# Pure Computation and Governed Mutation:

# Capability Transforms and Side Effects in Protocol-Governed Systems

**Bachi (aka Bhash Ganti)** Contact: bachipeachy@gmail.com

---

## Abstract

This paper formalizes the architectural separation between pure computation and governed mutation in protocol-governed systems. Building on the execution model defined in Paper 6 [Bachi, 2026f], it specifies the semantic boundary between Capability Transforms (CT_) and Capability Side Effects (CS_).

We define purity invariants for transforms, mutation declaration requirements for side effects, ordering guarantees, replay constraints, and mutation surface bounding. This separation is shown to be necessary for deterministic replay, trace completeness, and structural enforceability.

This paper does not analyze security implications (Paper 8), execution engine architecture (Paper 6), governance mechanics (Paper 4), or lifecycle economics (Paper 9). Its purpose is definitional: to formalize the world-interaction boundary that enables determinism and auditability guarantees.

**Categories:** Software Engineering (cs.SE), Programming Languages (cs.PL), Software Architecture

**Keywords:** pure computation, side effects, capability transforms, mutation boundary, determinism, replay safety, world interaction, functional purity

---

## 1. Position in the Series

Paper 5 defined what protocol artifacts mean semantically [Bachi, 2026e]. Paper 6 established how the execution engine enforces protocol law through deterministic DAG execution and trace emission [Bachi, 2026f].

This paper addresses a fundamental question:

> *Which operations compute, and which operations change the world?*

The answer involves a constitutional separation between two categories of capability:

- **CT_** — Capability Transforms: pure computation
- **CS_** — Capability Side Effects: governed mutation

This separation is not a stylistic preference or an optimization opportunity. It is architectural law. The distinction between computation and mutation determines what can be replayed, what must be traced, and what constitutes the system's world-interaction boundary.

The relationship between papers establishes the semantic model:

$$\text{Execution (Paper 6)} \xrightarrow{\text{invokes}} \text{CT\_ and CS\_} \xrightarrow{\text{bounded by}} \text{Mutation Surface (this paper)}$$

Execution is mechanical. The computation-mutation boundary is constitutional.

---

## 2. The Computation-Mutation Problem

In conventional software systems, functions routinely intermingle computation and mutation. A single method may read input, compute derived values, update a database, call an external API, and return a result. This interleaving creates fundamental difficulties for system properties that protocol-governed systems require.

### 2.1 Problems with Interleaved Mutation

When computation and mutation are interleaved:

| Problem | Description |
|---|---|
| **Replay Impossibility** | Re-executing a function may produce different results or duplicate side effects |
| **Trace Incompleteness** | Observers cannot distinguish computation from state change |
| **Conformance Ambiguity** | Testing cannot isolate behavioral equivalence from world state |
| **Audit Opacity** | Determining what changed requires reconstructing execution |
| **Determinism Violation** | Identical inputs may produce different outputs |

These problems are not bugs to be fixed; they are structural consequences of architectural ambiguity.

### 2.2 The Separation Requirement

Protocol-governed systems impose explicit separation:

$$\text{Capability} = \text{CT\_ (pure)} \cup \text{CS\_ (mutative)}$$

$$\text{CT\_} \cap \text{CS\_} = \emptyset$$

Every capability is classified as exactly one type. No capability may be both pure and mutative. No capability may be ambiguously classified.

This separation is constitutional, not advisory. The architecture enforces classification through concern prefixes, and execution routes capabilities differently based on classification.

### 2.3 Formal Problem Statement

Let $W$ represent mutable world state (persistent storage, external systems, registries, network endpoints).

The computation-mutation problem is:

> Given an execution trace, can we determine exactly which operations affected $W$ and which operations computed values without affecting $W$?

Without explicit separation, this question is unanswerable in general. With explicit separation, it is trivially answerable: CS_ operations affect $W$; CT_ operations do not.

---

## 3. Capability Transforms (CT_)

Capability Transforms are pure computational operations. They accept input, produce output, and do nothing else.

