Protocol-Governed Systems: An Architectural Foundation for the AI Era

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

Abstract

Modern software systems conflate business behavior with implementation code, rendering system intent implicit, difficult to audit, and vulnerable to semantic drift. To answer the question "What does this system do?", practitioners must inspect implementation logic rather than consult authoritative behavioral specifications. This architectural pattern fails increasingly as AI-generated code replaces human-authored implementations, accelerating change while eroding comprehension.

This paper introduces a protocol-governed software architecture in which system behavior is expressed as declarative, version-controlled artifacts that are independent of execution engines. The architecture enforces a constitutional separation between protocol (what the system does) and execution (how it does it), with formal governance ensuring deterministic and behaviorally equivalent outcomes across implementation changes.

We formalize two constitutional primitives that organize the architecture along orthogonal axes. **Layers** are the eight structural partitions through which humans design, author, govern, and evolve the system—from legislative intent to structural invariant. **Concerns** are the ten behavioral primitives through which machines load, validate, execute, and trace system behavior—from actor initiation to evidence emission. Layers organize human cognition about design; concerns organize machine enforcement of behavioral law. The distinction is not stylistic but architectural: conflating the two produces systems that are neither clearly designed nor reliably governed.

Determinism is enforced through trace-based conformance verification, enabling behavioral equivalence to be proven without implementation inspection. A prototype implementation validates feasibility for high-assurance domains, including hierarchical cryptographic workflows and immutable append-only systems. Results demonstrate that system behavior can remain explicit, auditable, and governable across execution engine evolution, optimization strategies, and AI-assisted code generation.

This work establishes a foundation for systems whose behavior remains stable and inspectable despite rapid implementation change—addressing a central governance challenge in contemporary software engineering.

Categories: Software Engineering (cs.SE), Programming Languages (cs.PL), Software Architecture
Keywords: software architecture, declarative systems, deterministic execution, protocol governance, AI-generated code, auditability, separation of concerns, constitutional primitives, architectural layers, behavioral concerns

1 Motivation: Why Protocol-Governed Architecture is Inevitable

1.1 The Imperative Architecture Paradox in the AI Era

The velocity of AI-assisted software development is accelerating beyond human capacity to audit, understand, or govern. Large language models now generate production-quality code at speeds

that dwarf human output [Chen et al., 2021]. Industry analysts project that AI-generated code will constitute the majority of production software within this decade [GitHub, 2023].

This presents a paradox. AI excels at generating and refactoring implementation code—the mechanical translation of intent into executable instructions. Yet what AI cannot reliably preserve is system meaning: the business rules, constraints, invariants, and behavioral contracts that define what a system must do rather than how it does it.

In traditional imperative architectures, business logic is intertwined with execution mechanics. Business rules are scattered across conditional statements, embedded in service boundaries, and encoded implicitly in data flows. There exists no authoritative artifact that declares system behavior independently of implementation.

This coupling creates a fundamental problem: you cannot dispose of code without losing business logic.

Organizations have invested decades in systems whose business processes are encoded implicitly within millions of lines of imperative code [Erlikh, 2000]. No one possesses authoritative documentation of what these systems actually do—the code is the specification. Redesigning from scratch is infeasible because the business knowledge exists only in working (if poorly understood) implementations.

The result is that AI-assisted development on legacy systems becomes, as practitioners observe, "putting lipstick on a pig." AI can optimize, refactor, and extend code, but it cannot extract or preserve implicit business meaning. Each AI-generated change risks subtle semantic drift—altered validation rules, shifted edge-case handling, or reordered operations—that tests may not catch [Pearce et al., 2022].

This requires a new paradigm.

1.2 Separating What Is Dispensable from What Must Endure

The core insight of protocol-governed architecture is that software is dispensable; business logic is not.

Implementation code—the Python, Java, Rust, or JavaScript that executes business rules—is a means to an end. It can be rewritten, optimized, migrated to new platforms, or regenerated by AI. What cannot be casually discarded is the authoritative specification of what the system must do: the business processes, compliance requirements, data contracts, and behavioral invariants that define organizational value.

Protocol-governed architecture enforces a strict separation:

- **Protocol artifacts** declare system behavior in explicit, declarative, version-controlled form. These are the authoritative source of truth—independent of any execution engine.
- **Execution engines** interpret protocol artifacts deterministically. They are interchangeable components that can be replaced, optimized, or AI-generated without altering system meaning.

This separation aligns naturally with AI-driven software development. AI can freely generate, optimize, and evolve execution engines provided they conform to protocol specifications. Behavioral equivalence is verified mechanically through trace-based conformance testing—not through human code review.

The result is reduced risk when switching implementations. Organizations can adopt new languages, platforms, or AI-generated code with confidence that behavioral contracts remain preserved. The protocol, not the code, defines what must endure.

