluates the Policy function* $\mathcal{P}$ *over the proposed action and its computed context:*

$$\mathcal{P}(\mathit{actor}(\alpha),\ \alpha,\ \mathbf{x},\ \mathbf{y}) \to \{\textsc{allow},\ \textsc{deny}(r),\ \textsc{constrain}(\sigma)\}$$

If $\mathcal{P}$ *returns* DENY*, REQUIRE returns* $\mathit{reject}(r)$ *and the action does not proceed. If* $\mathcal{P}$ *returns* CONSTRAIN(σ)*, the declared sets are restricted to those permitted by the constraint record* σ *(e.g. rate limits, stream scope restrictions, value caps). If* $\mathcal{P}$ *returns* ALLOW*, REQUIRE returns* $\mathit{proceed}(\mathit{ReadSet}, \mathit{WriteSet}, \mathit{CapSet})$*.*
Retry semantics: *a rejection at REQUIRE may cause the invoking system to re-enter the action at VALIDATE with a modified input (e.g. reduced transfer amount, alternative target stream), forming a* policy negotiation loop *that terminates when either the Policy accepts or a maximum retry count is exceeded.*
Permitted: *evaluating the Policy function; inspecting* $\mathbf{x}$ *and* $\mathbf{y}$*; constructing the declaration sets.*
Forbidden: *new remote reads; any state mutation; any capability invocations.*

Definition 5.4 (EXECUTE Phase)**.**

$$\textsc{Execute}_\alpha : \mathbf{x} \times \mathbf{y} \times (\mathit{ReadSet}, \mathit{WriteSet}, \mathit{CapSet}) \to \Delta$$

Input: *all prior phase outputs and the declared dependency sets.*
Output: *a delta record* $\Delta = \{\Delta_i \mid i \in \mathit{WriteSet}\}$*—one message sequence per output stream.*
Purpose: *EXECUTE is the sole phase that mutates state. It applies the action's effects to the* local *streams owned by the executing publisher. All appends are to streams in WriteSet, all of which must be locally owned (Definition 3.8). Mutating capabilities in CapSet (e.g. HTTP* `POST`*, database writes) are invoked here.*
Permitted: *appending messages to streams in WriteSet; invoking capabilities in CapSet.*
Forbidden: *reads or writes to streams outside the declared sets; capability invocations outside CapSet; writes to streams owned by other publishers (cross-publisher writes are mediated through CALL and subscription, never through direct append).*

Note: *Because* Execute *only writes locally owned streams, it requires no cross-publisher coordination. The distributed effects of an action propagate entirely through the event emission of* Call.

Definition 5.5 (Call Phase)**.**

$$\textit{Call}_\alpha : \mathbf{x} \times \Delta \to \textit{Events}$$

Input: *original inputs and the execute-phase delta.*
Output: *a set of* outbound events *Events—typed messages emitted to named streams that other actors or actions may subscribe to.*
Purpose: *Call is the* distributed propagation boundary*. It does not enqueue internal action invocations directly. Instead, it emits events onto named streams. Any actor or action that has subscribed to those streams will be triggered by the subscription mechanism (Rule Trigger). In $\mathcal{MM}$, actors are themselves streams, so emitting an event to an actor's inbox stream is the same operation as emitting to any other stream.*
Permitted: *constructing and emitting typed event messages to streams declared in the action's subscription-output manifest.*
Forbidden: *direct stream appends bypassing the subscription mechanism; direct capability invocations; direct mutation of any stream.*
Note: *The decoupling between event emission (Call) and event handling (the next action's Validate) is what makes $\mathcal{MM}$ a* fully evented, push-only *system. The emitting action does not know or care which downstream actions will handle its events; it only knows which streams it publishes to.*

5.3 Phase Ordering and Monotonic Commitment

Proposition 5.6 (Phase Monotonicity)**.** *Each phase strictly increases the* commitment level *of the action:*

Phase	*Role*	*Commitment Level*	*Reversible?*
Validate	*Structural pre-screening*	*Possibility*	*Yes*
Compute	*Data assembly (reads/fetches)*	*Derivation*	*Yes*
Require	*Governance gate (policy)*	*Declaration*	*Yes*
Execute	*Local state mutation*	*Effect*	*No (append-only)*
Call	*Distributed event emission*	*Propagation*	*No (events in flight)*

No state mutation occurs before Execute; no distributed propagation occurs before Call. The Policy function is evaluated exclusively in Require, ensuring that governance is enforced after all necessary data is assembled (Compute) but before any irreversible effects are applied.

Proof. By the forbidden-operations constraints of Definitions 5.1 through 5.5: Validate is forbidden from all capability calls and all appends; Compute is forbidden from all mutating (non-idempotent) capability calls and all appends, but is permitted to invoke view

capabilities $C \in \mathcal{C}_{view}$; REQUIRE evaluates Policy and either rejects (reversible) or produces the declaration (reversible), with no I/O of any kind; only EXECUTE may append to local streams or invoke mutating capabilities; only CALL may emit outbound events. The ordering VALIDATE < COMPUTE < REQUIRE < EXECUTE < CALL is enforced by the EXECUTE reduction rule (§4). □ □

5.4 Relation to CEI and Smart-Contract Patterns

The Checks–Effects–Interactions (CEI) pattern recommends that smart contracts perform all checks before any state changes, and all state changes before any external calls. The $\mathcal{MM}$ phase calculus formalizes and significantly extends this [2]:

CEI Pattern	$\mathcal{MM}$ Phase	Key addition
Checks	VALIDATE	Structural only; no I/O
(none)	COMPUTE	Reads, fetches, materialization
(none)	REQUIRE	Policy enforcement; retry loop
Effects	EXECUTE	Local state only
Interactions	CALL	Event emission to subscribers

The two phases CEI lacks are precisely what make $\mathcal{MM}$ safe and efficient at scale. COMPUTE assembles all external data *before* the governance gate, so the Policy in REQUIRE can make an informed decision without triggering irreversible effects. REQUIRE then enforces governance and may loop with a modified proposal until the Policy is satisfied or the action is abandoned—a negotiation pattern that cannot be expressed in CEI. Finally, the separation of EXECUTE (local-only writes) from CALL (distributed event emission) makes it impossible for an action to trigger remote side effects before its own local state is committed.

Remark 5.7 (Information-Theoretic Justification of Phase Separation)**.** *The separation of* COMPUTE *from* EXECUTE *has a quantitative information-theoretic justification beyond mere software engineering discipline. In the PLT framework [20], an input* $\mathbf{x}$ *has a* code length $L(\mathbf{x}) = \lceil -\log_2 P_{\mathcal{M}}(\mathbf{x}) \rceil$ *under the generative model* $\mathcal{M}$*. High-probability inputs (low* L*) are exactly the ones for which the* COMPUTE *output* $\mathbf{y}$ *is likely to have been computed before and cached. Low-probability inputs (high* L*) are genuine residuals requiring fresh computation. The* REQUIRE *phase's explicit dependency declaration plays the role of the PLT's trie traversal: it identifies, before execution, which output streams are likely to be cache-populated. Actions whose* COMPUTE *output depends only on high-probability stream prefixes can be served from cache; only those touching low-probability (residual) streams need proceed to* EXECUTE*. The four-tier computation spectrum of the PLT framework maps directly onto the* $\mathcal{MM}$ *phase structure: Tier 1 (exact cache hit)* $\subseteq$ COMPUTE*, Tier 4 (full computation)* $\subseteq$ EXECUTE*.*

5.5 Worked Example: Charging a Customer

The following example illustrates all five phases. COMPUTE fetches the user's account data via an idempotent read. REQUIRE checks the Policy (which may enforce a rate limit or

spending cap) and may reject, causing the caller to retry with a reduced amount. EXECUTE writes only to locally owned streams. CALL emits an event that any subscriber may act on.

```
defineTool({
  name: "chargeCustomer",

  // Structural pre-screening: no I/O.
  VALIDATE({ userId, amount }) {
    if (amount <= 0) reject("amount must be positive");
    if (typeof userId !== "string") reject("invalid userId");
  },

  // Assemble all needed data, including remote reads.
  // HTTP.get is a view capability (idempotent, no side effects).
  COMPUTE({ userId, amount }) {
    const user    = Streams.fetchOrCreate("User/" + userId);
    const account = HTTP.get("https://billing.internal/account/" +
       userId);
    const fee     = amount * account.feeRate;
    return { user, account, fee, net: amount - fee };
  },

  // Governance gate: apply Policy; may reject, causing a retry loop
     .
  // Returns the declared write/capability sets only if Policy
     allows.
  REQUIRE({ userId, amount }, { account }) {
    // Policy checks spending cap, rate limit, KYC status, etc.
    Policy.enforce({
      actor:   userId,
      action:  "charge",
      amount,
      account,           // Policy can inspect computed context
    });
    return {
      writes:       ["Payment/" + userId, "AuditLog"],
      capabilities: ["HTTP.stripe"],   // mutating capability
    };
  },

  // Local-only writes. No cross-publisher appends here.
  EXECUTE({ userId, amount }, { fee, net }) {
    const receipt = HTTP.stripe.charge(userId, amount);
    return {
      "Payment/" + userId: [{ amount, fee, net, receipt,
                              contentHash: H(receipt) }],
      "AuditLog":          [{ action: "charge", userId, amount,
                              ts: Date.now() }],
```

```
    };
  },

  // Emit events to named streams. Subscribers decide what to do.
  // Does not call notifyUser directly---it subscribes to "Payments/
     events".
  CALL({ userId }, delta) {
    const status = delta["Payment/" + userId][0].receipt.status;
    return [
      { stream: "Payments/events",
        event:  { type: "charged", userId, status } },
    ];
  }
});
```

Listing 1: Phase-structured action illustrating fetch in COMPUTE, policy enforcement in REQUIRE, local-only writes in EXECUTE, and event emission in CALL.