### 3.1 Definition

A Capability Transform is a pure function:

$$\mathrm{CT} : \mathrm{Input} \rightarrow \mathrm{Output}$$

Subject to the following constraints:

- No side effects (no modification of external state)
- No hidden state access (no reading of mutable external state)
- Deterministic output (identical inputs produce identical outputs)
- No world interaction (no I/O, no network, no persistence)

Transforms compute. They do not change the world.

### 3.2 Purity Invariants

The following invariants govern all Capability Transforms:

**I-M1 (Side-Effect Freedom)** CT_ artifacts may not perform mutation. No transform may write to storage, send network requests, or modify any state outside its return value.

$$\forall \mathrm{CT}_t, W : \mathrm{Execute}(\mathrm{CT}_t) \Rightarrow W' = W$$

World state before and after transform execution is identical.

**I-M2 (Input Determinism)** Identical input produces identical output. Given the same input values, a transform must return the same result every time.

$$\forall \mathrm{CT}_t, i : \mathrm{CT}_t(i) = \mathrm{CT}_t(i)$$

This invariant holds across time, across execution contexts, and across execution engines.

**I-M3 (No Ambient State)** CT_ may not access mutable external state. Transforms cannot read from databases, check system time, generate random numbers, or access any state not explicitly provided as input.

$$\mathrm{Dependencies}(\mathrm{CT}_t) \subseteq \mathrm{Input}(\mathrm{CT}_t)$$

All values a transform depends upon must arrive through its declared input interface.

**I-M4 (Replay Equivalence)** Re-execution must yield identical trace evidence. If a transform is executed twice with the same input, both executions produce identical output and identical trace events.

$$\mathrm{Trace}(\mathrm{CT}_t(i), \mathrm{run}_1) = \mathrm{Trace}(\mathrm{CT}_t(i), \mathrm{run}_2)$$

These four invariants are not design guidelines. They are structural requirements enforced by the architecture. A capability that violates any purity invariant is misclassified and should carry the CS_ prefix instead.

### 3.3 Examples of Pure Transforms

| Transform | Input | Output | Why Pure |
|---|---|---|---|
| Hash computation | byte sequence | hash value | Deterministic mathematical operation |
| Schema validation | data, schema | validation result | Structural comparison |

| Transform | Input | Output | Why Pure |
|---|---|---|---|
| JSON path extraction | document, path | extracted value | Read-only traversal |
| Signature verification | message, signature, key | boolean | Cryptographic computation |
| Data transformation | source format | target format | Structural mapping |

Each example computes a result from input without accessing or modifying external state.

### 3.4 Trace Implications

Each CT_ invocation produces specific trace evidence:

| Event | Content | Timing |
|---|---|---|
| `transform_start` | Transform code, input hash | Before computation |
| `transform_end` | Transform code, output hash, result status | After computation |

Notably absent from transform traces:

- Mutation events
- State change records
- Before/after snapshots

Transforms are observable but non-mutative. The trace proves what was computed; it does not record state changes because there are none.

### 3.5 Cacheability and Memoization

Purity invariants enable implementation optimizations:

**Cacheability:** Because identical inputs produce identical outputs, transform results can be cached. Cache lookup may replace re-execution without behavioral change.

$$\text{Cache}(\text{CT}_t, i) \equiv \text{Execute}(\text{CT}_t, i)$$

**Memoization:** Frequently invoked transforms with common inputs may maintain result tables. Memoization is semantically invisible.

**Parallel Execution:** Independent transforms may execute in parallel. Purity guarantees no interaction between concurrent transforms.

These optimizations are implementation choices, not semantic changes. A conforming execution engine may apply them freely because purity invariants guarantee behavioral equivalence.

---

## 4. Capability Side Effects (CS_)

Capability Side Effects are declared mutation operations. They interact with the world.

### 4.1 Definition

A Capability Side Effect is a mutation operation:

$$\text{CS} : (\text{Input}, \text{State}) \to \text{State}'$$

Side effects transform world state. They may:

- Write to persistent storage
- Modify registries
- Send network requests
- Update external systems
- Append to logs

Unlike transforms, side effects have observable impact beyond their return value.