1.3 Proven Patterns: This Is Not New to Technology

The separation of behavioral specification from execution mechanics is novel to mainstream software engineering, but it is a proven pattern in other technology domains. Two analogies illustrate the approach:

1.3.1 Industrial Control Systems and Programmable Logic Controllers

Industrial automation has operated under protocol-governed principles for over fifty years. Programmable Logic Controllers (PLCs) execute ladder logic—declarative specifications of industrial processes—without embedding process knowledge in the controller hardware [Bolton, 2015].

When a manufacturing process changes, engineers modify the ladder logic program. They do not design a new PLC. The controller is a generic execution engine; the specification declares behavior.

This architecture achieves complete separation of concerns. Process engineers define what happens; control engineers ensure the execution platform performs reliably. Neither domain contaminates the other.

Why does software engineering not follow this pattern? Historically, the answer was expressive complexity—business software seemed too varied for declarative specification. Protocol-governed architecture demonstrates that this assumption is incorrect. With appropriate vocabulary design and compositional primitives, complex business behavior can be declared rather than coded.

1.3.2 Operating Systems and Applications

The relationship between operating systems and applications provides another familiar analogy. No one designs a new operating system for each application. Applications declare their requirements; the operating system provides execution services.

Unix, VMS, MVS, DOS, and their descendants established this pattern fifty years ago. Applications are portable across operating system implementations (within compatibility bounds). The operating system—the execution engine—can evolve, optimize, and be replaced without invalidating applications.

Yet mainstream software engineering rebuilds the "operating system" for every business domain. Each enterprise application contains custom runtime infrastructure: database access patterns, transaction management, error handling, observability—all implemented from scratch and intertwined with business logic.

Protocol-governed architecture applies the OS/application separation to business software. The execution engine provides generic interpretation services; protocol artifacts declare domain-specific behavior. Like applications on an OS, protocol artifacts are portable across conformant execution engines.

2 Architectural Foundations

2.1 The WHAT vs. HOW Separation

Software systems conflate two distinct concerns:

- **WHAT:** system behavior, rules, constraints, and state transitions
- **HOW:** execution strategy, performance optimization, language, and deployment

Protocol-governed architecture enforces strict separation. Behavior is expressed declaratively; execution engines interpret that behavior using any compliant strategy.

Auditing behavior therefore requires inspecting protocol artifacts—not execution code.

2.2 Protocol Artifacts as Constitutional Law

Protocol artifacts are not documentation; they are binding constraints. They declare:

- units of work with explicit inputs and outputs,
- allowed data flows,
- declared side effects,
- failure semantics and determinism requirements.

Execution engines have no discretion to reinterpret protocol meaning. Conformance is defined by observable equivalence, not implementation similarity.

2.3 Trace-Based Conformance Verification

Determinism is enforced through trace-based verification:

- **Trace Schema** defines observable execution events.
- **Trace Capture** records execution outcomes.
- **Replay Verification** ensures identical outcomes under replay.
- **Conformance Testing** validates behavioral equivalence across engines.

This enables verification of AI-generated or optimized implementations without code inspection.

2.4 Governance Precedence

Governance defines non-negotiable constraints:

- vocabulary and schema rules,
- structural correctness,
- determinism requirements.

Execution engines that violate governance are non-conformant regardless of performance.

3 Layers and Concerns: Constitutional Primitives

Protocol-governed systems are organized along two orthogonal axes. The first axis—*layers*—partitions the system as humans encounter it: as a design artifact to be authored, reviewed, governed, and evolved. The second axis—*concerns*—partitions the system as machines encounter it: as a set of behavioral primitives to be loaded, validated, executed, and traced. Confusing the two is a categorical error that undermines both design clarity and runtime governance.

3.1 Why Two Taxonomies, Not One

In conventional software engineering, the terms "layer" and "concern" are used interchangeably or defined in terms of each other [Buschmann et al., 1996]. This conflation is tolerable in systems where human developers perform both design and execution reasoning simultaneously. It becomes intolerable in protocol-governed systems, where design authority (human) and execution enforcement (machine) are constitutionally separated.

A layer may contain artifacts belonging to multiple concerns. A concern may manifest across multiple layers. The relationship is not hierarchical but dimensional. Layers are the rows of the architectural matrix; concerns are the columns.

Conflating these axes produces two characteristic failure modes:

1. **Layers treated as concerns:** Developers assume that a module's filesystem location (layer) determines its runtime behavior (concern). This assumption breaks when a single layer houses artifacts with different execution semantics—e.g., the Protocol layer contains both workflow specifications (`WF_`) and capability contracts (`CC_`), which carry entirely different execution obligations.
2. **Concerns treated as layers:** Developers assume that artifacts with the same behavioral prefix belong in the same filesystem location. This assumption breaks when a concern's artifacts participate in different phases of the system lifecycle—e.g., a workflow (`WF_`) is authored in the Authoring layer, declared in the Protocol layer, validated in the Governance layer, and executed in the Execution layer.

Formal definitions prevent both failure modes by making the orthogonality explicit and constitutional.