6 Embarrassing Parallelism

6.1 Setup and Structural Enabling Theorem

Remark 6.1 (Framing)**.** *The result in this section is a* structural enabling theorem*: it establishes that $\mathcal{MM}$'s design—specifically, the combination of append-only streams, single-publisher ownership, and* REQUIRE*-declared write sets—creates the conditions under which embarrassing parallelism holds as a* schedulable *property. The key distinction from prior systems is not merely that disjoint writes commute (this is trivially true of any append-only structure), but that* disjointness is statically decidable at enqueue time *from the* REQUIRE *declaration, without executing the action. This is what separates $\mathcal{MM}$ from blockchains and ICP, where conflict detection requires speculative execution or runtime state inspection.*

Definition 6.2 (Embarrassingly Parallel Set)**.** *A set of action invocations $I = \{(\alpha_1, \mathbf{x}_1), \ldots, (\alpha_k, \mathbf{x}_k)\} \subseteq$ pending is* embarrassingly parallel *if: (i) every pair has disjoint declared write sets: $\forall i \neq j : \mathit{WriteSet}(\alpha_i) \cap \mathit{WriteSet}(\alpha_j) = \emptyset$; and (ii) no invocation in I reads a stream written by another: $\forall i \neq j : \mathit{ReadSet}(\alpha_i) \cap \mathit{WriteSet}(\alpha_j) = \emptyset$.*

Condition (i) prevents write–write conflicts; condition (ii) prevents read-after-write dependencies within the set, ensuring that no invocation in I needs to observe another's output. Together they define a *conflict-free* set.

Theorem 6.3 (Structural Enabling: Safe Parallel Execution)**.** *Let I be an embarrassingly parallel set of invocations. Then:*

(a) ***Non-interference:*** *executing any subset of I simultaneously on independent nodes produces no conflicts and requires no communication or locks.*

(b) ***Commutativity:*** *for any two orderings σ, σ' of I, executing I in order σ from state Σ_0 yields the same final state as executing in order σ'.*

(c) ***Static decidability:*** *the conflict-free condition is decidable at enqueue time from* REQUIRE *declarations, without executing any action.*

Proof. (a) Non-interference. By Definition 5.2, COMPUTE reads only streams in $\mathit{ReadSet}(\alpha_i)$; by Definition 5.4, EXECUTE appends only to streams in $\mathit{WriteSet}(\alpha_i)$. Since I is conflict-free, condition (ii) guarantees $\mathit{ReadSet}(\alpha_i) \cap \mathit{WriteSet}(\alpha_j) = \emptyset$, so no stream that α_i reads (in COMPUTE or EXECUTE) is one that α_j writes. Condition (i) guarantees $\mathit{WriteSet}(\alpha_i) \cap \mathit{WriteSet}(\alpha_j) = \emptyset$, so their appends land on distinct streams. Stream appends are non-destructive (Definition 3.6): α_j's writes extend streams that α_i never reads, and vice versa. Therefore neither action's behavior is affected by the other's execution, and no communication is required.

(b) Commutativity. Let Δ_i be the delta produced by α_i from state $\Sigma_0 = (\mathcal{S}_0, R_0, \emptyset)$. Since condition (ii) holds, $\mathit{ReadSet}(\alpha_i) \cap \mathit{WriteSet}(\alpha_j) = \emptyset$; therefore the streams read by α_i's COMPUTE phase are unmodified by α_j's EXECUTE, so Δ_i is the same whether or not α_j has executed first. Formally, for any $j \neq i$, let $\mathcal{S}^{(j)} = \mathit{applyDeltas}(\mathcal{S}_0, \Delta_j)$. Since $\mathit{WriteSet}(\alpha_j)$ is disjoint from both $\mathit{ReadSet}(\alpha_i)$ and $\mathit{WriteSet}(\alpha_i)$, the streams inspected by α_i are identical in $\mathcal{S}_0$ and $\mathcal{S}^{(j)}$, so α_i produces the same Δ_i from either starting stream set. By induction on $|I|$, applying the deltas in any order yields the same final stream set $\mathit{applyDeltas}(\mathcal{S}_0, \Delta_1 \cup \cdots \cup \Delta_k)$, where the union is well-defined because the Δ_i have pairwise disjoint domains (condition (i)) and *applyDeltas* over disjoint-domain deltas is order-independent. (The relation tables R_0 are unaffected since none of the Δ_i contain relation-typed messages by the local-action assumption.) □

(c) Static decidability. By Definition 5.3, $\text{REQUIRE}_{\alpha_i}$ takes inputs $(\mathbf{x}_i, \mathbf{y}_i, \Sigma_{\mathit{read}})$ and returns the declaration sets. For static decidability to hold, we require that REQUIRE's output depends only on the *structural identity* of streams accessed by COMPUTE (e.g. stream URIs derived from $\mathbf{x}_i$), not on the runtime *content* of remotely fetched data that varies between invocations. This is a well-formedness condition on action definitions: a well-formed action's REQUIRE declaration is a function of its input record $\mathbf{x}$ and the schema of accessed streams, not of the values returned by remote fetches. Under this condition, $\mathit{WriteSet}(\alpha_i)$ and $\mathit{ReadSet}(\alpha_i)$ are computable from $\mathbf{x}_i$ alone, before any phase executes. Set intersection is decidable in $O(|I|^2 \cdot \max_i |\mathit{WriteSet}(\alpha_i) \cup \mathit{ReadSet}(\alpha_i)|)$ time by pairwise checks, with no knowledge of action internals and no speculative execution. □ □

6.2 Scope and Scaling of Parallelism

Define the *write-conflict graph* $W = (V_W, E_W)$ where $V_W = \mathit{pending}$ and $((\alpha_i, \mathbf{x}_i), (\alpha_j, \mathbf{x}_j)) \in E_W$ iff $I = \{(\alpha_i, \mathbf{x}_i), (\alpha_j, \mathbf{x}_j)\}$ is *not* conflict-free. By Theorem 6.3, every independent set in W may be executed simultaneously.

Corollary 6.4 (Publisher-Proportional Parallelism)**.** *Let P be the number of distinct stream owners (publishers) among all invocations in pending, and assume (i) all invocations are* local actions *(Definition 13.1: both ReadSet and WriteSet are subsets of the executing publisher's own streams), and (ii) each publisher's streams are disjoint from all others (the standard ownership model, Definition 3.8). Then all pending invocations from distinct publishers form an embarrassingly parallel set, so the maximum independent set in W has size at least P.*

If furthermore each publisher has q pending invocations that are pairwise conflict-free within that publisher's namespace, the maximum parallel set has size at least $P \cdot q$.

Proof. Under assumption (ii), streams owned by publisher p are disjoint from streams owned by any $p' \neq p$. Under assumption (i), every invocation $(\alpha_i, \mathbf{x}_i)$ of publisher p has $\mathit{WriteSet}(\alpha_i) \cup \mathit{ReadSet}(\alpha_i) \subseteq \mathit{streams}(p)$. Therefore for any two invocations $(\alpha_i, \mathbf{x}_i)$ and $(\alpha_j, \mathbf{x}_j)$ of distinct publishers $p \neq p'$:

$$\mathit{WriteSet}(\alpha_i) \cap \mathit{WriteSet}(\alpha_j) = \emptyset \quad \text{and} \quad \mathit{ReadSet}(\alpha_i) \cap \mathit{WriteSet}(\alpha_j) = \emptyset$$

since $\mathit{streams}(p) \cap \mathit{streams}(p') = \emptyset$ by assumption (ii). Both conditions of the embarrassingly parallel definition are satisfied. Selecting one invocation per publisher gives an independent set of size P.

For the $P \cdot q$ bound: if publisher p has q pairwise conflict-free invocations, all q satisfy both conditions internally (by hypothesis) and also satisfy both conditions with respect to any other publisher's invocations (by the argument above). Therefore all $P \cdot q$ invocations form a conflict-free set. □ □

In practice this means: a federated social network with $P = 10^6$ distinct user-publishers can process one action per user in parallel with zero coordination. This scales linearly with the number of publishers—a property no existing blockchain achieves.

Corollary 6.5 (Efficient Conflict-Free Scheduling)**.** *Given a pending multiset of n action invocations, a maximal embarrassingly parallel subset can be computed in $O(n^2 \cdot d)$ time, where $d = \max_i |\mathit{WriteSet}(\alpha_i) \cup \mathit{ReadSet}(\alpha_i)|$, using a greedy algorithm: add each invocation to the current parallel set if it is conflict-free with all already-added invocations.*

Proof. By Theorem 6.3(c), the conflict-free check between any pair $(\alpha_i, \mathbf{x}_i)$ and $(\alpha_j, \mathbf{x}_j)$ costs $O(d)$ (two set-intersection checks, each on sets of size at most d). The greedy pass iterates over all n invocations and checks each against the current set (at most n comparisons per element), giving $O(n^2 \cdot d)$ total. The result is a maximal independent set in W by construction: an invocation is added if and only if it conflicts with no already-selected invocation. □ □

6.3 Comparison with Blockchains

In a permissionless blockchain (e.g. Ethereum), every transaction modifying the global state must be *totally ordered* by the consensus mechanism. Even two transactions that modify entirely disjoint account balances must be sequenced into a block. This is because the global state is a single mutable trie: there is no declared write set that would allow the protocol to determine at scheduling time that the transactions do not conflict.

Proposition 6.6 (Global-State Blockchain Serialization)**.** *In a blockchain system whose global state is a single shared trie (e.g. Ethereum's account-state trie), no two transactions can be certified non-conflicting without either (a) executing them and checking the accessed state keys, or (b) requiring senders to declare accessed state in advance. Without advance declaration, all transactions must be serialized in the worst case.*

Proof. Consider two transactions T_1 and T_2 that both modify state key k. Without execution or advance declaration, the protocol cannot determine whether T_1 and T_2 access the same key: the accessed key set is determined by the transaction's bytecode execution trace, which depends on the current state. We construct a worst-case workload: let T_1 read key k_1 and write key k iff state$[k_1] > 0$; let T_2 write key k unconditionally. Without executing T_1, no static analysis of its bytecode can determine whether it will access k, because the branch depends on runtime state. Therefore T_1 and T_2 cannot be certified non-conflicting without execution. Since this applies to any pair of transactions with data-dependent access patterns, the worst case requires full serialization. □ □

Remark 6.7. *UTXO-model chains (e.g. Bitcoin) and account-declaration chains (e.g. Solana) allow static conflict detection from transaction structure alone. $\mathcal{MM}$ generalizes the account-declaration approach to arbitrary typed streams, without restricting the model to a fixed account schema.*

$\mathcal{MM}$ avoids this by requiring that write sets be *declared in* REQUIRE *before execution*. The scheduler can determine at enqueue time whether two invocations conflict, enabling parallelism without optimistic concurrency control or speculative execution.