### 4.2 Types of Side Effects

Protocol-governed systems recognize categories of side effects:

| Category | Description | Examples |
|---|---|---|
| **CS_MUTABLE_** | Read-write persistent storage | Key-value stores, document databases |
| **CS_APPENDONLY_** | Append-only storage | Ledgers, audit logs, event streams |
| **CS_REGISTRY_** | Registry operations | Actor registration, capability registration |
| **CS_EXTERNAL_** | External system interaction | API calls, message queues |

Each category carries specific ordering and rollback semantics appropriate to its mutation type.

### 4.3 Mutation Invariants

The following invariants govern all Capability Side Effects:

**I-M5 (Declared Mutation)** All mutations must be declared in capability contracts. No side effect may perform mutation outside its declared scope.

$$\text{ActualMutation}(\text{CS}_s) \subseteq \text{DeclaredMutation}(\text{CS}_s)$$

The contract specifies what the side effect may mutate. The implementation may not exceed this scope.

**I-M6 (Ordering Preservation)** CS_ execution order must match workflow declaration. If a workflow declares side effect A before side effect B, execution must perform A before B.

$$\text{Declared}(A \prec B) \Rightarrow \text{Executed}(A \prec B)$$

Ordering is not a suggestion. It is a constitutional requirement enforced by the execution engine.

**I-M7 (Trace Before/After)** Mutation must emit trace evidence before and after state change. The trace records both the intent to mutate and the result of mutation.

$$\text{Execute}(\text{CS}_s) \Rightarrow \text{Traced}(\text{before}) \wedge \text{Traced}(\text{after})$$

This enables audit reconstruction without requiring state snapshots.

**I-M8 (Bounded Scope)** CS_ may only mutate declared state domains. A side effect declared to write to storage A may not write to storage B.

$$\text{Scope}(\text{CS}_s) = \text{DeclaredScope}(\text{CS}_s)$$

Scope bounding prevents mutation leakage—unauthorized state changes that occur outside declared boundaries.

### 4.4 Trace Requirements for Side Effects

Each CS_ invocation produces comprehensive trace evidence:

| Event | Content | Timing |
|---|---|---|
| `side_effect_start` | Side effect code, input, declared scope | Before mutation |
| `mutation_intent` | State domain, operation type | Before mutation |
| `mutation_result` | Success/failure, affected keys | After mutation |
| `side_effect_end` | Side effect code, result status | After completion |

The trace record enables:

- **Audit:** What mutations occurred and when
- **Replay verification:** Compare declared mutations to actual
- **Rollback analysis:** Determine scope of recovery needed

### 4.5 Non-Determinism Containment

Side effects may produce non-deterministic results. External systems change state independently; network operations may fail; concurrent access creates race conditions.

This non-determinism is:

1. **Contained:** Only CS_ prefixed operations may be non-deterministic
2. **Declared:** Contracts specify possible outcomes
3. **Traced:** Actual outcomes are recorded

The architecture does not prevent non-determinism in side effects. It contains non-determinism within declared boundaries and records actual outcomes for replay verification.

---

## 5. The World-Interaction Boundary

The CT_/CS_ separation defines a formal boundary between computation and world interaction.

### 5.1 Formal Definition

Let $W$ represent mutable world state—the totality of persistent storage, external systems, and shared registries accessible to the system.

Define the world-interaction boundary:

$$\text{CT\_} \cap W = \emptyset$$

Transforms operate entirely outside world state. They may not read from $W$ and may not write to $W$.

$$\text{CS\_} \subseteq W \to W$$

Side effects operate on world state. They may read from $W$ and write to $W$.

### 5.2 Boundary Properties

The world-interaction boundary enables critical system properties:

**Property 1: Deterministic Replay**

Replay requires re-executing CT_ but only verifying CS_ against trace.

$$\text{Replay} : \text{Re-execute}(\text{CT\_}) \land \text{Verify}(\text{CS\_}, \text{Trace})$$

Transforms can be re-executed because they produce identical results. Side effects cannot be re-executed (doing so would duplicate mutations) but can be verified against recorded outcomes.

