Protocol-Governed Software: Separating System Behavior from Execution Mechanics

Bachi
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 describe a seven-concern architectural model comprising governance, protocol, execution, pure capability transformations, controlled side effects, transport adapters, and authoring tools. 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 software 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)
Keywords: software architecture, declarative systems, deterministic execution, protocol governance, AI-generated code, auditability

1 Introduction

1.1 The Crisis of Implicit System Behavior

Contemporary software development operates under an implicit rule: to understand system behavior, one must read the code. Business rules, operational constraints, and failure semantics are embedded within imperative implementations, scattered across services, and intertwined with execution mechanics.

This architectural choice produces systemic failures:

- Behavior is not independently verifiable.
- Refactoring risks altering semantics without detection.
- Compliance and audit require code inspection rather than specification review.
- No authoritative artifact defines what behavior must be preserved.

As systems scale, behavior becomes an emergent property rather than a governed one.

1.2 AI-Generated Code as a Crisis Amplifier

Large language models now generate and refactor production code at scale. While tests may pass, subtle semantic drift—altered validation rules, edge-case handling, or execution ordering—can accumulate undetected.

Without authoritative behavioral specifications independent of implementation, verifying that AI-generated changes preserve system intent becomes infeasible. The industry faces a future where systems evolve faster than humans can audit, threatening long-term governability.

1.3 Why Existing Approaches Fall Short

Existing approaches address parts of the problem but not its root:

- **Workflow engines** govern orchestration, not behavior [2].
- **Low-code platforms** abstract coding but embed logic in proprietary representations [4].
- **Formal specification languages** define behavior but remain disconnected from execution [3].
- **Microservice architectures** distribute implicit behavior across independently evolving components [1, 5].

In all cases, behavior remains either non-executable, non-authoritative, or implementation-coupled.

1.4 Contribution

This paper presents a **protocol-governed architecture** that treats declarative protocol artifacts as authoritative, executable definitions of system behavior.

The architecture is distinguished by three constitutional principles:

1. **Protocol as Source of Truth**
   System behavior is defined entirely within version-controlled declarative artifacts, independent of execution engines.
2. **Deterministic Execution with Verification**
   Given identical protocol artifacts and inputs, all conformant engines must produce identical externally observable results.

3. **Governance Before Optimization**
   Structural correctness, determinism, and auditability are enforced before performance or execution strategy.

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:

1. **Trace Schema** defines observable execution events.
2. **Trace Capture** records execution outcomes.
3. **Replay Verification** ensures identical outcomes under replay.
4. **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 Seven-Concern Architectural Model

3.1 Overview

Protocol-governed systems decompose into seven orthogonal concerns. Each concern evolves independently, preventing architectural coupling.

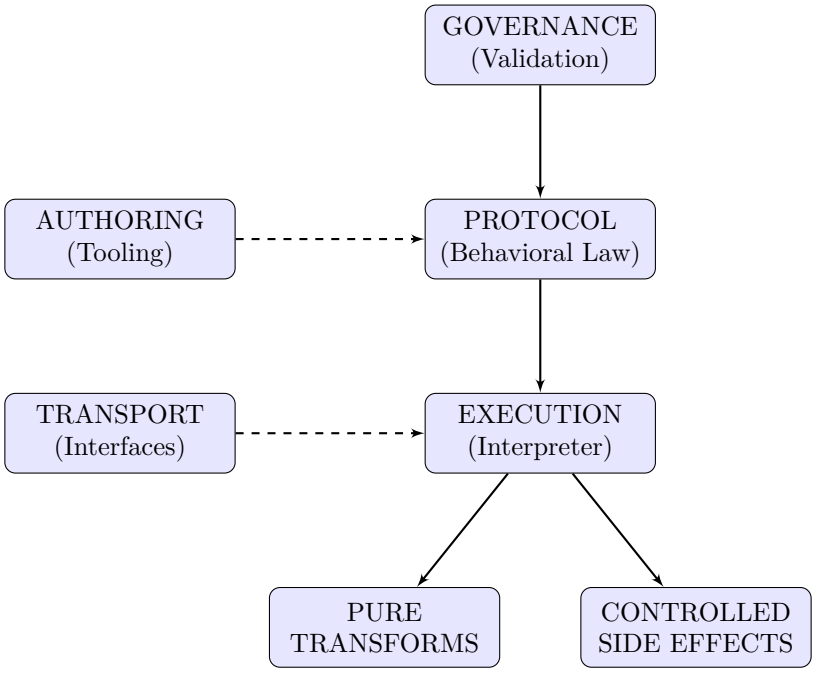

Figure 1: The Seven-Concern Architectural Model. Solid lines indicate runtime dependencies; dashed lines indicate design-time or interface interactions.

3.2 Governance Layer

Defines constitutional constraints and conformance rules. Governance validates protocols but does not execute them.

3.3 Protocol Layer

Pure declarative artifacts describing system behavior. No executable logic exists at this layer.

3.4 Execution Layer

Interprets protocols deterministically. Multiple implementations may exist, provided they satisfy conformance requirements.

3.5 Capability Transformations

Pure functions with no side effects. Identical inputs always produce identical outputs.

3.6 Controlled Side Effects

Explicitly declared mutations or external interactions. Effects are observable and ordered according to protocol semantics.

3.7 Transport and Authoring

Transport adapters expose execution via interfaces. Authoring tools assist protocol creation and validation but are never part of runtime execution.

4 Case Studies (Abstracted)

4.1 Cryptographic Workflow System

A prototype system implements hierarchical cryptographic workflows entirely through protocol artifacts. Deterministic transforms handle key derivation; controlled side effects persist metadata.

Execution traces verify identical outcomes across multiple engine strategies.

4.2 Immutable Append-Only System

An append-only system demonstrates protocol-governed state evolution. Integrity is enforced through declarative constraints rather than storage mechanics.

System state is derivable solely through event replay.

4.3 Protocol Evolution Without Implementation Change

Behavioral changes are introduced through new protocol versions while execution engines remain unchanged. Legacy and new behavior coexist without refactoring.

5 Evaluation

5.1 Determinism Verification

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

5.2 Auditability Assessment

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

5.3 Implementation Replaceability

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

6 Related Work

Workflow engines govern sequencing, not semantics [2]. Low-code platforms abstract coding without governable behavior [4]. Formal methods specify behavior but do not govern execution [3]. Microservices distribute implicit behavior across components [1, 5].

Protocol-governed architecture unifies specification and execution under a single authoritative model.

7 Discussion

Limitations

The approach is best suited for systems requiring explicit governance, auditability, and long-term behavioral stability. It is less appropriate for exploratory or highly non-deterministic domains.

Future Work

- formal engine verification,
- standardized conformance suites,
- protocol authoring tooling,
- potential standards development.

8 Conclusion

As AI accelerates code generation, implicit system behavior becomes untenable. Protocol-governed architecture provides a path toward software systems whose behavior remains explicit, deterministic, and governable across implementation change.

By separating behavioral intent from execution mechanics, systems can evolve without losing meaning—addressing a fundamental challenge in modern software engineering.

Author Information

Bachi
Contact: bachi.bachi@myyahoo.com

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

References

[1] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.

[2] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[3] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

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

[5] Martin Fowler and James Lewis. Microservices: a definition of this new architectural style. *Martin Fowler*, 2014. `https://martinfowler.com/articles/microservices.html`