System	Parallelism	Coordination Required	Declared Writes
Ethereum (single-threaded EVM)	Serial	Global consensus	No
Ethereum (parallel EVM, e.g. Sei)	Optimistic	Conflict detection	Partial
Solana (Sealevel)	Parallel	Declared accounts	Yes (partial)
ICP Canisters (intra-subnet)	Serial per subnet	Subnet consensus	No
ICP Canisters (cross-subnet)	Async	Inter-subnet messaging	No
Magarshak Machine	Embarrassing	None (disjoint writes)	Yes (complete)

Solana comparison. Solana's Sealevel runtime [29] requires transactions to declare the accounts they access, enabling parallel execution of non-conflicting transactions. This is the closest prior art to $\mathcal{MM}$'s approach. $\mathcal{MM}$ generalizes Sealevel in three ways: (i) write-set declaration is extended to full phase-structured actions; (ii) the model is not restricted to a fixed account model but applies to arbitrary typed streams; (iii) there is no global validator set—each publisher is the sole authority on their own streams.

ICP comparison. ICP canisters within a subnet execute sequentially in message order, enforced by subnet consensus. Cross-subnet calls are asynchronous and may involve rollback on one-way traps. $\mathcal{MM}$ achieves what ICP achieves within a subnet—ordered execution per publisher—but also achieves inter-publisher parallelism that ICP does not, since $\mathcal{MM}$ has no notion of a "subnet" that serializes otherwise-disjoint work.

7 Push-Only Reactive Execution

7.1 No Polling

A fundamental principle of $\mathcal{MM}$ is that *no action polls a stream.* There is no "read stream, check if updated" loop in the model. Instead, all execution is push-based: an action executes only when enqueued by either an external append (Rule TRIGGER) or a downstream CALL invocation (Rule EXECUTE).

Definition 7.1 (Subscription)**.** *A* subscription *of action type α to stream S_i is a registered association (S_i, α, F) where $F \subseteq \mathcal{T}$ is a filter set of message types. When message m with $\tau(m) \in F$ is appended to S_i by Rule TRIGGER, the invocation $(\alpha, \mathbf{x})$ is enqueued in pending with $\mathbf{x} = m$ (the triggering message serves as the input record).*

Proposition 7.2 (Polling-Freedom)**.** *In a well-formed $\mathcal{MM}$ instance, no action invocation is scheduled except by Rules TRIGGER or EXECUTE. Therefore no action ever accesses a stream at a time not causally downstream of an event on that stream.*

Proof. By inspection of the operational semantics (§4). The only rules that add entries to *pending* are TRIGGER (labeled APPEND in the inference rule, enqueuing subscriptions on external appends) and EXECUTE (via *triggerAll*($\mathcal{E}_{\text{out}}$) in CALL). There is no "timer" or "poll" rule in the four-rule semantics. Therefore the only way an action can be scheduled is in causal response to a prior append event. □ □

7.2 Causal Ordering Without Polling

Theorem 7.3 (Causal Consistency of Reactive Execution)**.** *Let m_1 be appended to S_i and let m_2 be the first message appended to S_j in response (via any finite chain of TRIGGER/EXECUTE steps). Then $t(m_1) < t(m_2)$. More generally, if $m_1, m_2, \ldots, m_n$ is any causal chain where each m_{k+1} is produced in response to m_k, the timestamps are strictly increasing: $t(m_1) < t(m_2) < \cdots < t(m_n)$.*

Proof. By induction on the length of the trigger chain.

Base case. m_2 is produced by action α directly subscribed to S_i. By Rule TRIGGER and Definition 7.1, the invocation $(\alpha, \mathbf{x})$ with $\mathbf{x} = m_1$ is enqueued at the moment m_1 is appended to S_i. Rule EXECUTE applies α, whose EXECUTE phase appends m_2 to S_j. By Definition 3.6, the append operation requires $t(m_2) > t(\text{last}(M_j))$; since m_2 is appended strictly after m_1 (all of α's execution is causally posterior to the enqueuing of (α, m_1), which occurs at timestamp $t(m_1)$), and timestamps are elements of $\mathbb{N}$ drawn from the strictly increasing sequence required by Definition 3.3, we have $t(m_2) > t(m_1)$.

Inductive step. Suppose $t(m_k) > t(m_1)$ for message m_k produced in the k-th step of the chain. Any action β subscribed to the stream containing m_k is enqueued by Rule TRIGGER only after m_k is appended; its output m_{k+1} must satisfy $t(m_{k+1}) > t(\text{last}(\cdot))$ by Definition 3.6, and since its execution is causally posterior to m_k's append, $t(m_{k+1}) > t(m_k) > t(m_1)$. □

□

Remark 7.4 (Stronger Causal Guarantees via Frontier Metadata)**.** *The theorem above establishes timestamp ordering, which is the property derivable from the operational semantics alone. Stronger causal guarantees — such as "any observer of S_j after m_2 is guaranteed to have observed S_i at the state including m_1" — hold when actions are additionally required to propagate the causal frontier (e.g. vector-clock-style) in their output message payloads. This is a valid and natural design constraint on well-formed actions but is not enforced by the $\mathcal{MM}$ model itself, and is therefore left as an application-level concern.*

7.3 Reactive Completeness

Theorem 7.5 (Reactive Completeness)**.** *A $\mathcal{MM}$ instance with subscriptions is equivalent to a complete event-driven system: (i) every state change is caused by a message append; (ii) every message whose type matches a subscription filter F triggers all subscribed downstream actions; (iii) the complete history of all state changes is recoverable from stream logs.*

Proof. (i) The only rules that modify state are CREATE (adds a stream, triggered by a governance-authorized creation request), TRIGGER (appends a message and atomically updates R via *updateR* for relation-typed messages), and EXECUTE (applies deltas to streams via *applyDeltas*). Every such modification is initiated by or directly results from a message append. Stream log appends are messages (Definition 3.3); relation table changes are caused by relation-typed messages posted to streams, which are themselves appends. (ii) Rule TRIGGER enqueues all subscriptions (S_i, α, F) where $\tau(m) \in F$. (iii) By Definition 3.6, all messages are permanently retained in M_i; stream state is recoverable by scanning M_i from index 0. Relation table state is recoverable by replaying all relation-typed messages up to time T. □ □

8 Ripple Deduplication

8.1 The Problem: Event Storms in Evented Systems

In a fully evented system, a single external append can trigger a cascade of reactions across many streams. If the subscription graph contains cycles (stream A triggers an action writing to stream B, which triggers an action writing back to A), or if the same event propagates along multiple paths to the same subscriber, an action may be invoked multiple times for a single originating event. This is the *event storm* or *double-ripple* problem.

Standard solutions—global deduplication tables, distributed sequence numbers, two-phase commit across subscribers—all require global coordination, defeating the locality benefits established in §6 and §13.

8.2 Ripple Identifiers

Definition 8.1 (Ripple)**.** *A* ripple *is the complete causal chain of events, actions, and state changes that originates from a single external append. Each ripple is assigned a globally*

unique ripple identifier*:*

$$rid = H(originStreamId,\ t_{origin},\ originMessageHash)$$

where H is a collision-resistant hash function. The ripple identifier is deterministic given the originating event and requires no coordination to compute.

When an action is triggered by an event carrying ripple identifier *rid*, it propagates *rid* to all events it emits in Call. Concretely, each emitted message includes a field `rippleId` = *rid* in its payload. Thus every event in the causal chain of a ripple carries the same *rid* value, stored under the key `rippleId`.

8.3 Local Deduplication

Each stream maintains a *ripple log*: a bounded, locally managed set of recently processed ripple identifiers.

Definition 8.2 (Ripple Log)**.** *The* ripple log *of stream S_i is a finite set $RL_i \subseteq \mathcal{H}$ of ripple identifiers, where $\mathcal{H}$ is the hash space. Entries expire after a configured time-to-live TTL (long enough to cover any realistic propagation delay in the deployment topology).*

Definition 8.3 (Deduplication Rule)**.** *When action α subscribed to S_i is triggered by event m whose payload contains field `rippleId` = $rid(m)$, the invocation is* deduplicated *if $rid(m) \in RL_i$. In that case the invocation is discarded without execution. Otherwise, $rid(m)$ is added to RL_i and the invocation proceeds normally. Events without a `rippleId` field (i.e. originating external appends) are assigned a fresh rid by the receiving publisher before being checked against RL_i.*

Proposition 8.4 (Local Deduplication Sufficiency)**.** *Under the standard ownership model, ripple deduplication requires only local state: no distributed coordination, no global sequence numbers, and no two-phase commit.*

Proof. Each stream S_i is owned by a single publisher p_i (Definition 3.8). The ripple log RL_i is maintained by p_i and checked at append time—a purely local operation. Two events carrying the same *rid* that arrive at S_i from different propagation paths are deduplicated at S_i's publisher without any communication with other publishers. The hash-based *rid* is computed deterministically from the originating event, so all copies of the same ripple event carry identical *rid* values regardless of the path they traveled. □ □

8.4 Bounded Memory and TTL Expiry

The ripple log is bounded: entries are evicted after *TTL* seconds. The *TTL* must exceed the longest realistic event propagation delay in the deployment topology. For a federation of servers with millisecond latency, a TTL of minutes is sufficient. The memory cost is $O(R \cdot TTL/interval)$ where R is the maximum ripple arrival rate and *interval* is the expected inter-ripple gap — easily bounded for governed systems where the Policy function enforces rate limits (§9).

Remark 8.5 (No Global Sequence Numbers)**.** *Classical message-broker deduplication (e.g. Kafka's idempotent producer, AMQP delivery tags) requires a globally monotone sequence number assigned by a central broker. $\mathcal{MM}$'s hash-based ripple IDs require no such central authority: the originating event's content uniquely and verifiably identifies the ripple, consistent with $\mathcal{MM}$'s broader principle that all coordination is local wherever possible.*

Corollary 8.6 (Ripple Termination)**.** *Under the standard ownership model, every ripple terminates in a finite number of action invocations, provided: (i) the subscription graph over streams has finite out-degree at every stream; and (ii) the ripple log RL_i persists for at least TTL seconds after each entry is added.*

Proof. Let G_{sub} be the directed graph whose vertices are streams and whose edges represent subscriptions: $(S_i, S_j) \in G_{\text{sub}}$ iff some action subscribed to S_i writes to S_j. By condition (i), G_{sub} has finite out-degree, so each stream has at most $d < \infty$ subscribed action types.