**Property 2: Clear Audit Boundary**

All world changes occur through CS_ operations. The audit boundary is:

$$\text{Audit} = \bigcup \text{Trace}(\text{CS\_})$$

To audit what the system changed, examine CS_ traces. CT_ traces show computation; CS_ traces show mutation.

**Property 3: Replaceable Computation Layer**

Because CT_ operations are pure, they can be reimplemented without behavioral change:

$$\text{CT}_{v1}(i) = \text{CT}_{v2}(i) \Rightarrow \text{Equivalent}(\text{CT}_{v1}, \text{CT}_{v2})$$

Optimized implementations, alternative algorithms, or hardware acceleration may replace transform implementations. Purity guarantees behavioral equivalence.

### 5.3 Boundary Invariant

**I-M9 (World-Interaction Boundary)** CT_ operations do not interact with world state. CS_ operations are the exclusive mechanism for world interaction.

$$\text{WorldInteraction}(op) \Rightarrow op \in \text{CS\_}$$

This invariant is constitutional. There is no mechanism for a CT_ operation to interact with the world because the execution routing prevents it.

---

## 6. Replay Semantics Under Mutation

Replay is a fundamental capability in protocol-governed systems. It enables debugging, verification, and audit. The CT_/CS_ separation makes replay semantically coherent.

### 6.1 Replay Model

Given an execution trace $T$, replay operates as follows:

1. **Re-execute CT_:** Run all transforms with captured inputs
2. **Validate CT_ outputs:** Compare results to recorded outputs
3. **Validate CS_ ordering:** Verify side effects occurred in declared order
4. **Validate CS_ outcomes:** Compare declared mutations to trace records

### 6.2 What Replay Does Not Do

Replay does not re-execute side effects. Re-executing mutations would:

- Duplicate state changes
- Produce different results (world has changed)
- Violate idempotency expectations

Instead, replay validates that:

- Declared mutations match actual mutations
- Ordering constraints were satisfied
- Scope boundaries were respected

### 6.3 Replay Invariants

**I-M10 (Transform Replay)** CT_ operations are replayable. Re-execution with captured inputs produces identical outputs.

$$\text{Replay}(\text{CT}_t, \text{Input}_{\text{captured}}) = \text{Output}_{\text{original}}$$

**I-M11 (Mutation Verifiability)** All state transitions must be derivable from CS_ trace entries. Given a trace, the complete mutation history is reconstructible.

$$\text{Mutations}(\text{Execution}) = \text{Derive}(\text{Trace}(\text{CS\_}))$$

### 6.4 Replay for Conformance Testing

Replay enables conformance testing without re-executing side effects:

1. Capture trace from execution with mocked side effects
2. Replay transforms with same inputs
3. Verify transform outputs match
4. Verify side effect declarations match expected

This testing model isolates behavioral conformance from world state.

---

## 7. Mutation Surface Bounding

The mutation surface is the total set of operations that may change world state.

### 7.1 Definition

$$\text{MutationSurface} = \{s : s \in \text{CS\_}\}$$

The mutation surface is finite and governance-enumerated. Every operation that may change world state is:

1. Declared as a CS_ capability

2. Validated through governance
3. Registered in the capability vocabulary

### 7.2 Bounding Invariants

**I-M12 (Finite Mutation Surface)** The set of mutative primitives is finite and governance-enumerated.

$$|\text{MutationSurface}| < \infty$$

New mutation types require governance ratification. There is no dynamic mutation generation.

