The Layer-Concern Constitutional Model:

A Formal Structural Taxonomy for Protocol-Governed Systems

Bachi (aka Bhash Ganti) Contact: bachi.bachi@myyahoo.com

Abstract

This paper formalizes the structural core of protocol-governed systems through a constitutional taxonomy based on two orthogonal primitives: **layers** and **concerns**. Prior work established the separation between behavioral specification and execution mechanics [Bachi, 2026a] and demonstrated that constitutional constraint is compatible with universal computation [Bachi, 2026b]. This paper isolates the structural grammar that makes such governance enforceable.

We define eight canonical layers representing lifecycle-bound human responsibilities, and ten canonical concerns representing machine-enforced behavioral types. Each primitive is specified formally, including invariants governing completeness, authority, determinism, prefix uniqueness, and trace attribution. The orthogonality between layers and concerns is defined explicitly to prevent architectural conflation. The resulting taxonomy provides a deterministic method for artifact classification and governance enforcement.

This paper introduces no economic analysis, no computational universality claims, and no AI-era motivation. Its purpose is structural: to define the minimal constitutional grammar required for governable systems. In this paper, "formal" denotes invariant-based structural specification rather than full operational semantics.

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

Keywords: structural taxonomy, constitutional architecture, layers, concerns, governance primitives, artifact classification, software architecture, protocol governance

---

1. Position and Scope

Paper 1 introduced the architectural separation between WHAT (behavior) and HOW (execution) [Bachi, 2026a]. Paper 2 argued that constitutional constraint does not diminish computational universality [Bachi, 2026b].

This paper answers a narrower question:

> *What structural primitives must exist so that governance can be enforced architecturally rather than procedurally?*

The answer is a dual taxonomy composed of:

- **Layers** — organizing artifacts by lifecycle and human authority.
- **Concerns** — organizing artifacts by behavioral type and execution semantics.

These two primitives are independent and must remain so. Their separation is constitutional.

---

2. Structural Premise

In conventional software architecture, "layers" and "concerns" are often conflated [Buschmann et al., 1996]. Modules are placed in directories that simultaneously imply lifecycle responsibility and runtime semantics. This dual use is tolerable in informal systems. It becomes structurally dangerous in governed systems.

Protocol-governed systems require:

1. **Deterministic artifact classification.** Every artifact must belong to exactly one classification cell without ambiguity.
2. **Explicit authority boundaries.** The governing authority for any artifact must be identifiable from its classification.
3. **Machine-verifiable behavioral typing.** Runtime behavior must be inferrable from artifact classification without implementation inspection.
4. **Vocabulary-bounded extensibility.** New artifact types may only be introduced through constitutional amendment, not informal addition.

These requirements cannot be satisfied unless lifecycle structure and runtime behavior are modeled separately.

The Layer-Concern model formalizes this separation.

---

3. Formal Definition of Layer

3.1 Definition

A **Layer** is a structurally bounded partition of a system that organizes artifacts according to design-time responsibility.

Formally, a layer L_i is defined as:

$$L_i = (N_i, R_i, A_i, \mathcal{T}_i)$$

Where:

- N_i is the canonical name of the layer.
- R_i defines the human-facing responsibility of the layer.
- A_i defines the governing authority responsible for artifacts in the layer.
- $\mathcal{T}_i$ defines which artifact types are admissible in that layer.

Layers are lifecycle constructs. They reflect how humans reason about and govern system evolution.

3.2 Layer Invariants

All layers must satisfy four invariants:

I-L1 (Completeness) At any point in time, each artifact belongs to exactly one layer. The layer partition forms a total, disjoint cover over the artifact set. Let $\mathcal{A}$ be the set of all artifacts; then $\{L_1, \ldots, L_8\}$ partitions $\mathcal{A}$ at every instant.

I-L2 (Responsibility Coherence) All artifacts within a layer serve the same lifecycle responsibility. A layer does not mix design-time responsibilities; artifacts with different human-cognitive functions must reside in different layers.

I-L3 (Authority Boundedness) Each layer has a clearly defined governing authority. Modification of artifacts within a layer requires authorization from the layer's governing authority. Authority does not leak across layer boundaries.

I-L4 (Human Addressability) Artifact placement within a layer can be determined without reasoning about runtime semantics. A human stakeholder can identify the correct layer for any design question by considering lifecycle responsibility alone.

These invariants prevent structural ambiguity and eliminate "miscellaneous" containers that accumulate ungoverned artifacts.

---

4. The Eight Canonical Layers

The architecture defines exactly eight layers, ordered by lifecycle position:

#	Layer	Responsibility
1	**Authoring**	Legislative intent creation
2	**Governance**	Validation and ratification
3	**Protocol**	Declared behavioral law
4	**Execution**	Deterministic enforcement engine
5	**Capability Transforms**	Pure computational primitives
6	**Capability Side Effects**	Governed world interaction
7	**Transport**	Ingress and egress mechanics
8	**Structure**	Boot-time structural invariants

Each layer exists for a specific reason. The ordering reflects lifecycle progression from human intent (Layer 1) to structural substrate (Layer 8).

4.1 Authoring

Authoring is where behavioral law originates. It contains drafts, proposals, and human-authored artifacts prior to ratification. Its authority lies with domain stakeholders and institutional governance processes.

Responsibility: The human legislative process—expressing behavioral intent in protocol-declarative form.

Authority: Domain stakeholders, product governance, architectural review.

Artifacts: Template instances, draft protocol artifacts, build registry entries, conformance test specifications.

Authoring defines intent but does not validate it. All system behavior originates in human intent expressed through this layer [Bachi, 2026a].

4.2 Governance

Governance validates authored artifacts against constitutional rules. It enforces schema compliance, vocabulary constraints, and invariant preservation.

Responsibility: Law validation, constitutional enforcement, and compliance verification.

Authority: Constitution validators, schema enforcement engines, compliance officers.

Artifacts: Constitutional schemas, validation rules, conformance test frameworks, vocabulary registries.

Governance ratifies; it does not execute. Once artifacts pass governance, they may enter the Protocol layer. The relationship between Authoring and Governance is legislative: Authoring proposes; Governance ratifies.

Architectural property: Governance is evaluated at authoring time and load time, never at execution time. By the time execution begins, all artifacts are known-valid.

4.3 Protocol

Protocol contains the ratified, version-controlled behavioral artifacts that define what the system is permitted to do. These artifacts are authoritative.

Responsibility: The declaration of lawful behavior.

Authority: Version-controlled protocol repository, institutional change management.

Artifacts: Workflow specifications (WF_), intent declarations (IN_), capability contracts (CC_), runtime binding declarations (RB_), actor registrations (AC_), event definitions (EV_).

Protocol is the specification layer. It stands between governance (which validates) and execution (which enforces). Protocol artifacts are the constitutional law of the system; everything below this layer is enforcement mechanics.

Versioning: Each artifact carries an explicit version identifier. Behavioral change requires creation of new protocol versions; in-place mutation is prohibited. Every artifact version is independently addressable and immutable.

4.4 Execution

Execution enforces protocol law deterministically. It compiles workflows, routes intents, invokes capabilities, and produces traces.

Responsibility: Deterministic enforcement of protocol-declared behavior.

Authority: Execution engine implementation, DAG compiler, node router.

Artifacts: DAG construction logic, workflow executor, execution context, node router, trace sink.

The execution engine is semantically blind: it interprets structure, not domain meaning [Bachi, 2026a]. Given identical artifacts and inputs, execution produces identical observable results.

Key properties: - **Semantic blindness:** No knowledge of business domain meaning. - **Determinism:** Identical inputs produce identical outputs. - **Replaceability:** Multiple execution engines may exist; conformance is verified through trace comparison [Bachi, 2026b].

4.5 Capability Transforms

This layer contains pure computational units. Transforms must be deterministic, side-effect free, and replayable.

Responsibility: Pure, deterministic computation.

Authority: Capability contract specifications, transform atom registry.

Artifacts: Transform executor, transform atoms (individual pure-computation implementations).

Given identical inputs, transforms produce identical outputs without mutation.

Purity guarantees: - Identical inputs always produce identical outputs - No side effects (memory, disk, network, time) - No dependency on external state - Deterministic execution

Compositional property: Pure transformations compose freely. Complex behaviors emerge from combining simple, testable primitives.

4.6 Capability Side Effects

This layer contains governed mutation operations: storage writes, external interactions, registry updates.

Responsibility: Governed interaction with the external world.

Authority: Capability contract specifications, side effect governance declarations.

Artifacts: Persistent storage implementations (CS_MUTABLE_JSON_V0, CS_APPENDONLY_JSONL_V0, CS_REGISTRY_V0).

All mutations must be explicitly declared and traced.

Side effect types: - **Mutable storage:** JSON document updates, database writes - **Append-only storage:** Immutable event logs, audit trails - **Registry operations:** Actor registration, configuration updates - **External interactions:** API calls, message publication

Separation rationale: Capability transforms are separated from capability side effects to enforce a constitutional distinction between computation (pure, replayable, safe) and mutation (world-affecting, governed, irreversible). This separation is not an organizational convenience; it is a governance invariant.

4.7 Transport

Transport handles ingress and egress. It converts external representations into protocol-recognized intents and formats outputs for external consumption.

Responsibility: Ingress and egress mechanics.

Authority: Interface specifications, API contract definitions.

Artifacts: CLI runner, HTTP/REST server, ingress handlers, egress formatters.

Transport carries no behavioral authority. It determines how external actors access what the system does, not what the system does. Behavioral authority resides in the Protocol layer, not in Transport.

Architectural constraint: Transport adapters contain no business logic. They perform only protocol invocation and response formatting.

4.8 Structure

Structure defines boot-time invariants required for protocol resolution and execution determinism. It includes path resolution, artifact discovery rules, and environmental invariants.

Responsibility: Structural invariants required for protocol discovery, resolution, and deterministic execution, independent of domain or implementation.

Authority: System architecture, constitutional path governance.

Artifacts: Path registry (canonical filesystem resolution), environment facts (invariant configuration), protocol loading (lawful protocol discovery and materialization).

Structure is loaded before protocol artifacts and remains immutable at runtime. It is the axiomatic substrate of the system—loaded first, modified never at runtime, and required by all other layers.

Properties: - **Declarative:** Describes invariants, not behavior. - **Authority-bearing:** Path registry is the single source of truth for all artifact resolution. - **Pre-protocol:** Loaded before any protocol artifact can be discovered. - **Domain-independent:** Does not change with business domain or deployment context. - **Non-replaceable:** Unlike implementations, structural invariants cannot be swapped per deployment.

Naming rationale: This layer was previously designated "Common"—a name that resists nothing. Any artifact can plausibly claim to be "common." The name provides no exclusion criteria, no semantic boundary, and no governance leverage. Names in a constitutional architecture carry normative force: they signal what belongs and, equally important, what does not [Bachi, 2026a]. "Structure" actively resists misuse: a utility function is not structural; a helper class is not structural; a shared type is not structural. The name itself enforces governance by making inappropriate additions semantically incoherent.

---

5. Formal Definition of Concern

5.1 Definition

A **Concern** is a behaviorally typed category of artifact with defined runtime semantics.

Formally:

$$C_j = (P_j, N_j, S_j, \Lambda_j)$$

Where:

- P_j is a unique prefix (two-letter identifier).
- N_j is the concern name.
- S_j defines execution semantics.
- Λ_j defines lifecycle handling rules.

Concerns are execution constructs. They reflect how the machine interprets artifacts.

5.2 Concern Invariants

All concerns satisfy four invariants:

I-C1 (Prefix Uniqueness) Each concern has a globally unique identifier prefix. No two concerns share a prefix. Prefix assignment is constitutional and immutable once ratified.

I-C2 (Semantic Determinism) Concern execution semantics are deterministic. Given an artifact of concern C_j with identical inputs and protocol context, the execution engine produces identical behavior.

I-C3 (Trace Attribution) Every execution trace event is attributable to exactly one concern. The concern prefix appears in the trace event, enabling concern-level auditing and compliance verification.

I-C4 (Vocabulary Boundedness) The set of concerns is finite and constitutionally enumerated. Introduction of a new concern requires constitutional amendment—it cannot be introduced by implementation or configuration.

No concern may be introduced informally.

---

6. The Ten Canonical Concerns

The architecture defines ten concerns grouped by behavioral role.

Concern Group I: Authority and Intent

Prefix	Name	Description
AC_	Actors	Authority-bearing principals
IN_	Intents	Requested state transitions

AC_ Actors: Define who may initiate, authorize, or participate in protocol execution. They carry identity, role, and permission attributes that are validated before execution proceeds.

IN_ Intents: Declare what an actor wishes to accomplish without specifying how. Intents are the protocol's unit of behavioral request—they are routed to workflows for fulfillment.

Concern Group II: Orchestration and Binding

Prefix	Name	Description
WF_	Workflows	Lawful orchestration
CC_	Capability Contracts	World-facing permissions
RB_	Runtime Bindings	Late implementation resolution

WF_ Workflows: Define the DAG of steps that fulfill an intent, specifying ordering, branching, and composition. Workflows are the protocol's unit of behavioral orchestration.

CC_ Capability Contracts: Declare what a computation or side effect is permitted to do, what inputs it accepts, what outputs it produces, and what governance constraints apply. Contracts are the governed interface between intent and implementation.

RB_ Runtime Bindings: Connect capability contracts to concrete implementations at deployment time, enabling environment-specific resolution without altering protocol semantics.

Concern Group III: Execution Mechanics

Prefix	Name	Description
CT_	Capability Transforms	Pure computation
CS_	Capability Side Effects	Controlled mutation

CT_ Capability Transforms: Execute deterministic, side-effect-free computations: hashing, formatting, validation, derivation. They are replayable, cacheable, and safe to retry.

CS_ Capability Side Effects: Perform governed world interaction: storage writes, external API calls, message publication. Each side effect is traced, bounded by contract, and irreversible.

Concern Group IV: Observation and Delivery

Prefix	Name	Description
EV_	Events	Emitted facts (trace)
TI_	Transport Ingress	External entry points
TE_	Transport Egress	External exit points

EV_ Events: The trace material produced by execution—structured records of what occurred, in what order, under what authority, and with what outcome. Events are not commands; they are evidence. This distinction matters for governance: events cannot be governed as actions (they cannot fail, be retried, or be authorized), but they must be governed as evidence (they must be complete, ordered, and tamper-evident).

TI_ Transport Ingress: Govern how external actors submit requests to the system—protocol translation, boundary validation, and request normalization.

TE_ Transport Egress: Govern how the system delivers results to external actors—response formatting, protocol translation, and delivery confirmation.

Each concern carries distinct execution semantics independent of layer placement.

---

7. Orthogonality

Layers and concerns are independent axes.

A layer answers: > *Where in the lifecycle does this artifact belong?*

A concern answers: > *What behavioral semantics does this artifact carry?*

An artifact may change layers during its lifecycle but never changes concern. This orthogonality prevents structural conflation.

Formal statement: For any artifact a in the system, a has coordinates (L_i, C_j) where L_i is the layer and C_j is the concern. For all artifacts a, Concern(a) is invariant under lifecycle transition; Layer(a, t) may vary over time t. The concern coordinate is assigned at artifact creation and never changes; the layer coordinate reflects current lifecycle position.

8. The Layer-Concern Matrix

Every artifact in the system has two coordinates: (Layer, Concern).

Examples:

Artifact State	Layer	Concern	Coordinates
A workflow draft	Authoring	WF_	(Authoring, WF_)
A ratified workflow	Protocol	WF_	(Protocol, WF_)
A compiled workflow	Execution	WF_	(Execution, WF_)
A transform atom	Capability Transforms	CT_	(Capability Transforms, CT_)
A storage handler	Capability Side Effects	CS_	(Capability Side Effects, CS_)

Concern remains constant; layer changes with lifecycle stage.

8.1 The Orthogonality Matrix

The following matrix illustrates which concerns manifest in which layers. An "x" indicates that the concern manifests within the layer.

	AC_	IN_	WF_	CC_	RB_	CT_	CS_	EV_	TI_	TE_
Authoring	x	x	x	x	x			x		
Governance	x	x	x	x	x	x	x	x	x	x
Protocol	x	x	x	x	x			x		
Execution		x	x	x	x	x	x	x		
Capability Transforms						x				
Capability Side Effects							x			
Transport									x	x
Structure										

Key observations:

1. **Governance spans all concerns.** Governance validates every type of artifact regardless of concern—this is the constitutional function of the Governance layer applied uniformly. Governance does not introduce new execution semantics; it validates artifacts independent of concern-specific behavior. Its spanning nature is lifecycle-based, not behavioral, distinguishing it from cross-cutting concerns in aspect-oriented models [Kiczales et al., 1997].
2. **Structure spans no concerns.** Structure is pre-concern: it provides the invariants that make concern-based execution possible but does not itself participate in any concern's execution semantics.
3. **Most layers span multiple concerns.** This confirms that layers and concerns are genuinely orthogonal; layers cannot be reduced to concerns or vice versa.
4. **Capability Transforms and Capability Side Effects are concern-aligned.** These two layers each house exactly one concern, reflecting the architectural decision to give world-interaction mechanics their own structural boundaries for governance purposes.

This two-dimensional classification eliminates ambiguity in artifact placement and governance.

9. Constitutional Extension

The taxonomy is extensible but only constitutionally.

9.1 Adding a Layer

Adding a layer requires:

1. **Lifecycle justification:** The new layer must represent a distinct lifecycle phase not covered by existing layers.
2. **Invariant preservation:** The new layer must satisfy all four layer invariants (I-L1 through I-L4).
3. **Position justification:** The layer's position in the lifecycle ordering must be explicitly justified.
4. **Governance review:** All existing governance rules must be reviewed for impact.

9.2 Adding a Concern

Adding a concern requires:

1. **Unique prefix:** Assignment of a globally unique two-letter prefix.
2. **Defined execution semantics:** Explicit specification of how the execution engine handles artifacts of this concern.
3. **Trace attribution rules:** Specification of how trace events from this concern are identified and attributed.
4. **Execution engine extension:** Modification of the execution engine to handle the new concern's lifecycle semantics.
5. **Vocabulary extension:** Constitutional amendment to the concern vocabulary.

Extension is a governance act, not an implementation decision. The constitutional expense of extension is deliberate: it prevents vocabulary inflation and maintains the bounded behavioral surface that enables governance [Bachi, 2026a].

---

10. Structural Consequences

The Layer-Concern model ensures:

1. **Deterministic artifact classification.** Every artifact has exactly one (Layer, Concern) coordinate pair, determined by structural rules rather than implementation judgment.
2. **Vocabulary-bounded behavioral surface.** The ten concerns define the complete space of behavioral types. New behavioral types require constitutional amendment.
3. **Machine-enforceable governance boundaries.** Layer and concern boundaries can be enforced automatically through prefix validation, path resolution, and schema enforcement.
4. **Separation of design reasoning from runtime semantics.** Humans reason about layers (lifecycle); machines enforce concerns (behavior). Neither axis contaminates the other.
5. **Clear extension pathways.** When the taxonomy must grow, the extension protocol provides explicit criteria and procedures.

The Layer-Concern model constitutes a constrained Cartesian product over lifecycle partitions and behavioral types, subject to invariant enforcement on both axes. This two-dimensional classification with dual-axis constraints is the central formal contribution.

It provides architectural grammar without prescribing economic or computational claims.

---

11. Limitations

This paper:

- Does not formalize full operational semantics. The execution semantics of each concern are characterized but not given complete formal treatment.
- Does not prove computational universality. That claim is established in prior work [Bachi, 2026b] and is not restated here.
- Does not analyze lifecycle economics. Cost-benefit analysis of governance overhead is outside scope.
- Does not restate AI-era motivations. The motivational context is established in prior work [Bachi, 2026a] and is not repeated here.

Its contribution is structural taxonomy—the minimal grammar required for architectural governance.

---

12. Conclusion

The Layer-Concern Constitutional Model defines the minimal structural grammar necessary for protocol-governed systems.

By separating lifecycle partitions (layers) from behavioral types (concerns) and enforcing invariants on both axes, the architecture achieves:

- **Determinism:** Artifact classification is rule-based, not judgment-based.
- **Bounded vocabulary:** The behavioral surface is finite and enumerated.
- **Enforceable governance:** Boundaries are machine-verifiable.
- **Orthogonal evolution:** Layers and concerns evolve independently.

This structural taxonomy underpins the arguments presented in earlier and subsequent papers but stands independently as a formal classification system for governable software architecture.

The eight layers organize human reasoning about system evolution. The ten concerns organize machine enforcement of behavioral law. Their orthogonality is not a design choice but a constitutional requirement: conflating them produces systems that are neither clearly designed nor reliably governed [Bachi, 2026a].

Names, prefixes, invariants, and extension protocols function as structural enforcement mechanisms rather than documentation artifacts. They constitute the grammar that makes governance architectural rather than procedural—the foundation upon which protocol-governed systems rest.

License and Use

Author Information

Bachi (aka Bhash Ganti) Contact: bachi.bachi@myyahoo.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*

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.

Dijkstra, E.W. (1982). On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer.

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Object Management Group (2011). Business process model and notation (BPMN) version 2.0. OMG Standard.

Object Management Group (2014). Model driven architecture (MDA) guide revision 2.0. OMG Document ormsc/14-06-01.

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Spivey, J.M. (1989). *The Z Notation: A Reference Manual*. Prentice Hall.

van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., and Barros, A.P. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51.