Consider a single ripple with identifier *rid*. Each time a stream S_i receives an event with *rid*, the event is either: (a) deduplicated (if $rid \in RL_i$), enqueuing no further invocations; or (b) processed, adding *rid* to RL_i. By condition (ii), once *rid* is in RL_i, all future arrivals of the same ripple at S_i are deduplicated. Therefore each stream processes the ripple *at most once*.

When stream S_i processes the ripple, it enqueues at most d_i action invocations, where d_i is the number of action types subscribed to S_i with a filter matching the ripple message's type. Each such invocation writes to streams in its declared WriteSet, triggering the ripple at those streams—which are also processed at most once. The total number of action invocations is therefore at most

$$\sum_{S_i \in \mathcal{S}^T} d_i \;\leq\; |\mathcal{S}^T| \cdot d_{\max}$$

where $d_{\max} = \max_i d_i < \infty$ by condition (i). Since $|\mathcal{S}^T|$ is finite (Definition 3.4) and $d_{\max} < \infty$, the total number of invocations is finite. □ □

9 Governance on Metadata: Rate-Bounded Value Transmission

9.1 Streams as Governance Layers Over Opaque Content

As noted in Remark 3.5, stream messages may carry content-address hashes rather than raw data. This has a significant governance consequence: $\mathcal{MM}$'s policy enforcement operates on *metadata* — stream identifiers, message types, timestamps, content hashes, actor identities, and declared dependency sets — without requiring access to the underlying content.

Proposition 9.1 (Governance Without Content Access)**.** *The Policy function $\mathcal{P}$ can enforce access control, rate limits, value caps, and auditing requirements on any stream regardless of whether the stream's message payloads are encrypted, hashed, or otherwise opaque.*

Proof. The REQUIRE phase is evaluated with inputs $(\mathbf{x}, \mathbf{y}, \Sigma_{read})$ (Definition 5.3), where Σ_{read} provides read-only access to the stream set including message logs. Within REQUIRE,

the Policy function is invoked as $\mathcal{P}(actor, \alpha, \mathbf{x}, \mathbf{y})$; rate-limit and ACL checks use additional metadata from Σ_{read}. Concretely: (1) **Access control** is enforced via ACL_i (Definition 3.4), which is part of the stream object accessible via Σ_{read} — metadata, not content; (2) **Rate limits** are enforced by counting messages in M_i within a time window — timestamps of messages are in the log itself, not encrypted payloads; they are accessible via Σ_{read} without decrypting any d field; (3) **Value caps** are enforced by reading declared structured fields in $\mathbf{x}$ such as `amount` — the action input, not an encrypted payload; (4) **Auditing** requirements are met by *actor* (a public key) and α (the action type identifier), both known to REQUIRE without content access. None of (1)–(4) require decrypting any message payload d. □ □

9.2 Pre-Set Rails and Rate-Bounded Topology

A key design principle of $\mathcal{MM}$ is that the *topology of allowed interactions is established in advance* through the subscription and ACL system, subject to governance. Subscriptions define which streams can trigger which actions; ACLs define which actors can append to which streams; and the Policy function constrains the rate and volume of value that can flow along each edge.

Definition 9.2 (Interaction Rails)**.** *The* rails *of an $\mathcal{MM}$ instance are the set of active subscriptions and ACL entries at a given time, together with the rate and value bounds encoded in the Policy function. Rails define the* governance topology*: which information and value flows are permitted, at what rates, and under what conditions.*

Rails are themselves stream data: subscriptions are created by appending subscription events to a registry stream, ACL changes are delegation messages (Definition 3.9), and Policy updates are governed transitions. This means the governance topology is auditable and version-controlled in the same append-only log model as all other state.

Remark 9.3 (Efficiency of Pre-Set Rails)**.** *Because the rails are set up in advance and checked locally at REQUIRE, an action that falls within pre-authorized bounds requires no cross-publisher negotiation at execution time. The governance cost is paid once (when the rails are established) rather than on every event. This contrasts with smart-contract systems, where each transaction independently validates against the current global state, paying consensus costs every time.*

10 Caching as a Derived Property

10.1 Immutable Prefixes Enable Safe Caching

Lemma 10.1 (Prefix Immutability)**.** *For any stream S_i and any time $T' > T$, M_i^T is a prefix of $M_i^{T'}$.*

Proof. The rules that modify M_i are TRIGGER (external appends, applying Definition 3.6 directly) and EXECUTE (via *applyDeltas*, which also applies Definition 3.6 to each stream in the declared WriteSet). In both cases Definition 3.6 strictly extends the sequence: it appends a single new message and rejects any message whose timestamp does not strictly exceed the

current last entry. It never modifies or removes existing elements. Therefore every historical state M_i^T is a prefix of every future state $M_i^{T'}$ for $T' > T$. □ □

Lemma 10.1 is the formal basis for safe caching: any cached prefix is guaranteed to remain valid indefinitely—it is not "stale" but merely "behind the head."

Corollary 10.2 (Append-Only Safety)**.** *No message in any stream is ever removed or modified. For all streams S_i and all times $T' > T$: $M_i^T[k] = M_i^{T'}[k]$ for every index $k \leq |M_i^T|$.*

Proof. Immediate from Lemma 10.1: M_i^T is a prefix of $M_i^{T'}$, so all elements of M_i^T appear unchanged at the same indices in $M_i^{T'}$. □ □

Corollary 10.3 (Monotone Stream Universe)**.** *The active stream set $\mathcal{S}^T$ is non-decreasing in T: for all $T' > T$, $\mathcal{S}^T \subseteq \mathcal{S}^{T'}$.*

Proof. The only reduction rule that modifies $\mathcal{S}$ is CREATE, which adds a stream: $\mathcal{S} \mapsto \mathcal{S} \cup \{S_{id}\}$. No rule removes a stream. Therefore $\mathcal{S}^T \subseteq \mathcal{S}^{T'}$ for all $T' > T$. □ □

Definition 10.4 (Cache Entry and Incremental Update)**.** *A* cache entry *for S_i at actor a is a pair $(k_a, M_i[1..k_a])$ where $k_a \leq |M_i|$. An* incremental update *is the suffix $M_i[k_a + 1.. |M_i|]$.*

10.2 Dependency-Aware Cache Invalidation

Definition 10.5 (Stream Dependency Graph)**.** *The* stream dependency graph *$D = (\mathcal{S}, F_D)$ has $(S_i, S_j) \in F_D$ iff there exists an action type α with $S_i \in \mathit{ReadSet}(\alpha)$ and $S_j \in \mathit{WriteSet}(\alpha)$. An edge (S_i, S_j) means "a change in S_i can cause a change in S_j."*

Theorem 10.6 (Minimal Cache Invalidation)**.** *When a new message is appended to S_i, the only streams whose derived (materialized) views may change are those reachable from S_i in D. The cached views of all other streams remain valid.*

Proof. An action α that writes to S_j can only be triggered (via Rule TRIGGER and the subscription mechanism) by an append to some stream in $\mathit{ReadSet}(\alpha)$, i.e. some stream S_k with $(S_k, S_j) \in F_D$. If S_j is not reachable from S_i in D—meaning no path in F_D leads from S_i to S_j—then no append to S_i can trigger any action whose write set includes S_j. Therefore M_j is not modified. By Lemma 10.1, the cached prefix of S_j remains valid. □ □

Theorem 10.6 formalizes the infamously hard problem of cache invalidation: because all dependencies are declared in REQUIRE, the dependency graph D is statically known, and invalidation is computable without runtime tracking.

Remark 10.7 (Quantitative Complement via PLT)**.** *Theorem 10.6 establishes* which *streams require re-materialisation when a new message arrives — a structural result derived from declared dependencies. It does not quantify* how much *benefit accrues from pre-populating the cache based on prior probability rather than observed frequency. This quantitative question is answered by the Prior-Guided Caching Theorem of Magarshak [20]: under a stationary generative distribution, a cache initialized from the model's own probability estimates achieves strictly lower expected cost than any LFU-based cache for all query counts below a threshold*

T_0 that grows with the entropy concentration of the distribution. In $\mathcal{MM}$ terms, this means that a publisher who populates the cache of its high-probability streams before *traffic arrives — using the prior distribution over likely action invocations — strictly dominates a reactive cache for the initial operating phase. The two results are complementary: $\mathcal{MM}$ provides the structural invalidation policy; the PLT theorem provides the initialization and retention policy.*

11 Replay and Determinism

11.1 State Reconstruction

Theorem 11.1 (State Reconstruction)**.** *The complete state of a $\mathcal{MM}$ instance at any time T is uniquely determined by the initial state Σ_0 and the sequence of external appends $\mathbf{a}_1, \mathbf{a}_2, \ldots$ up to time T.*

Proof. By induction on the number of reduction steps from Σ_0.

Base case. Σ_0 is given; it is trivially determined.

Inductive step. Suppose Σ^k is uniquely determined by Σ_0 and the external appends up to step k. There are two cases for the next step:

Rule Trigger: the next step is an external append $\mathbf{a}_{k+1} = (a, i, m)$. The new state Σ^{k+1} is uniquely determined by Σ^k and $\mathbf{a}_{k+1}$: TRIGGER deterministically computes $\mathit{triggers}(i, m)$ from the subscription registry (fixed) and $\tau(m)$ (part of $\mathbf{a}_{k+1}$); and $\mathit{updateR}(R^k, i, m)$ is a deterministic function of (R^k, i, m) (Definition 3.10) that updates both tables atomically. Therefore both the new stream set $\mathcal{S}^{k+1}$ and the new relation tables R^{k+1} are uniquely determined.

Rule Execute: the next step is the execution of some $(\alpha, \mathbf{x}) \in \mathit{pending}$. The phase functions COMPUTE_α, REQUIRE_α, and EXECUTE_α are deterministic functions of $(\mathbf{x}, \Sigma^k)$, *except* for external capability calls in EXECUTE. These capability call results are recorded as messages in designated audit streams (appended by the capability itself), so they are part of Σ^k's stream logs. A replay reading from these audit streams instead of re-invoking the capability recovers the same result. Therefore Σ^{k+1} is uniquely determined by Σ^k.