**I-M13 (No Implicit State Writes)** All state writes occur through declared CS_ operations. There are no implicit writes, no side-channel mutations, and no hidden state updates.

$$\text{StateWrite}(op) \Rightarrow op \in \text{MutationSurface}$$

### 7.3 Risk Bounding

Finite mutation surface bounds system risk structurally:

| Risk Category | How Bounding Helps |
|---|---|
| **Data corruption** | All writes go through declared operations with validation |
| **Unauthorized access** | Mutation scope is declared and enforced |
| **Audit gaps** | Every mutation is traced; no hidden writes |
| **Cascade failures** | Mutation ordering is explicit and controlled |

The mutation surface is not a list of things that might go wrong. It is a constitutional enumeration of every operation that can change world state.

### 7.4 Surface Expansion Governance

Expanding the mutation surface requires governance ratification:

1. New CS_ capability must be authored
2. Mutation scope must be declared
3. Trace requirements must be specified
4. Governance must validate and ratify

This governance expense is deliberate. Adding mutation capability is a constitutional act, not a development convenience.

---

## 8. Composition Rules

Workflows compose CT_ and CS_ operations in sequence. Composition rules govern how pure computation and governed mutation may interleave.

### 8.1 Permitted Composition Patterns

| Pattern | Description | Example |
|---|---|---|
| CT_ $\rightarrow$ CT_ | Transforms in sequence | Validate, then transform |
| CT_ $\rightarrow$ CS_ | Compute, then mutate | Prepare data, then persist |

| Pattern | Description | Example |
|---|---|---|
| CS_ → CT_ | Mutate, then compute | Store, then derive confirmation |
| CS_ → CS_ | Sequential mutations | Update registry, then write log |

All patterns are permitted. The constraint is explicitness, not prohibition.

### 8.2 Prohibited Composition Patterns

| Pattern | Description | Why Prohibited |
|---|---|---|
| CT_ embeds CS_ | Transform internally performs mutation | Violates I-M1 |
| CS_ invokes CT_ implicitly | Side effect calls undeclared transforms | Violates explicit composition |
| Hidden interleaving | Mutation occurs without workflow declaration | Violates I-M5 |

### 8.3 Composition Invariant

**I-M14 (Explicit Composition)** All transitions between computation and mutation must be declared in workflows.

$$\forall (op_1, op_2) \in \text{Execution} : \text{Declared}(op_1 \to op_2)$$

The workflow declares the composition. The execution engine enforces the declaration. There is no hidden interleaving.

### 8.4 Composition and Payload Flow

Data flows through composed operations via payload:

$$\text{Payload}_0 \xrightarrow{\text{CT}_1} \text{Payload}_1 \xrightarrow{\text{CS}_1} \text{Payload}_2 \xrightarrow{\text{CT}_2} \text{Payload}_3$$

Each operation receives input from payload and writes output to payload. Payload flow is explicit and traceable.

---

## 9. Error Semantics

CT_ and CS_ operations have different error semantics because they have different relationships to world state.

### 9.1 Transform Errors

CT_ failures are deterministic:

| Property | Description |
|---|---|
| **Deterministic** | Same input produces same error every time |
| **Non-mutative** | No world state changes even on error |
| **Retryable** | Safe to retry because no side effects occurred |

| Property | Description |
|---|---|
| **Cacheable** | Error results can be cached like success results |

Transform error handling:

$$\text{Error}(\text{CT}_t, i) \Rightarrow W' = W$$

World state is unchanged on transform error.

### 9.2 Side Effect Errors

CS_ failures require explicit handling:

| Property | Description |
|---|---|
| **Non-deterministic** | Errors may depend on world state |
| **Potentially mutative** | Partial mutation may have occurred |
| **Rollback-dependent** | Recovery depends on mutation type |
| **Trace-required** | Failure must be recorded for audit |

Side effect error handling:

$$\text{Error}(\text{CS}_s) \Rightarrow \text{Traced}(\text{failure}) \wedge \text{RollbackPolicy}(\text{CS}_s)$$

### 9.3 Error Invariant

**I-M15 (Failure Transparency)** Failure behavior must be explicitly declared. Every CS_ operation must declare its failure modes and rollback policy.

$$\forall \text{CS}_s : \text{Declared}(\text{FailureModes}(\text{CS}_s)) \wedge \text{Declared}(\text{RollbackPolicy}(\text{CS}_s))$$

There is no implicit error handling. Failure behavior is part of the capability contract.