3.2 Formal Definition: Layer

Layer (noun): A structurally bounded partition of a protocol-governed system that organizes design-time responsibilities from the human perspective. Each layer encapsulates a coherent set of activities, artifacts, and authorities that humans must reason about when designing, authoring, governing, or evolving the system.

Formally, a layer L_i is a tuple $(N_i, R_i, A_i, \mathcal{A}_i)$ where:

- N_i is the constitutional identifier of the layer.
- R_i is the human-cognitive function the layer serves.
- A_i specifies who (human roles, institutional processes) governs the layer's content.
- $\mathcal{A}_i$ is the set of artifact types that may reside within the layer.

Layers satisfy the following invariants:

I-L1 (Completeness): Every artifact in the system belongs to exactly one layer. There are no artifacts that exist outside the layer structure.

I-L2 (Responsibility Coherence): All artifacts within a layer serve the same human-cognitive function. A layer does not mix design-time responsibilities.

I-L3 (Authority Boundedness): Each layer has a well-defined authority structure. Modification of artifacts within a layer requires authorization from the layer's governing authority.

I-L4 (Human Addressability): Layers are named and organized such that a human stakeholder can identify the correct layer for any design question without consulting execution semantics.

3.3 The Eight Canonical Layers

The protocol-governed architecture comprises exactly eight layers, ordered by their position in the design lifecycle—from legislative intent to structural invariant. Figure 1 illustrates the layer structure with velocity characteristics.

Layers 1–3 operate at human speed, requiring institutional deliberation. Layers 4–7 operate at machine speed, performing deterministic enforcement. Layer 8 operates at boot time, establishing structural invariants before protocol discovery. The authority gradient flows from maximal human authority at Layer 1 to fully encoded machine enforcement at Layer 7, with Layer 8 providing the axiomatic substrate.

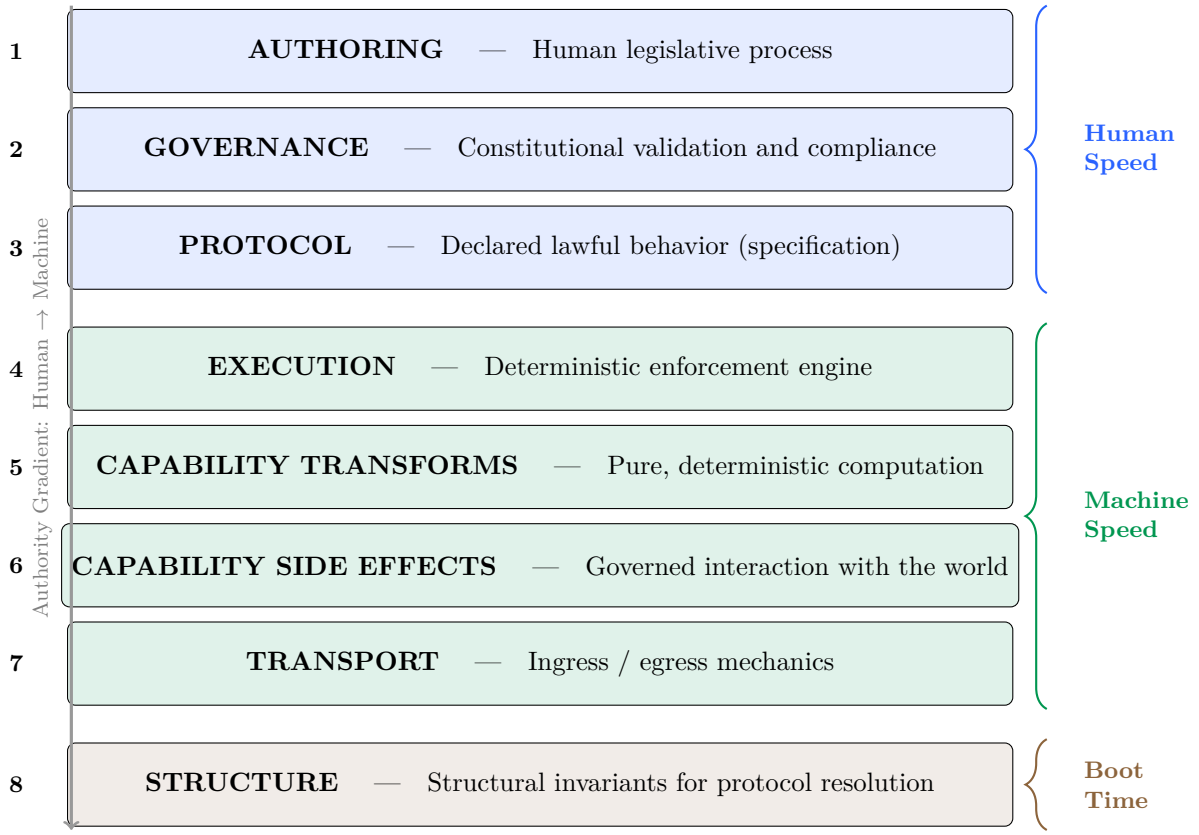

Figure 1: The Eight Canonical Layers of protocol-governed architecture.