Rule Retry: the next step is a policy rejection. $\text{REQUIRE}_\alpha(\mathbf{x}, \mathbf{y}, \Sigma^k_{read})$ is a deterministic function of $(\mathbf{x}, \mathbf{y}, \Sigma^k)$ and the Policy function P (fixed), where Σ^k_{read} is the read-only view of Σ^k. The rejection reason r and the negotiated input $\mathbf{x}' = \mathit{negotiate}(\mathbf{x}, r)$ are both deterministic. The new pending set is therefore uniquely determined. Σ^{k+1} contains the same stream set and relation tables as Σ^k (neither is modified by RETRY).

By induction, Σ^T is uniquely determined by Σ_0 and $\mathbf{a}_1, \ldots, \mathbf{a}_T$ for all T. □ □

11.2 Deterministic Replay and Time Travel

Definition 11.2 (Replay)**.** *A* replay *of a $\mathcal{MM}$ instance from time T_0 is the execution of the reduction rules starting from $\Sigma^{T_0} = (\mathcal{S}^{T_0}, R^{T_0}, \mathit{pending}^{T_0})$, with all external capability results replaced by their recorded values from audit streams. The relation tables R^{T_0} are part of the starting state and are fully determined by replaying all relation-typed messages up to T_0 (Reactive Completeness, Theorem 7.5(iii)).*

Corollary 11.3 (Replay Determinism)**.** *A replay from T_0 produces an execution trace identical to the original.*

Proof. By Theorem 11.1, the state at every step from T_0 onward is uniquely determined by Σ^{T_0} and the sequence of external appends and recorded capability results after T_0. A replay starts from the same Σ^{T_0}, uses the same external appends from the append log, and substitutes recorded capability results from the audit streams for live capability calls. Since both inputs are identical to those of the original execution, every reduction step produces the same state transition by the inductive argument of Theorem 11.1. □ □

11.3 Stream Forking for Scenario Analysis

Definition 11.4 (Fork)**.** *A* fork *of S_i at message index k is a new stream $S'_i \in \mathbb{S} \setminus \mathcal{S}$ with $M^0_{i'} = M_i[1..k]$ and a fresh owner, ACL, and subscription set.*

Forking enables counterfactual analysis: starting from a known historical state, one may apply a different action set or policy function and compare outcomes. This is a formal foundation for regulatory scenario analysis, A/B testing, and distributed system debugging.

12 AI Capabilities, Mostly-Deterministic Computation, and Probabilistic Consensus

12.1 Generative Models as View Capabilities

The Compute phase permits invocations of generative AI models—large language models (LLMs), diffusion models, embedding models, and similar inference systems—registered as view capabilities $C_{AI} \in \mathcal{C}_{view}$. This is justified because such calls produce no observable side effect at the remote inference system from the caller's perspective: the model's weights are read-only, and two callers sending the same request receive (essentially) the same response.

This treatment has three important practical consequences. First, AI inference can be freely composed within Compute: an action may call an LLM to classify a document, a diffusion model to verify an image hash, and a traditional database in the same phase, with all results feeding into the governance decision made in Require. Second, because Compute results are recorded in audit streams, AI inferences are auditable—a regulator or auditor can inspect exactly which model, with which inputs, produced which output for any historical action. Third, and most consequentially, the content-addressable structure of streams makes AI inference results cacheable and deduplicatable, as developed below.

12.2 Content-Addressable Memoization of AI Inference

Definition 12.1 (Compute Cache Key)**.** *For a generative model invocation with model identifier mid, input i, and hyperparameters h (temperature, seed, top-k, etc.), the* compute cache key *is:*

$$\kappa(mid, i, h) \triangleq H(mid \parallel i \parallel h)$$

where H is a collision-resistant hash function and $\|$ denotes concatenation. The result of the invocation, once computed, is stored as a stream message under the URI $(\texttt{ai-cache}, \kappa)$.

When COMPUTE encounters an AI inference call, it first checks whether $(\texttt{ai-cache}, \kappa)$ exists in the active stream set. If so, the cached result is returned immediately—the invocation is *exact* and requires no model call. Only on a cache miss is the model actually invoked; the result is then appended to $(\texttt{ai-cache}, \kappa)$ for future use.

This is a direct instantiation of the PLT framework's artifact memoization (§15): the compute cache key κ is the content address $H(f, i)$ of [20], the model identifier *mid* plays the role of the function f, and the cached stream message is the artifact $a = f(i)$. By the PLT Prior-Guided Caching Theorem (Theorem 15.2), pre-populating the cache with the model's high-probability outputs before any traffic arrives strictly dominates reactive caching during the initial operating phase.

12.3 The Determinism Spectrum

COMPUTE outputs range across a spectrum of determinism:

COMPUTE invocation type	**Determinism**	**Cache behaviour**
Pure function (arithmetic, hashing)	Exact	N/A (always fresh)
Stream/database read (fixed snapshot)	Exact	Prefix-immutable (Lemma 10.1)
AI inference, cache hit	Exact	Returns stored result
AI inference, cache miss	ε-deterministic	Result stored on first call
Truly stochastic (no seed)	Non-deterministic	Must record; breaks replay

The practically important regime is *ε-determinism*: given fixed (mid, i, h), two independent invocations of the same model on the same hardware configuration agree with probability $1 - \varepsilon$, where ε is negligible (typically $< 10^{-6}$ for deterministic inference modes with a fixed seed on the same hardware generation).

Definition 12.2 (ε-Deterministic View Capability)**.** *A view capability $C \in \mathcal{C}_{view}$ is* ε-deterministic *if for any two independent invocations with the same inputs and hyperparameters:*

$$\Pr\Big[C(mid, i, h) = C'(mid, i, h)\Big] \;\geq\; 1 - \varepsilon$$

where C and C' are independent draws from the invocation distribution (e.g. on different hardware nodes). When $\varepsilon = 0$ the capability is exactly deterministic.

In practice, ε-determinism is achieved by:

1. fixing the random seed as part of h;
2. using quantized or integer arithmetic where available; or
3. accepting the negligible ε and handling divergence via fork detection (§12.5).

12.4 Implications for Replay

Theorem 11.1 guarantees that $\mathcal{MM}$ state is reconstructible from the initial state and the sequence of external appends. AI capabilities refine this into a three-tier replay guarantee:

Corollary 12.3 (Tiered Replay Determinism)**.** *Under the memoization scheme of Definition 12.1:*

(a) ***Cache-hit replay (exact):*** *if the compute cache key κ is present at replay time, the replayed* COMPUTE *returns the identical result as the original. The replay is bit-for-bit identical.*

(b) ***Cache-miss replay (ε-exact):*** *if κ is absent (the cache was not populated), the replayed invocation agrees with the original with probability at least $1 - \varepsilon$.*

(c) ***Audit-trail replay (exact from record):*** *if the original invocation's result was recorded in the action's audit stream (as required by* COMPUTE*'s determinism note), a replay that reads from the audit stream rather than re-invoking the model is always bit-for-bit exact, regardless of cache state.*

Proof. (a) By Definition 12.1, the cached stream message is the unique result stored on the first invocation. Lemma 10.1 guarantees the cache entry is never removed. Therefore the replayed call reads the same cached value.

(b) By Definition 12.2, two independent invocations of an ε-deterministic capability with identical (mid, i, h) agree with probability $\geq 1 - \varepsilon$.

(c) The audit stream is an append-only $\mathcal{MM}$ stream (Definition 3.6). By Corollary 10.2, its entries are never removed or modified. A replay that substitutes audit-stream values for live capability calls (as in the Replay definition of §11) is therefore deterministic by construction. □ □

12.5 Probabilistic Consensus and Fork Detection

The ε-determinism of AI view capabilities has a striking implication for distributed execution: two independent nodes running the same action on the same input will, with overwhelming probability, compute the same COMPUTE output $\mathbf{y}$, the same REQUIRE declaration sets, and therefore the same EXECUTE delta. This enables a lightweight *probabilistic consensus* protocol that requires only hash comparison rather than full output replication.

Definition 12.4 (Compute Fingerprint)**.** *For an action invocation $(\alpha, \mathbf{x})$ executed on node ν, the* compute fingerprint *is:*

$$\phi_\nu(\alpha, \mathbf{x}) \triangleq H(\mathbf{y}_\nu \parallel decl_\nu \parallel \Delta_\nu)$$

where $\mathbf{y}_\nu$, $decl_\nu$, Δ_ν are the COMPUTE output, REQUIRE declaration, and EXECUTE delta produced by node ν.

Theorem 12.5 (Probabilistic Consensus via Fingerprint Comparison)**.** *Let ν_1 and ν_2 be two nodes independently executing the same invocation $(\alpha, \mathbf{x})$, where all AI capabilities used in* COMPUTE *are ε-deterministic and all other COMPUTE operations are exactly deterministic. Let k be the number of AI capability calls in a single* COMPUTE *execution, and assume the*

k calls are mutually independent *(i.e. the output of each call does not influence the inputs of any other call in a way that introduces correlation between their failure events). Then:*

$$\Pr\Big[\phi_{\nu_1}(\alpha, \mathbf{x}) = \phi_{\nu_2}(\alpha, \mathbf{x})\Big] \;\geq\; (1-\varepsilon)^k$$

For typical values $\varepsilon = 10^{-6}$ and $k \leq 10$, this probability exceeds $1 - 10^{-5}$—effectively certain. If fingerprints match, the nodes have reached consensus on the action's effect without exchanging any stream content.

Proof. Let A_j be the event that the j-th AI call agrees between ν_1 and ν_2. By Definition 12.2, $\Pr[A_j] \geq 1-\varepsilon$ for each j. By the mutual independence assumption, the k events $A_1, \ldots, A_k$ are independent. Therefore:

$$\Pr\Big[\mathbf{y}_{\nu_1} = \mathbf{y}_{\nu_2}\Big] \;\geq\; \Pr[A_1 \cap \cdots \cap A_k] \;=\; \prod_{j=1}^{k} \Pr[A_j] \;\geq\; (1-\varepsilon)^k$$

All non-AI computations in COMPUTE are exactly deterministic given the same $\mathbf{x}$ and the same stream snapshots; stream prefixes are immutable by Lemma 10.1 so both nodes read identical prefixes. Since *decl* and Δ are deterministic functions of $\mathbf{y}$ (Definitions 5.3 and 5.4), the fingerprints $\phi_{\nu_1} = \phi_{\nu_2}$ whenever $\mathbf{y}_{\nu_1} = \mathbf{y}_{\nu_2}$. Hash collision probability is $O(2^{-\lambda})$ for a λ-bit hash, negligible compared to $(1-\varepsilon)^k$, and is dominated by the stated bound. □ □

Definition 12.6 (Compute Fork)**.** *A* compute fork *occurs when two nodes executing the same invocation produce differing fingerprints:*

$$\phi_{\nu_1}(\alpha, \mathbf{x}) \neq \phi_{\nu_2}(\alpha, \mathbf{x})$$

By Theorem 12.5, this occurs with probability at most $1 - (1-\varepsilon)^k \leq k\varepsilon$, i.e. with negligible probability under typical parameters.

Remark 12.7 (Fork Detection Without Global State)**.** *Compute fork detection requires no global state. Each node broadcasts its fingerprint ϕ_ν alongside the stream URI and invocation identifier. Any node that receives two differing fingerprints for the same invocation detects a fork and may trigger a governance response (e.g. escalating to audit-trail replay, alerting operators, or applying a majority-vote rule among three or more nodes). This is strictly local: no blockchain, no global ledger, no consensus round is required for the common case. Only on an actual fork—an event with probability $\leq k\varepsilon$ per invocation—is any cross-node coordination triggered.*

12.6 Implications for Blockchain-Like Consistency

The probabilistic consensus mechanism of Theorem 12.5 yields a comparison with classical blockchain consensus worth making precise.

In a permissionless blockchain, consensus is required on *every* transaction, paying the full cost of the consensus protocol (communication, latency, energy) regardless of whether any conflict exists. This is because the shared global state is mutable and the outcome of any transaction depends on the ordering of all prior transactions.

In $\mathcal{MM}$ with ε-deterministic COMPUTE:

1. **Common case (no fork, probability $\geq (1-\varepsilon)^k$):** nodes compare fingerprints, agree in $O(1)$ messages, and proceed. No consensus round. The embarrassing parallelism of Theorem 6.3 applies in full.

2. **Rare case (fork detected, probability $\leq k\varepsilon$):** nodes escalate to audit-trail replay (Corollary 12.3(c)) or a designated tiebreaker node, resolving the fork deterministically. Only this rare case incurs coordination cost.

The expected coordination cost per invocation is therefore proportional to $k\varepsilon$—negligible in practice. This yields an *amortized consensus* result: the system behaves as if it had embarrassing parallelism for $(1 - k\varepsilon)$ of invocations and incurs coordination only for the remaining $k\varepsilon$ fraction.

Corollary 12.8 (Amortized Consensus Cost)**.** *In an $\mathcal{MM}$ instance with n concurrent invocations per time unit, each making k ε-deterministic AI calls, the expected number of invocations requiring cross-node coordination is at most $n \cdot k \cdot \varepsilon$ per time unit. All other invocations are resolved by fingerprint comparison in $O(1)$ messages, with no consensus protocol.*

Proof. By Theorem 12.5, each invocation requires coordination iff a fork occurs, which happens with probability $\leq k\varepsilon$. By linearity of expectation over n independent invocations, the expected count is $\leq n \cdot k \cdot \varepsilon$. All other invocations are resolved by fingerprint agreement. □ □

12.7 Connection to the PLT Execution Cache

The content-addressable memoization scheme of Definition 12.1 is precisely the PLT artifact store of [20] applied to AI inference. The PLT prior-guided caching theorem (Theorem 15.2) immediately applies: a cache pre-populated by sampling the model at low temperature (finding its high-probability outputs) strictly dominates a reactive cache for all query counts below threshold T_0.

In $\mathcal{MM}$ terms: before any action traffic arrives, a publisher may run the relevant AI models in sampling mode, store their high-probability outputs as streams under their compute cache keys, and thereby convert what would be ε-deterministic cache-miss invocations into exactly deterministic cache-hit invocations—raising the fingerprint agreement probability from $(1 - \varepsilon)^k$ to exactly 1 for the pre-populated inputs. The pre-population cost is bounded by the entropy of the high-probability region of the model's output distribution, and is amortized over all future invocations that hit the cache.

13 Distributed Execution Without Global Consensus

13.1 Per-Publisher Locality

Definition 13.1 (Local Action)**.** *Let $streams(p) \triangleq \{S_i \in \mathcal{S} \mid Meta_i.\texttt{owner} = id(p)\}$ denote the set of all streams owned by publisher p. Action α is* local to publisher *p if every stream in $WriteSet(\alpha) \cup ReadSet(\alpha)$ is owned by p, i.e. $WriteSet(\alpha) \cup ReadSet(\alpha) \subseteq streams(p)$.*

Proposition 13.2 (Local Linearizability)**.** *Local actions of the same publisher are linearizable without distributed locking.*

Proof. All appends to p's streams pass through p's append authority (the TRIGGER rule requires $ACL_i(a, \text{APPEND}) = \mathit{true}$, and p controls its ACLs). p can enforce a local total order on appends to each stream. Concurrent local actions on disjoint streams within p's scope are trivially non-interfering by Theorem 6.3. □ □

13.2 Cross-Publisher Interaction

Cross-publisher interaction occurs when an action reads a stream owned by a different publisher. This is modeled as a capability call: $C_{\texttt{fetchStream}}$ resolves a remote stream URI and returns its current head. The read frontier is recorded in the EXECUTE audit trail.

Because reads are non-destructive (no write to a remote stream is involved), cross-publisher reads require no coordination. The remote publisher's stream evolves independently; the reading action observes a consistent snapshot at the time of the fetch [6]. The read frontier is recorded in the action's audit stream (appended during EXECUTE) for deterministic replay.

13.3 No Global Consensus Theorem

Theorem 13.3 (Consensus Freedom)**.** *A well-formed $\mathcal{MM}$ instance executing only local actions and non-mutating cross-publisher reads requires no global consensus protocol.*

Proof. Global consensus protocols [17, 18] are required when two parties must agree on an ordering of conflicting writes to shared state. In $\mathcal{MM}$ under the stated conditions: (i) Each action is local to its publisher (Definition 13.1), so $\mathit{WriteSet}(\alpha) \subseteq \mathit{streams}(p)$. Since stream ownership is unique (Definition 3.8), no two publishers share any writable stream; write conflicts across publishers cannot arise. (ii) Local actions of the same publisher are linearized by that publisher without external coordination (Proposition 13.2); no cross-publisher agreement is needed. (iii) Cross-publisher reads via $C_{\texttt{fetchStream}}$ are non-destructive—they observe a snapshot but write nothing—so they cannot create a write conflict requiring global ordering. Since no write conflict across publishers is possible and all intra-publisher ordering is local, no global consensus protocol is required. □ □

13.4 Consistency Model and the CAP Theorem

$\mathcal{MM}$ provides *per-stream sequential consistency* [15]: all appends to S_i are totally ordered by the publisher. Cross-stream consistency is *causal*: established by Theorem 7.3.

We now state precisely how $\mathcal{MM}$ relates to the CAP theorem [3, 10], which states that no distributed system can simultaneously guarantee Consistency, Availability, and Partition-tolerance.

Theorem 13.4 (CAP Classification of $\mathcal{MM}$)**.** *Under the standard ownership model:*

(a) ***Per-publisher (AP):*** *During a network partition, a publisher p continues to accept local appends to its own streams (availability), but observers on the other side of the partition cannot see those appends (sacrificing cross-partition consistency). Causal consistency is restored when the partition heals.*

(b) ***Cross-publisher with optional CP:*** *When an application requires linearizability across two publishers p and p' (e.g. an atomic two-party transfer), this can be achieved by modeling the joint operation as a capability in* EXECUTE *that implements a two-phase commit or consensus sub-protocol. This sacrifices availability during partition for that specific cross-publisher operation, but does not affect any other stream.*

Proof. (a) Under a network partition separating publisher p from a set of observers, the TRIGGER rule can still fire locally at p: the preconditions $ACL_i(a, \text{APPEND})$ and the timestamp check are evaluated locally. p's streams grow monotonically and observers will receive the missing suffixes when the partition heals (Lemma 10.1 ensures no history is lost). $\mathcal{MM}$ therefore chooses Availability and Partition-tolerance (AP) at the per-publisher level, at the cost of eventual (not immediate) cross-publisher consistency.

(b) If strong cross-publisher consistency is required, an action declares a consensus capability $C_{2\text{PC}} \in \mathit{CapSet}$ in REQUIRE. During EXECUTE, $C_{2\text{PC}}$ implements a two-phase commit or equivalent consensus sub-protocol [17] across the publishers in $\mathit{WriteSet}(\alpha)$. Under a network partition there are two outcomes: (1) the protocol blocks until the partition heals, sacrificing availability for this action; or (2) the protocol aborts, returning a failure result, in which case EXECUTE produces no delta (the action has no effect) and causal consistency is trivially preserved. In either case, streams *outside* $\mathit{WriteSet}(\alpha)$ are unaffected: by the EXECUTE definition, appends occur only to declared write streams. Therefore the CP guarantee is scoped exactly to the streams in $\mathit{WriteSet}(\alpha)$, and all other streams retain their AP semantics from part (a). □ □

Remark 13.5. *The per-publisher AP / optional-CP decomposition means $\mathcal{MM}$ does not make a global CAP choice. The tradeoff is made at action granularity, not system granularity. This is a consequence of stream ownership partitioning: there is no single shared state over which a system-wide CAP choice must be made.*

14 HTTP Compatibility and Deployment

14.1 Streams over Standard Protocols

$\mathcal{MM}$ requires no new transport protocol. All stream operations map onto standard HTTP/WebSocket/SSE primitives:

HTTP Endpoint	Semantics
`GET /s/{owner}/{name}`	Current stream head (messages, count, hash)
`GET /s/{owner}/{name}?from=k`	Incremental update $[k, \mathit{head}]$
`POST /s/{owner}/{name}`	Append (policy-checked, signed)
`WS /s/{owner}/{name}/sub`	Push subscription

The `POST` endpoint enforces the TRIGGER rule: it validates the actor's signature against the ACL and the timestamp ordering before accepting the message.

14.2 CDN Caching

By Lemma 10.1, all stream prefixes are immutable. A prefix identified by its terminal hash is permanently cacheable with `Cache-Control: immutable`. CDNs may cache arbitrarily many historical stream prefixes. Only the "head" endpoint is dynamic. This reduces origin load proportionally to the number of reads against historical data.