### 9.4 Rollback Categories

| Category | Behavior | Example |
|---|---|---|
| **Automatic rollback** | System reverts mutation on failure | Transaction rollback |
| **Manual rollback** | Operator must intervene | External system inconsistency |
| **No rollback** | Mutation is permanent | Append-only log write |
| **Compensating action** | Separate operation to undo | Credit after failed debit |

The rollback category is declared in the capability contract and enforced by the execution engine.

---

## 10. Structural Consequences

The CT_/CS_ separation yields structural properties that enable system guarantees.

### 10.1 Deterministic Replay

Because CT_ operations are pure and CS_ operations are traced, execution is replayable:

- Re-execute transforms to verify computation
- Compare trace records to verify mutation

Replay is not reconstruction from logs. It is re-execution of computation and verification of mutation records.

### 10.2 Clear World Boundary

The world-interaction boundary is explicit:

$$\text{Boundary} = \{op : op \in \text{CS\_}\}$$

Everything inside the boundary may change world state. Everything outside the boundary cannot.

### 10.3 Explicit Mutation Tracking

Every mutation is:

1. Declared in capability contract
2. Authorized through workflow
3. Traced during execution
4. Verifiable through replay

There are no hidden mutations, no untracked state changes, no audit gaps.

### 10.4 Replaceable Computation Layer

CT_ implementations may be replaced without behavioral change:

- Optimized implementations
- Alternative algorithms
- Hardware acceleration
- Cross-platform ports

Purity guarantees equivalence.

### 10.5 Bounded Mutation Risk

The mutation surface is finite and governance-controlled. Risk analysis can enumerate every operation that may change world state.

### 10.6 Trace-Verifiable State Transitions

Every state transition is derivable from CS_ trace entries:

$$\text{StateTransitions} = f(\text{Trace}(\text{CS\_}))$$

Audit does not require state snapshots. Audit reconstructs transitions from trace records.

---

## 11. Relationship to Security (Deferred)

This paper does not analyze security. However, the CT_/CS_ separation provides structural prerequisites for security properties addressed in Paper 8.

### 11.1 Structural Prerequisites

The computation-mutation boundary provides:

| Prerequisite | Security Implication |
|---|---|
| **Bounded mutation surface** | Attack surface is enumerable |
| **No ambient authority** | Capabilities must be explicitly granted |
| **Explicit mutation scope** | Access control has clear targets |
| **Mandatory tracing** | All mutations are auditable |

### 11.2 Inversion of Trust Foundation

Paper 8 will address the "inversion of trust" model where security emerges from mutation discipline rather than defensive layering. This paper provides the foundation:

- Mutation is bounded (finite surface)
- Mutation is declared (explicit scope)
- Mutation is traced (mandatory recording)
- Mutation is governed (ratification required)

Security emerges from mutation discipline, not from perimeter defense.

---

## 12. Relationship to Execution (Clarified)

Paper 6 defines how execution invokes CT_ and CS_ operations [Bachi, 2026f]. This paper defines what those invocations mean semantically.

### 12.1 Execution Mechanics (Paper 6)

Paper 6 specifies:

- DAG compilation from workflow artifacts
- Node routing by concern prefix
- Deterministic execution ordering
- Trace emission mechanics

### 12.2 Semantic Boundaries (This Paper)

This paper specifies:

- Purity invariants for CT_
- Mutation invariants for CS_
- World-interaction boundary
- Replay semantics
- Composition rules

### 12.3 Complementary Relationship

The papers are complementary:

$$\text{Execution (Paper 6)} : \text{How capabilities are invoked}$$

$$\text{Mutation Boundary (Paper 7)} : \text{What invocations mean semantically}$$

Execution is mechanical. The computation-mutation boundary is constitutional.