3.3.1 Layer 1: Authoring

Responsibility: The human legislative process. Authoring is where human stakeholders—domain experts, product owners, architects—express behavioral intent in protocol-declarative form. Authoring produces the raw material of governance: intent declarations, workflow specifications, capability definitions, and actor registrations.

Authority: Domain stakeholders, product governance, architectural review.

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

Authoring is deliberately positioned as the first layer because all system behavior originates in human intent. No artifact exists in the system that was not authored by a human stakeholder. AI may assist in generating implementations, but the authoring of behavioral law is a human institutional act.

3.3.2 Layer 2: Governance

Responsibility: Law validation, constitutional enforcement, and compliance verification. Governance validates that authored artifacts conform to constitutional schemas, vocabulary constraints, and institutional policies. Governance does not create behavioral law; it validates that authored law is well-formed and admissible.

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

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

Governance follows Authoring because authored artifacts must be validated before they become protocol law. The relationship 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.

3.3.3 Layer 3: Protocol

Responsibility: The declaration of lawful behavior. The Protocol layer houses the ratified, version-controlled artifacts that constitute the system's behavioral law. These artifacts specify—in declarative, machine-readable form—what the system is permitted to do, what actors may participate, what workflows are sanctioned, and what capabilities are available.

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. There is no "version 3.1.4" semantic drift—every artifact version is independently addressable and immutable. Behavioral change requires new versions, not in-place modification.

3.3.4 Layer 4: Execution

Responsibility: Deterministic enforcement of protocol-declared behavior. The Execution layer contains the runtime engine that compiles protocol artifacts into executable DAGs, routes execution through declared intents and capability contracts, and produces structural traces that prove compliance.

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

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

Execution is the enforcement boundary. Above this layer, humans author and declare. Below this layer, the system interacts with the world. Execution ensures that world-interaction occurs only as protocol permits.

Key properties:

- **Semantic blindness:** The execution engine has no knowledge of business domain meaning. It interprets protocol structure, not business semantics.
- **Determinism:** Given identical artifacts and inputs, execution produces identical observable results.
- **Replaceability:** Multiple execution engines may exist. Conformance is verified through trace comparison.

3.3.5 Layer 5: Capability Transforms

Responsibility: Pure, deterministic computation. Capability transforms perform data transformations—hashing, formatting, validation, calculation—without producing side effects. They are the system's pure functions: given the same inputs, they produce the same outputs, unconditionally.

Authority: Capability contract specifications, transform atom registry.

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

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.

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.

3.3.6 Layer 6: Capability Side Effects

Responsibility: Governed interaction with the external world. Capability side effects perform mutations—database writes, file operations, API calls, message publication—under the governance constraints declared in capability contracts. Every side effect is traced, bounded, and attributable.

Authority: Capability contract specifications, side effect governance declarations.

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

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

Ordering guarantees: Side effects execute in protocol-declared order. Non-determinism (e.g., network latency) is bounded and observable.

Auditability: All side effects are traced. The complete effect history is reconstructible from execution traces. A system's risk profile is proportional to its side-effect surface, and that surface is constitutionally bounded by this layer's contents.

3.3.7 Layer 7: Transport

Responsibility: Ingress and egress mechanics. The Transport layer manages how external actors and systems communicate with the protocol-governed system. It handles protocol translation (HTTP to internal representation), input validation at the system boundary, and output formatting. Transport is delivery, not behavior.

Authority: Interface specifications, API contract definitions.

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

Transport is positioned as a late layer because it is mechanically necessary but behaviorally inert. Transport does not determine what the system does; it determines how external actors access 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.

3.3.8 Layer 8: Structure

Responsibility: Structural invariants required for protocol discovery, resolution, and deterministic execution, independent of domain or implementation. The Structure layer houses the system's foundational infrastructure: canonical path resolution, environment invariants, and protocol loading mechanics. These are not business-specific artifacts; they are the axiomatic substrate upon which all other layers depend.

Authority: System architecture, constitutional path governance.

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

Formal definition:

> *Structure contains the structural invariants required for protocol discovery, resolution, and deterministic execution, independent of domain or implementation. It is the axiomatic substrate of the system—loaded first, modified never at runtime, and required by all other layers.*

The contents of this layer are:

- **Declarative:** They describe invariants, not behavior.
- **Authority-bearing:** Path registry is the single source of truth for all artifact resolution.
- **Pre-protocol:** They are loaded before any protocol artifact can be discovered.
- **Domain-independent:** They do 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. "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. Alternative names considered and rejected: *common* (implies grab bag), *utils* (trivializes authority), *core* (overloaded), *foundation* (vague), *kernel* (implies OS-level execution), *constitution* (would blur design vs. law), *axioms* (too abstract for engineering teams).

Separation guarantee: Structure tools are resolved at boot time. They are never modified during execution and carry no domain-specific semantics.

3.4 Layer Ordering and Lifecycle Position

The eight layers follow a deliberate ordering that mirrors the system lifecycle, as summarized in Table 1.

Table 1: Layer lifecycle ordering with velocity characteristics.

#	Layer	Lifecycle Phase	Speed
1	Authoring	Legislative intent	Human
2	Governance	Constitutional validation	Human
3	Protocol	Behavioral specification	Human
4	Execution	Deterministic enforcement	Machine
5	Capability Transforms	Pure computation	Machine
6	Capability Side Effects	Governed world interaction	Machine
7	Transport	Delivery mechanics	Machine
8	Structure	Structural invariants	Boot-time

Layers 1–3 operate at human speed: they require institutional deliberation and human authority. Layers 4–7 operate at machine speed: they are deterministic, automated, and enforcement-driven. Layer 8 operates at boot-time: it is resolved once during system initialization and remains invariant throughout execution.

This ordering produces a natural authority gradient: human authority is maximal at Layer 1 (where intent originates) and decreases through the layers as behavioral law is progressively

compiled into executable form. By the time execution reaches Layers 5–7, human authority has been fully encoded in protocol artifacts; no further human judgment is required.

3.5 Formal Definition: Concern

Concern (noun): A behaviorally typed category of protocol artifact that carries specific execution semantics within the runtime engine. Each concern defines a class of entities that the machine loads, validates, routes, executes, and traces according to concern-specific rules.

Formally, a concern C_j is a tuple $(P_j, N_j, S_j, \Lambda_j)$ where:

- P_j is the canonical two-letter prefix used in all artifact naming (e.g., `WF_`, `CC_`, `EV_`).
- N_j is the human-readable concern name.
- S_j specifies the execution behavior associated with artifacts of this concern.
- Λ_j describes how the runtime engine discovers, loads, validates, and executes artifacts of this concern.

Concerns satisfy the following invariants:

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

I-C2 (Semantic Determinism): The execution semantics of a concern 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 event in a trace 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, enumerated, and vocabulary-controlled. Introduction of a new concern requires constitutional amendment—it cannot be introduced by implementation or configuration.

3.6 The Ten Canonical Concerns

The protocol-governed architecture recognizes exactly ten concerns, grouped by their behavioral role in the execution lifecycle. The ordering reflects execution dependency—earlier concerns establish context that later concerns consume. Figure 2 illustrates the causal flow.

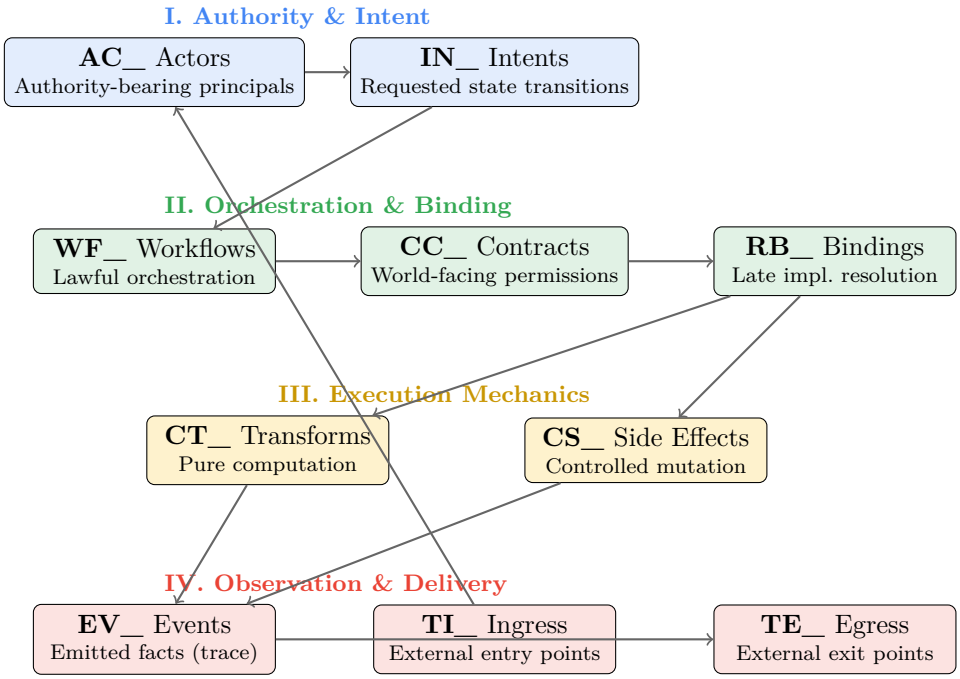

Figure 2: The Ten Canonical Concerns and their causal execution flow.

An Actor (AC_) initiates execution with an Intent (IN_), which is routed to a Workflow (WF_) that invokes Capability Contracts (CC_). Contracts resolve to implementations via Runtime Bindings (RB_), which execute as pure Transforms (CT_) or governed Side Effects (CS_). Execution produces Events (EV_) as trace evidence. External access enters via Transport Ingress (TI_) and exits via Transport Egress (TE_).

3.6.1 Concern Group I: Authority and Intent

These concerns establish *who* acts and *what* they intend.

`AC_` **Actors:** Authority-bearing principals. 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:** Requested state transitions. An intent declares 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.

3.6.2 Concern Group II: Orchestration and Binding

These concerns establish *how* intents are fulfilled and *with what* resources.

`WF_` **Workflows:** Lawful orchestration of intent. 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:** World-facing permissions. A capability contract declares 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:** Late binding to implementations. Runtime bindings connect capability contracts to concrete implementations at deployment time, enabling environment-specific resolution without altering protocol semantics.

3.6.3 Concern Group III: Execution Mechanics

These concerns perform the *work*—computation and world interaction.

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

`CS_` **Capability Side Effects:** Controlled mutation. Side effects perform governed world interaction: storage writes, external API calls, message publication. Each side effect is traced, bounded by contract, and irreversible.

3.6.4 Concern Group IV: Observation and Delivery

These concerns handle *evidence* of execution and *access* to the system.

`EV_` **Events:** Emitted facts. Events are 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:** External entry points. Ingress concerns govern how external actors submit requests to the system—protocol translation, boundary validation, and request normalization.

`TE_` **Transport Egress:** External exit points. Egress concerns govern how the system delivers results to external actors—response formatting, protocol translation, and delivery confirmation.

3.7 Concern Prefix as Vocabulary Control

The concern prefix system (`AC_`, `WF_`, `IN_`, etc.) serves as a vocabulary control mechanism. Every protocol artifact's name begins with its concern prefix, making the artifact's behavioral type lexically visible:

- `WF_CREATE_WALLET_V0` — a workflow (orchestration)
- `CC_HASH_SHA256_V0` — a capability contract (world-facing permission)
- `CS_MUTABLE_JSON_V0` — a capability side effect (controlled mutation)
- `AC_SYSTEM_ADMIN_V0` — an actor (authority-bearing principal)

This naming convention is constitutional. The prefix is a machine-parseable declaration of behavioral type that enables automated routing by the execution engine, concern-scoped governance validation, trace-level behavioral attribution, and vocabulary-bounded security (artifacts with unknown prefixes are rejected).

3.8 The Layer-Concern Orthogonality Matrix

Layers and concerns intersect but do not align. Table 2 illustrates which concerns manifest in which layers.

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.
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.

Table 2: Layer-concern orthogonality matrix. An "×" indicates that the concern manifests within the layer.

	AC/	*IN*/	*WF*/	*CC*/	*RB*/	*CT*/	*CS*/	*EV*/	*TI*/	*TE*/
Authoring	×	×	×	×	×			×		
Governance	×	×	×	×	×	×	×	×	×	×
Protocol	×	×	×	×	×			×		
Execution		×	×	×	×	×	×	×		
Capability Transforms						×				
Capability Side Effects							×			
Transport									×	×
Structure										

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.

The matrix resolves a common architectural question: "Where does this artifact belong?" The answer requires two coordinates: *layer* (where in the design lifecycle does this artifact's responsibility reside?) and *concern* (what behavioral type is this artifact?). A workflow specification (`WF_`) resides in the Authoring layer as a draft, in the Protocol layer as ratified law, in the Governance layer during validation, and in the Execution layer as a compiled DAG. Its concern does not change across layers; its layer changes as it moves through the lifecycle.

3.9 Extension Protocol

The sets of layers and concerns defined here are canonical but not eternal. Institutional evolution may require new layers or concerns. The extension protocol is:

Adding a layer: Requires constitutional amendment. The new layer must satisfy all four layer invariants (I-L1 through I-L4). The layer's position in the lifecycle ordering must be justified. All existing governance rules must be reviewed for impact.

Adding a concern: Requires constitutional amendment and vocabulary extension. The new concern must be assigned a unique two-letter prefix. The concern must satisfy all four concern invariants (I-C1 through I-C4). The execution engine must be extended to handle the new concern's lifecycle semantics.

Both extensions are deliberate, institutional acts—not implementation decisions. Layers and concerns are governance primitives, not engineering conveniences.

4 Architectural Properties

4.1 Complete Separation of Concerns

The orthogonal layer-concern model achieves complete separation. Each layer addresses exactly one design-time responsibility; each concern carries exactly one set of execution semantics. No layer bleeds into another, and each can be modified, replaced, or AI-generated independently.

The separation is enforced at two levels:

- **Design-time** (layers): Artifact placement is governed by layer invariants. A transport adapter cannot reside in the Protocol layer; a governance schema cannot reside in the Capability Transforms layer.
- **Runtime** (concerns): Artifact routing is governed by concern prefixes. The execution engine routes `WF_` artifacts to the workflow executor, `CT_` artifacts to the transform pipeline,

and `CS_` artifacts to the side-effect handler. Routing is determined by prefix, not by filesystem location.

4.2 Vocabulary-Bounded Attack Surface

Security emerges from architectural constraint rather than defensive programming. The protocol vocabulary defines the complete space of possible behaviors. Behaviors not expressible in the vocabulary cannot occur.

The concern prefix system reinforces this property: only artifacts with recognized prefixes (`AC_`, `IN_`, `WF_`, `CC_`, `RB_`, `CT_`, `CS_`, `EV_`, `TI_`, `TE_`) are admissible. Artifacts with unknown prefixes are rejected at load time. The behavioral space is bounded by the ten canonical concerns, and expansion of that space requires constitutional amendment.

Implications:

- No undeclared side effects
- No ambient authority
- No implicit control flow
- Attack surface remains bounded as system grows

Vocabulary expansion requires explicit governance approval, providing structural security review.

4.3 Granular Version Control

Traditional software versioning operates at coarse granularity: application version 3.1.4, library version 2.0.1. Semantic drift accumulates invisibly within version boundaries.

Protocol-governed systems version at artifact granularity:

- Each workflow has an independent version
- Each capability contract has an independent version
- Each intent definition has an independent version

Benefits:

- Behavioral change is always explicit
- Compatibility is mechanically verifiable
- Rollback operates at behavioral unit granularity
- No "what changed in this release?" ambiguity

4.4 Extreme Scalability Through Composition

Protocol-governed systems scale through composition rather than code duplication. Complex behaviors assemble from stable, tested primitives.

This compositional approach exhibits sub-linear complexity growth: new behaviors compose from existing vocabulary rather than introducing novel interactions.

5 Case Studies

5.1 Cryptographic Workflow System

A prototype system implements hierarchical cryptographic workflows entirely through protocol artifacts. The system manages cryptographic key derivation, wallet creation, and actor verification.

Validation: Identical protocol artifacts produce identical key derivations across execution engine variants. Behavioral equivalence is proven through trace comparison without cryptographic code inspection.

5.2 Immutable Append-Only System

An append-only ledger system demonstrates protocol-governed state evolution. The system maintains immutable event logs with cryptographic integrity verification.

Property demonstrated: State evolution is auditable through protocol artifact inspection. No implementation knowledge is required to verify ledger integrity.

5.3 Protocol Evolution Without Implementation Change

The prototype demonstrates behavioral evolution through protocol versioning. New workflow versions introduce modified business rules while execution engines remain unchanged.

Legacy and new protocol versions coexist. Clients migrate at their own pace.

6 Evaluation

6.1 Determinism Verification

Repeated executions with identical inputs produce equivalent traces across engine variants, validating deterministic guarantees.

Result: Complete trace equivalence across tested engine variants for workflows without declared non-determinism.

6.2 Auditability Assessment

Domain experts can determine system behavior by inspecting protocol artifacts alone, without code inspection.

Result: Experts correctly identified workflow behavior, decision points, and side effects from artifact inspection.

6.3 Implementation Replaceability

Execution strategies can change without behavioral re-certification, provided conformance verification succeeds.

Result: Execution engine replacement achieved with conformance verification only. No behavioral re-certification required.

7 Related Work

Workflow engines such as those based on Petri nets [van der Aalst et al., 2003] and BPMN [Object Management Group, 2011] govern orchestration and control flow but typically delegate semantic authority to the code invoked at each step. Protocol-governed systems extend governance to the behavioral semantics of each step, not merely their sequencing.

Low-code platforms abstract coding but embed logic in proprietary representations [Waszkowski, 2019]. Behavior remains implementation-coupled, hidden behind visual abstractions.

TLA+ [Lamport, 2002], Alloy [Jackson, 2012], and Z [Spivey, 1989] enable rigorous behavioral specification but traditionally separate specification from execution. Protocol-governed architecture requires that specifications are the execution authority.

Microservices distribute implicit behavior across independently evolving components [Newman, 2015, Fowler and Lewis, 2014]. System behavior becomes emergent and non-local. Protocol-governed architecture maintains behavioral authority in explicit artifacts regardless of deployment topology.

The OMG's Model-Driven Architecture separates platform-independent models from platform-specific implementations [Object Management Group, 2014]. However, MDA focuses on code generation—the generated code becomes authoritative. Protocol governance inverts this: the model remains authoritative throughout execution.

Programmable Logic Controllers demonstrate long-standing separation of behavioral specification from execution hardware [Bolton, 2015]. Protocol-governed architecture applies this proven industrial pattern to business software.

Separation of concerns in software architecture. Parnas [Parnas, 1972] established that modular decomposition should be based on design decisions likely to change. Our layer decomposition follows this principle: each layer encapsulates a coherent set of design decisions. Dijkstra [Dijkstra, 1982] introduced separation of concerns as a principle for managing intellectual complexity. Our concern taxonomy operationalizes this principle by assigning machine-enforceable prefixes to each concern, making separation not merely a design aspiration but a runtime invariant. Buschmann et al. [Buschmann et al., 1996] cataloged architectural patterns including layered architectures. Our layer model differs in a critical respect: conventional layered architectures define dependency relationships between layers; protocol-governed layers define *lifecycle* relationships—each layer represents a phase in the progression from human intent to machine execution, not a dependency in the call graph. Kiczales et al. [Kiczales et al., 1997] identified cross-cutting concerns as aspects that span multiple modules. In our framework, concerns are not cross-cutting—they are orthogonal to layers. The Governance layer validates all concerns, but this is the constitutional function of the layer applied uniformly, not aspect-oriented cross-cutting.

8 Discussion

8.1 Limitations

Governance overhead: Protocol authoring requires explicit behavioral specification, imposing upfront cost that may not be justified for exploratory or short-lived systems.

Expressiveness constraints: Vocabulary-bounded behavior may prevent expression of legitimately needed capabilities until governance approves vocabulary expansion. The governance process for vocabulary extension must be efficient without being permissive.

Tooling maturity: The ecosystem for protocol-governed development is nascent compared to imperative programming environments.

Concern taxonomy granularity: The ten-concern taxonomy may prove insufficient as the architecture encounters new domains. The extension protocol (Section 3) addresses this, but extending the concern set is constitutionally expensive by design.

Layer-concern matrix stability: The orthogonality matrix (Table 2) reflects the current reference implementation. As the architecture evolves, some cells may change. The matrix should be treated as a constitutional snapshot, not an eternal truth.

8.2 Appropriate Domains

The architecture is optimized for systems requiring long-term auditability and compliance, deterministic and reproducible behavior, multi-stakeholder governance, resistance to semantic drift under evolution, and AI-assisted development with behavioral guarantees.

Examples include financial systems, healthcare records, regulatory compliance systems, and critical infrastructure.

8.3 Names as Governance

The renaming of "Common" to "Structure" illustrates a broader principle: names in a constitutional architecture carry normative force. They signal what belongs and, equally important, what does not belong. "Common" is a name that resists nothing—any artifact can plausibly claim to be common. "Structure" actively resists misuse: a utility function is not structural; a helper class is not structural. Only artifacts that define the system's invariant substrate qualify. The naming principle for constitutional architectures is that **names should resist misuse by their semantics, not by their documentation**.

8.4 Future Work

- Formal semantics enabling mechanical proof of behavioral properties
- Standardized conformance suites for engine certification
- Enhanced protocol authoring tooling
- Cross-organization governance for federated systems
- Systematic study of AI code generation under protocol constraint
- Formal proof that protocol-governed enforcement satisfies specific regulatory frameworks
- Empirical measurement of the governance deficit in AI-accelerated development environments

9 Conclusion

As AI accelerates code generation, implicit system behavior becomes untenable. Code evolves faster than humans can audit, while the business meaning embedded in that code drifts beyond reliable governance.

Protocol-governed architecture provides a path forward: systems whose behavior remains explicit, deterministic, and governable across implementation change. By separating what is dispensable (execution code) from what must endure (behavioral specification), organizations can embrace AI-driven development without sacrificing comprehension, auditability, or control.

The architecture is organized along two orthogonal constitutional primitives. **Layers**—Authoring, Governance, Protocol, Execution, Capability Transforms, Capability Side Effects, Transport, and Structure—partition the system as humans encounter it, following the lifecycle from legislative intent to structural invariant. **Concerns**—Actors, Intents, Workflows, Capability Contracts, Runtime Bindings, Capability Transforms, Capability Side Effects, Events, Transport Ingress, and Transport Egress—partition the system as machines encounter it, following the causal chain from actor initiation to evidence emission. Layers answer *where* a design responsibility resides; concerns answer *what* behavioral type an artifact carries. Conflating them produces systems that are neither clearly designed nor reliably governed. Formalizing them as independent primitives enables precise governance rules, unambiguous artifact placement, and machine-enforceable behavioral boundaries.

Each layer evolves independently; each concern carries deterministic execution semantics; neither contaminates the other. This achieves complete separation of concerns—not as a design aspiration but as a constitutional invariant enforced by the architecture itself.

This is not merely an architectural improvement—it is an architectural necessity for the AI era. When AI generates the majority of production code, the question "What does this system do?" cannot be answered by reading that code. It must be answered by consulting authoritative behavioral specifications that exist independently of any implementation.

Protocol-governed architecture provides those specifications—and the enforcement mechanisms to ensure they remain authoritative as implementations evolve.

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

Bolton, W. (2015). *Programmable Logic Controllers*. Newnes, 6th edition.

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

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.O., Kaplan, J., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Fowler, M. and Lewis, J. (2014). Microservices: a definition of this new architectural term. Martin Fowler. https://martinfowler.com/articles/microservices.html

GitHub (2023). GitHub Copilot research recitation. Technical report, GitHub.

Glass, R.L. (2002). *Facts and Fallacies of Software Engineering*. Addison-Wesley.

Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition.

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.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.

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.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *IEEE Symposium on Security and Privacy*, pages 754–768.

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.

Waszkowski, R. (2019). Low-Code Platform: A Revolution in Software Development? In *International Conference on Computational Collective Intelligence*, pages 299–310. Springer.

Weyuker, E.J. (1982). On testing non-testable programs. *The Computer Journal*, 25(4):465–470.