14.3 Subscriptions as SSE

The `sub` endpoint delivers incremental updates as Server-Sent Events: the server emits only the suffix $M_i[k+1..\mathit{head}]$ whenever new messages arrive. Clients maintain their own cache entry $(k_a, M_i[1..k_a])$ and apply each received suffix. No polling occurs on either side.

14.4 Relation Events over Standard REST

Relation events (`relateTo`, `relateFrom`, etc.) are operations on the R tables, typically issued alongside a `POST` that appends a message. This maps naturally onto federated social protocols such as ActivityPub [28], where social graph edges (follow, like, announce) are declared alongside actor activity messages. The induced relation graph (§3.4) is served on demand by a projection query:

```
GET /graph?streams={id_1,...,id_k}&type={tau}&at={T}
```

which scans the specified streams up to time T and returns G^T filtered by type τ.

15 Connection to Probabilistic Language Tries

The Probabilistic Language Trie (PLT) framework [20] and the Magarshak Machine share the same author and are designed as complementary formal systems addressing different levels of abstraction. This section makes the formal relationship precise.

15.1 PLT Execution Traces as MM Streams

In the PLT framework, the sequence of tool or model invocations in an agentic system forms an *execution trace*:

$$e = ((f_1, i_1, a_1),\ (f_2, i_2, a_2),\ \ldots,\ (f_n, i_n, a_n))$$

where f_k is a function (model or tool), i_k is its input, and $a_k = f_k(i_k)$ is its output artifact.

Proposition 15.1 (PLT Traces as MM Streams)**.** *An execution trace e in the PLT framework embeds as an $\mathcal{MM}$ stream S_{exec} in which:*

- *Each triple (f_k, i_k, a_k) is a message $m_k = (k, \texttt{invoke}, (f_k, i_k, a_k))$ appended to S_{exec};*

- *The PLT's content address $h = H(f, i)$ is the $\mathcal{MM}$ stream URI $(\texttt{publisherId}, f{:}i)$, with the artifact a_k stored as the head of a stream S_h;*
- *The PLT's reuse probability $P_{\mathcal{M}}(i)$ is the prior probability that action invocation $(\alpha, \mathbf{x})$ appears in pending at the next step.*

Proof. Each step is a direct correspondence. The append-only property of $\mathcal{MM}$ streams (Definition 3.6) matches the PLT's requirement that execution traces are immutable histories. The $\mathcal{MM}$ stream URI $(\texttt{publisherId}, \texttt{name})$ serves the same role as the content address $H(f, i)$: it uniquely identifies a cached artifact. The PLT's artifact determinism requirement ($a = f(i)$ is deterministic given (f, i)) corresponds to $\mathcal{MM}$'s COMPUTE phase being deterministic for cached invocations (Corollary 12.3(a)) and ε-deterministic for cache-miss AI invocations (Definition 12.2). □ □

15.2 What MM Adds to PLT: Governance and Ownership

The PLT framework does not model *who owns* an artifact, *what policy* governs its creation, or *which actors* may read it. These are $\mathcal{MM}$'s contributions:

1. **Ownership.** In $\mathcal{MM}$, every stream — including execution trace streams and artifact streams — has a cryptographic owner (Definition 3.8). Artifact reuse across publishers requires explicit delegation (Definition 3.9). This makes the PLT's "cross-model reuse" scenario [20] formally governed: reuse is not implicit but requires the original publisher to grant read rights.

2. **Policy.** The PLT's Bayesian retention policy $V(a) = \hat{p}(a) \cdot C_c - C_s$ (evict if expected value is negative) is an economic policy over artifacts. In $\mathcal{MM}$, this maps directly to the Policy function $\mathcal{P}$: a publisher can encode the retention threshold as a policy rule evaluated at each CREATE step for new artifact streams.

3. **Relational structure.** The PLT records that artifact a_k was produced from input i_k but does not reify the relationship between a_k and its dependencies. In $\mathcal{MM}$, the action that produced a_k posts a $\texttt{relateTo}(S_{input}, \texttt{derivedFrom}, 1.0)$ message to the artifact stream, causing the infrastructure to write $(S_{artifact}, S_{input}, \texttt{derivedFrom}, 1.0)$ into *relatedTo*, making the dependency graph efficiently queryable (Definition 3.10).

4. **Push-only execution.** The PLT framework does not specify when reuse is triggered — it provides the caching policy but not the scheduling model. $\mathcal{MM}$'s push-only execution (§7) fills this gap: artifact retrieval is triggered reactively when a downstream action's COMPUTE phase discovers a cache hit, not by periodic polling of the artifact store.

15.3 What PLT Adds to MM: Quantitative Caching Theory

$\mathcal{MM}$'s caching results (§10) are structural: they establish *which* caches need invalidation, derived from the dependency graph D. The PLT framework adds quantitative depth.

Theorem 15.2 (PLT Caching Advantage, imported)**.** *Let requests to an $\mathcal{MM}$ publisher arrive i.i.d. from a distribution $P_{\mathcal{M}}$ over action invocations, with the optimal cache $\mathcal{C}^* = \{(\alpha_1, \mathbf{x}_1), \ldots, (\alpha_K, \mathbf{x}_K)\}$ being the K most probable invocations, ranked $p_1 \geq p_2 \geq \cdots \geq p_M > 0$. Let $\delta_p = p_K - p_{K+1} > 0$ denote the* probability gap *at the cache boundary (distinct from Δ, the execute-phase delta of Definition 5.4), and let $\rho = C_c - C_l > 0$ be the per-invocation savings from a cache hit. Then for all query counts $T \leq T_0$ where*

$$T_0(\delta) = \frac{\ln(K(M-K)/\delta)}{2\delta_p^2} + \frac{\ln K}{p_K},$$

a prior-guided cache (initialized with $\mathcal{C}^*$ from $P_{\mathcal{M}}$) achieves expected cost at least $\frac{\delta}{2} \cdot \delta_p \cdot \rho$ lower than any LFU cache, with probability at least $1 - \delta$.

Proof. This follows from Theorem 1 of [20] applied to the $\mathcal{MM}$ setting, with action invocations $(\alpha, \mathbf{x})$ playing the role of PLT inputs and EXECUTE cost C_c playing the role of the computation cost. The $\mathcal{MM}$ structure ensures the conditions of the theorem: artifacts are content-addressed by stream URI; the prior $P_{\mathcal{M}}$ is available from the action type's subscription weights; and invocations are deterministic functions of their inputs when COMPUTE uses only pure computations or cached AI results (cache-hit invocations are exactly deterministic by Corollary 12.3(a)). For cache-miss AI invocations, COMPUTE is ε-deterministic (Definition 12.2); the PLT theorem applies in the limit $\varepsilon \to 0$, and the ε-correction to the advantage bound is $O(\varepsilon)$, negligible for standard parameters. □ □

Corollary 15.3 (Entropy-Dependent Publisher Advantage)**.** *For a publisher whose action distribution follows Zipf(α_z) with $p_j \propto j^{-\alpha_z}$, the probability gap satisfies $\delta_p \approx C\alpha_z K^{-\alpha_z - 1}$ for large K, giving a cache advantage window $T_0 = O(K^{2\alpha_z+2}\alpha_z^{-2} \ln K)$. When the publisher's action distribution is highly concentrated (large α_z), the advantage window is small but the per-hit savings $\delta_p \cdot \rho$ are large. When the distribution is near-uniform (small α_z, high entropy), the advantage window is unbounded: an empirical cache can never reliably identify the optimal cache contents, and the prior-guided cache maintains its advantage indefinitely.*

Corollary 15.3 has a direct practical consequence for $\mathcal{MM}$ deployments: publishers with *highly predictable* action patterns (federated social platforms, financial systems, content delivery) benefit most from prior-guided cache initialization, as their action distribution is concentrated and the cache advantage is realized quickly. Publishers with *unpredictable* patterns (research agents, anomaly-detection systems) have high execution entropy and benefit most from the structural invalidation guarantee of Theorem 10.6 rather than from prior initialization.

15.4 Execution Entropy and MM Replay

The PLT framework introduces *execution entropy* $H(P_{\text{exec}})$ as a measure of system complexity: a system with low execution entropy repeatedly invokes the same tools on the same inputs and is highly compressible; a system with high entropy performs mostly novel computations.

This concept strengthens $\mathcal{MM}$'s replay semantics (§11). State Reconstruction (Theorem 11.1) guarantees that any $\mathcal{MM}$ instance is replayable; the PLT framework adds that

the *cost* of replay depends on execution entropy. A low-entropy $\mathcal{MM}$ instance can be replayed largely from cached artifacts (cheap); a high-entropy instance must re-execute most actions from capability calls (expensive). The execution entropy $H(P_{\text{exec}})$ is thus the natural complexity measure for $\mathcal{MM}$ replay cost, providing the quantitative complement to the qualitative determinism guarantee.

16 Comparative Analysis

Table 1: Feature comparison across computational and distributed models.

Feature	**Turing**	**von Neumann**	**Actor**	**Blockchain**	**PLT**	**Magarshak**
State model	Tape (mutable)	Shared mutable	Actor-local mutable	Append-only global log	Implicit (model weights)	Append-only streams
Relations	None	Pointers	Implicit message graph	None formal	Trie edges (probabilities)	Bidirectional index tables (*relatedTo*/*relatedFrom*
Actor identity	None	OS process	Address/PID	Public key	None	Public key (cryptographic)
Execution	Sequential	Sequential	Concurrent async	Serialized (consensus)	Query-driven	Embarrassingly parallel
Side effects	None	Implicit	Implicit	Explicit (gas)	Implicit	Phase-isolated (EXECUTE only)
Policy	None	OS-layer	None	Smart contracts	None	First-class (REQUIRE)
Caching	N/A	External	None	None formal	Prior-guided (quantitative)	Structural (dependency graph)
Ownership	None	OS process	None	Public key	None	Cryptographic (per stream)
Push/pull	N/A	Pull	Push	Pull	Pull (query-driven)	Push-only
Replay	No	Limited	No	Full (global)	Partial (artifact reuse)	Full (per stream)

16.1 Turing Machine

The Turing Machine defines *computability*. Individual $\mathcal{MM}$ actions may be Turing-complete programs; $\mathcal{MM}$ does not compete with the Turing Machine on this axis. $\mathcal{MM}$ addresses a different question: not "what can be computed" but "how should a network of actors safely co-evolve their shared state."

16.2 von Neumann Architecture

The von Neumann model defines how a single program executes. Its shared-mutable-memory abstraction is the root of concurrency hazards. $\mathcal{MM}$ replaces shared memory with append-only streams; the analog of a register write is a stream append—a monotone, non-destructive operation.

16.3 Actor Model

The Actor Model is the classical model most aligned with $\mathcal{MM}$ in spirit. Key differences: (i) Actor state is mutable; $\mathcal{MM}$ state is append-only. (ii) Actor relationships are implicit in message-passing patterns; $\mathcal{MM}$ reifies them as relation events in streams. (iii) The Actor Model has no policy primitive; $\mathcal{MM}$'s policy function is first-class. (iv) The Actor Model has no phase calculus; $\mathcal{MM}$'s VALIDATE–CALL sequence enforces separation of concerns.

16.4 Blockchains

Blockchains achieve append-only global state under adversarial conditions via consensus. $\mathcal{MM}$ shares the append-only commitment but replaces global consensus with per-publisher sequential consistency, yielding embarrassing parallelism across publishers. As proved in §6, this is a structural consequence of declared write sets—a mechanism absent from all existing blockchain designs except Solana's partial account declaration.

16.5 Process Calculi

The π-calculus [22] and CSP [12] are formal process calculi for concurrent communicating systems. They model communication channels as primitives and provide bisimulation-based equivalence. $\mathcal{MM}$ differs in that: (i) channels ($\approx$ streams) are persistent and append-only, not transient; (ii) channel access is policy-governed; (iii) the phase calculus imposes structure on computation within a process that π/CSP do not. $\mathcal{MM}$ can be given a translation into π-calculus by mapping each stream to an output-only channel and each subscription to an input process, but the translation would lose the append-only and policy properties.

16.6 Probabilistic Language Tries

The PLT framework [20] and $\mathcal{MM}$ operate at complementary levels of abstraction. PLTs provide a quantitative caching theory (how much better prior-guided caching is relative to empirical frequency caching, as a function of distribution entropy) and a four-tier computation spectrum routing inputs by code length. $\mathcal{MM}$ provides the governance layer that

PLTs lack: ownership, policy, relational structure, and push-only scheduling. Together, the two frameworks form a complete theory of *governed artifact reuse*: PLTs determine what to cache and when to evict; $\mathcal{MM}$ determines who may access the cache, what policy governs writes to it, and how downstream computations are triggered reactively when cache content changes. Section 15 establishes the formal embedding and imported theorems.

17 Discussion

17.1 What Has Been Proved vs. Asserted

It is worth distinguishing the paper's proven results from its design choices.
Derived theorems from the operational semantics:

- Structural Enabling for Embarrassing Parallelism (Theorem 6.3)
- Publisher-Proportional Parallelism (Corollary 6.4)
- Efficient Conflict-Free Scheduling (Corollary 6.5)
- Causal Consistency of Reactive Execution (Theorem 7.3)
- Reactive Completeness (Theorem 7.5)
- Polling-Freedom (Proposition 7.2)
- Ripple Termination (Corollary 8.6)
- Local Deduplication Sufficiency (Proposition 8.4)
- Governance Without Content Access (Proposition 9.1)
- Prefix Immutability (Lemma 10.1)
- Append-Only Safety (Corollary 10.2)
- Monotone Stream Universe (Corollary 10.3)
- Minimal Cache Invalidation (Theorem 10.6)
- State Reconstruction (Theorem 11.1)
- Tiered Replay Determinism (Corollary 12.3)
- Probabilistic Consensus via Fingerprint Comparison (Theorem 12.5)
- Amortized Consensus Cost (Corollary 12.8)
- Replay Determinism (Corollary 11.3)
- Local Linearizability (Proposition 13.2)
- Consensus Freedom (Theorem 13.3)
- CAP Classification (Theorem 13.4)
- Vector Clock Embedding (Proposition 3.13)

Imported results from the PLT framework [20]: PLT Caching Advantage (Theorem 15.2) and Entropy-Dependent Publisher Advantage (Corollary 15.3). These provide the quantitative caching theory that $\mathcal{MM}$'s structural results do not.
Design choices whose consequences are explored: the append-only constraint, the five-phase calculus, the push-only execution model, and the cryptographic actor identity.

17.2 Deliberate Limitations

$\mathcal{MM}$ intentionally excludes:

- **Global mutable state.** All state lives in owned streams.

- **Polling.** All scheduling is push-based.
- **Implicit side effects.** All external effects are in EXECUTE.
- **Ultra-low-latency inner loops.** The phase overhead makes $\mathcal{MM}$ unsuitable as a model for tight numerical loops; actions model business-level operations, not instruction-level computation.

These are features: they are precisely what enable the proved theorems.

17.3 Open Problems

1. **Schema evolution.** How should stream schemas evolve while preserving backward compatibility for subscribers?
2. **Policy composition.** When multiple policies govern the same stream (owner policy, publisher policy, regulatory overlay), what are the conflict resolution rules and their soundness conditions?
3. **Liveness.** Under what scheduling assumptions does every enqueued invocation eventually execute? This requires a fairness condition on the scheduler that is left as future work.
4. **Cryptographic hardening.** Under what assumptions (hash collision resistance, append-only storage) do Merkle-chained streams provide cryptographic tamper evidence against a computationally bounded adversary?
5. **Formal verification.** Can $\mathcal{MM}$ instance properties be automatically verified in a proof assistant (e.g. Coq, Lean)? The operational semantics defined here is a suitable starting point.
6. **Performance bounds.** What are the throughput and latency characteristics of $\mathcal{MM}$ implementations under realistic workloads, and how do they compare to Kafka, Flink, and Solana?

17.4 Relation to Deployed Systems

The constructions in this paper are not purely theoretical. The Qbix platform and the Intercoin community currency network implement instantiations of the $\mathcal{MM}$ model: the phase calculus corresponds to the `defineTool` API in the Qbix Q-framework; stream-based relations underlie the Groups social platform's federated data model; the embarrassing parallelism property informs SafeBox's multi-tenant execution isolation architecture. A direct formal instantiation of $\mathcal{MM}$ as a decentralised economic network is developed in the companion paper Intercloud [19], which builds chilling-effect consensus and value-proportional Watcher swarms on top of the stream and ripple-deduplication primitives defined here. A forthcoming implementation paper will report on performance measurements and empirical validation of the derived theorems under realistic workloads.

17.5 Conclusion

The Magarshak Machine defines a new position in the landscape of computational formalisms through the **SPACER** framework:

- **Streams** (S): append-only, publisher-owned state logs serving as governance and metadata layers over (potentially encrypted) content;
- **Policy** (P): a first-class governance function enforced in REQUIRE with retry-loop semantics, operating on stream metadata without requiring access to content;
- **Actions** (A): computation units structured into the five-phase calculus VALIDATE→COMPUTE→REC with COMPUTE performing idempotent remote fetches, EXECUTE writing only local state, and CALL emitting events to subscribers;
- **Capabilities** (C): explicitly declared view and mutating interfaces, isolating all external I/O to declared phases;
- **Execution** (E): a push-only reactive engine with ripple deduplication, pre-set interaction rails, and no polling; and
- **Relations** (R): typed append-only event streams inducing a dynamic directed graph that generalizes vector clocks, with no separate infrastructure required.

From these, we derived: embarrassing parallelism scaling linearly with publisher count, causal consistency, minimal cache invalidation, deterministic replay, consensus freedom, and a precise AP/CP CAP classification.

Final Statement. The Turing Machine defines *what* can be computed. The von Neumann architecture defines *how* programs execute on hardware. The Magarshak Machine—through its SPACER framework of Streams, Policy, Actions, Capabilities, Execution, and Relations—defines *how distributed reactive systems evolve safely, transparently, and composably over time*: in parallel, without global consensus, with governance over metadata even when content is opaque, and with bounded, locally-deduplicated event propagation.

Acknowledgements

The author thanks the open-source communities behind Apache Kafka, Apache Flink, ActivityPub, the AT Protocol, and Solana, whose practical systems informed many of the abstractions formalized here. Thanks also to the teams behind the Qbix, Intercoin, and SafeBox projects for stress-testing these ideas in production.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986. ISBN 978-0-262-01092-4.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust (POST)*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017. doi: 10.1007/978-3-662-54455-6_8.

[3] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 7, 2000. doi: 10.1145/343477.343502. Invited keynote.

[4] Vitalik Buterin. A next-generation smart contract and decentralized application platform. Ethereum Foundation White Paper, https://ethereum.org/en/whitepaper/, 2014.

[5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.

[6] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985. doi: 10.1145/214451.214456.

[7] DFINITY Foundation. The Internet Computer for geeks. Cryptology ePrint Archive, Report 2022/087, https://eprint.iacr.org/2022/087, 2022.

[8] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988.

[9] Martin Fowler. Event sourcing. *martinfowler.com*, 2005. URL https://martinfowler.com/eaaDev/EventSourcing.html. Accessed 2024.

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002. doi: 10.1145/564585.564601.

[11] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, 1973.

[12] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21 (8):666–677, 1978. doi: 10.1145/359576.359585.

[13] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB Workshop at ACM SIGMOD*, 2011.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi: 10.1145/359545.359563.

[15] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439.

[16] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994. doi: 10.1145/177492.177726.

[17] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. doi: 10.1145/279227.279229.

[18] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[19] Gregory Magarshak. Intercloud: Eventual consistency for decentralised economies via chilling-effect consensus. Preprint, 2026. Companion paper to the Magarshak Machine.

[20] Gregory Magarshak. Probabilistic language tries: A unified framework for compression, decision policies, and execution reuse, 2026. URL https://arxiv.org/abs/2604.06228. Submitted 29 March 2026.

[21] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.

[22] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992. doi: 10.1016/0890-5401(92)90008-4.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[24] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. doi: 10.1007/978-3-642-24550-3_29.

[25] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. doi: 10.1112/plms/s2-42.1.230.

[26] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013. ISBN 978-0-321-83457-7.

[27] John von Neumann. First draft of a report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[28] Christopher Webber, Jessica Tallon, Evan Shepherd, Amy Guy, and Evan Prodromou. ActivityPub. W3C Recommendation, https://www.w3.org/TR/activitypub/, January 2018.

[29] Anatoly Yakovenko. Sealevel: Parallel processing thousands of smart contracts. Solana Labs Technical Blog, https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192, 2019.