---

## 13. Limitations

This paper:

- **Does not define runtime binding mechanics.** How CS_ operations are bound to implementations is addressed in Paper 6.
- **Does not analyze threat models.** Security analysis of the computation-mutation boundary is addressed in Paper 8.
- **Does not address economic scaling.** The cost implications of mutation governance are addressed in Paper 9.
- **Does not restate universality.** Computational completeness was established in Paper 2 and is not revisited here.
- **Does not specify implementation.** How CT_ and CS_ operations are implemented is outside the scope of protocol specification.

Its scope is the computation-mutation boundary only: what distinguishes pure computation from governed mutation, and what properties that distinction enables.

---

## 14. Conclusion

Protocol-governed systems require explicit separation between pure computation and governed mutation.

This paper established fifteen invariants governing the computation-mutation boundary:

| Invariant | Statement |
|---|---|
| I-M1 | CT_ artifacts may not perform mutation |
| I-M2 | Identical input produces identical output |
| I-M3 | CT_ may not access mutable external state |
| I-M4 | Re-execution must yield identical trace evidence |
| I-M5 | All mutations must be declared in capability contracts |
| I-M6 | CS_ execution order must match workflow declaration |
| I-M7 | Mutation must emit trace evidence before and after state change |
| I-M8 | CS_ may only mutate declared state domains |
| I-M9 | CT_ operations do not interact with world state |
| I-M10 | CT_ operations are replayable |
| I-M11 | All state transitions must be derivable from CS_ trace entries |
| I-M12 | The set of mutative primitives is finite and governance-enumerated |
| I-M13 | All state writes occur through declared CS_ operations |
| I-M14 | All transitions between computation and mutation must be declared |
| I-M15 | Failure behavior must be explicitly declared |

These invariants are constitutional constraints, not design guidelines. The architecture enforces them through concern prefixes, execution routing, and governance validation.

The relationship between papers establishes the complete semantic model:

> **Protocol (Paper 5) declares behavioral law. Execution (Paper 6) enforces behavioral law. Mutation Boundary (this paper) defines what constitutes computation versus world change. Security (Paper 8) emerges from mutation discipline.**

The separation between CT_ and CS_ is not stylistic—it is structural. A system that cannot distinguish computation from mutation cannot guarantee determinism, cannot provide complete audit, and cannot bound its attack surface.

By enforcing purity invariants on transforms, declaration invariants on side effects, and explicit composition rules on workflows, the architecture establishes a deterministic, replayable, and auditable world-interaction boundary.

---

## License and Use



---

## Author Information

**Bachi (aka Bhash Ganti)** Contact: bachipeachy@gmail.com

**Conflict of Interest:** The author is developing commercial implementations of the described architecture.

---

## References

Bachi aka Bhash Ganti (2026a). Protocol-Governed Systems: An architectural foundation for the AI era. *Zenodo Working Paper. DOI: https://doi.org/10.5281/zenodo.18715516*

Bachi aka Bhash Ganti (2026b). Protocol-Governed Systems: A constitutional realization of Turing-complete systems. *SZenodo Working Paper. DOI: https://doi.org/10.5281/zenodo.18718409*

Bachi aka Bhash Ganti (2026c). The Layer-Concern Constitutional Model: A formal structural taxonomy for protocol-governed systems. *SZenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18719589*

Bachi aka Bhash Ganti (2026d). Governance and Authoring: The legislative process of behavioral law. *SZenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18929868* .

Bachi aka Bhash Ganti (2026e). Protocol as Law: Behavioral specification and versioned authority. *SZenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18930048.*

Bachi aka Bhash Ganti (2026f). Deterministic Enforcement: Runtime binding, execution, and trace conformance. *SZenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18930314.*

Church, A. (1941). The calculi of lambda-conversion. *Annals of Mathematics Studies*, 6.

Dijkstra, E.W. (1976). *A Discipline of Programming.* Prentice Hall.

Hoare, C.A.R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hughes, J. (1989). Why functional programming matters. *The Computer Journal*, 32(2):98–107.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, 93(1):55–92.

Peyton Jones, S. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering Theories of Software Construction*, pages 47–96. IOS Press.

Wadler, P. (1995). Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer.