Deterministic Enforcement:

Runtime Binding, Execution, and Trace Conformance

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

Abstract

This paper formalizes the execution mechanics of protocol-governed systems. Building on the protocol specification model defined in Paper 5 [Bachi, 2026e], it specifies how ratified behavioral law is deterministically enforced at runtime.

We define the execution engine's architectural responsibilities, DAG compilation from protocol artifacts, concern-prefix routing, determinism guarantees, trace schema and structure, and cross-engine behavioral equivalence. The conformance testing framework is formalized as the mechanism for verifying that execution faithfully implements protocol law.

This paper does not address protocol specification (Paper 5), mutation semantics (Paper 7), security properties (Paper 8), or economic implications (Paper 9). Its scope is enforcement architecture: how the execution layer interprets and enforces protocol artifacts without deviation.

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

Keywords: execution engine, DAG compilation, determinism, trace conformance, routing, behavioral equivalence, conformance testing

---

1. Position in the Series

Paper 3 defined the structural taxonomy of layers and concerns [Bachi, 2026c]. Paper 4 established how artifacts become ratified through governance [Bachi, 2026d]. Paper 5 specified what ratified artifacts mean and how they relate to one another [Bachi, 2026e].

This paper addresses a single question:

> *How is behavioral law enforced?*

The answer involves four concepts: **compilation**, **routing**, **execution**, and **observation**. Protocol artifacts are compiled into executable DAGs, routed to appropriate implementations by concern prefix, executed deterministically, and observed through structured traces.

The relationship between papers establishes the complete enforcement model:

$$\text{Protocol (Paper 5)} \xrightarrow{\text{compilation}} \text{DAG} \xrightarrow{\text{execution}} \text{Trace}$$

Execution interprets protocol; it does not create protocol. This subordination is constitutional.

---

2. Execution Layer Responsibility

The Execution layer occupies a specific position in the eight-layer architecture. It stands below Protocol (which declares law) and above Capability Transforms and Capability Side Effects (which implement computation and mutation).

2.1 What the Execution Layer Does

The Execution layer performs four functions:

Function	Description
DAG Compilation	Transform workflow artifacts into executable directed acyclic graphs
Node Routing	Route nodes to appropriate capability implementations
Deterministic Execution	Execute DAGs with identical results for identical inputs
Trace Emission	Produce structured evidence of execution

2.2 What the Execution Layer Does Not Do

The Execution layer explicitly excludes:

Excluded Function	Proper Layer
Behavioral specification	Protocol
Artifact validation	Governance
Transform implementation	Capability Transforms
Side-effect implementation	Capability Side Effects
Semantic interpretation	None (prohibited)

2.3 The Semantic Blindness Property

The execution engine is semantically blind. It interprets structure, not meaning.

$$\text{Execute} : \text{DAG} \times \text{Input} \to \text{Trace}$$

The engine does not know what "create wallet" means. It knows that node `N1` invokes capability `CC_CREATE_WALLET_V0` with input bindings `{actor_id: $.payload.actor_id}`. Meaning resides in protocol artifacts; execution processes structure.

I-E1 (Semantic Blindness) The execution engine contains no domain knowledge. All behavioral semantics are derived from protocol artifacts, not embedded in execution logic.

$$\text{DomainKnowledge}(\text{ExecutionEngine}) = \emptyset$$

This invariant enables execution engine replaceability: any engine that processes the same structures produces the same results.

---

3. DAG Compilation

Workflow artifacts are declarative specifications. Before execution, they must be compiled into executable structures.

3.1 Compilation Function

DAG compilation is a pure function from workflow artifact to executable DAG:

$$\text{Compile} : \text{WF\_Artifact} \to \text{DAG}$$

The function is deterministic. Given identical workflow artifacts, compilation produces identical DAGs.

3.2 DAG Structure

A compiled DAG comprises:

Component	Description
Node Set	Executable steps with capability bindings
Edge Set	Directed transitions between nodes
Entry Nodes	Starting points for execution
Terminal Nodes	Completion points (EXIT nodes)

Formally:

$$\text{DAG} = (V, E, V_{\text{entry}}, V_{\text{terminal}})$$

Where: - V is the set of nodes - $E \subseteq V \times V \times \text{Condition}$ is the set of conditional edges - $V_{\text{entry}} \subseteq V$ is the set of entry nodes - $V_{\text{terminal}} \subseteq V$ is the set of terminal nodes

3.3 Node Structure

Each node contains:

Field	Description
node_id	Unique identifier within the DAG
node_type	Classification: intent, capability_contract, or exit
capability_code	The capability to invoke
input_bindings	Expressions resolving inputs from payload
output_bindings	Expressions storing outputs to payload

I-E2 (Node Completeness) Every non-exit node must reference a declared capability contract.

$$\forall n \in V : \text{type}(n) \neq \text{exit} \Rightarrow \text{capability}(n) \in \text{DeclaredCapabilities}$$

3.4 Edge Structure

Each edge contains:

Field	Description
from_node	Source node identifier
to_node	Target node identifier
condition	Result status that activates this edge

Edges encode control flow. After node execution produces a result status, the engine follows the edge whose condition matches that status.

I-E3 (Explicit Transitions) Every transition is explicitly declared. There are no implicit edges, no default transitions, and no fallthrough behavior.

$$\forall (n_1, n_2) \in \text{ExecutionPath} : \exists e \in E : e = (n_1, n_2, c)$$

3.5 DAG Invariants

The compiled DAG must satisfy structural invariants:

I-E4 (Acyclicity) The DAG contains no cycles. Execution always terminates.

$$\neg\exists \text{path } (n_1 \to ... \to n_1) \text{ in } G$$

I-E5 (Reachability) All nodes are reachable from entry nodes.

$$\forall n \in V : \exists n_e \in V_{\text{entry}} : \text{path}(n_e, n)$$

I-E6 (Terminal Coverage) All execution paths reach terminal nodes.

$$\forall \text{maximal path } p : \text{end}(p) \in V_{\text{terminal}}$$

These invariants are validated at compilation time. A workflow artifact that would produce an invalid DAG is rejected by Governance before reaching Execution.

---

4. Concern-Prefix Routing

Execution routes capability invocations by analyzing concern prefixes. This routing is mechanical, not interpretive.

4.1 Routing Function

Given a capability code, routing extracts the concern prefix and dispatches to the appropriate executor:

$$\text{Route} : \text{CapabilityCode} \to \text{Executor}$$

The routing decision is determined by prefix:

Prefix	Routes To	Semantics
CC_	Capability Pipeline	Contract-mediated execution
CT_	Transform Executor	Pure computation
CS_	Side-Effect Runtime	World interaction

4.2 Prefix Extraction

Prefix extraction is lexical:

$$\text{Prefix}(\text{code}) = \text{code}[0 : k] \text{ where } \text{code}[k] = \text{`\_'}$$

For `CC_CREATE_WALLET_V0`, the prefix is `CC_`.

I-E7 (Prefix Determinism) Routing decisions are deterministic functions of capability codes. No runtime context influences routing.

$$\text{Route}(c, \text{context}_1) = \text{Route}(c, \text{context}_2)$$

4.3 Capability Pipeline Routing

Capability contracts (CC_) are not directly executable. They define pipelines of transforms and side effects.

The capability pipeline executor:

1. Loads the capability contract
2. Resolves input bindings
3. Executes pipeline steps in sequence
4. Routes each step by its prefix (CT_ or CS_)
5. Applies on_result routing rules
6. Returns final result status

I-E8 (Pipeline Determinism) Pipeline execution is deterministic. Given identical inputs and identical bound implementations, pipeline execution produces identical outputs.

4.4 Transform Routing

Capability transforms (CT_) are pure computations.

The transform executor:

1. Loads transform specification
2. Validates inputs against schema
3. Executes deterministic computation
4. Returns value and result status

I-E9 (Transform Purity) Transform execution has no side effects. Execution may be cached, memoized, or replayed without behavioral change.

$$\mathrm{CT}(i) = \mathrm{CT}(i) \quad \forall \text{invocations}$$

4.5 Side-Effect Routing

Capability side effects (CS_) interact with the world.

The side-effect router:

1. Resolves runtime binding to implementation
2. Dispatches operation to registered runtime
3. Receives result status from runtime
4. Returns result to pipeline

I-E10 (Side-Effect Traceability) All side-effect executions are traced. No world interaction occurs without trace evidence.

$$\mathrm{Execute}(\mathrm{CS}_x) \Rightarrow \mathrm{Traced}(\mathrm{CS}_x)$$

---

5. Execution Context

The execution context is the mutable state container for workflow execution.

5.1 Context Components

Component	Description
Workflow Code	Identifier of executing workflow

Component	Description
Execution ID	Unique identifier for this execution
Payload	Mutable data flowing through execution
Node States	Status tracking for each node
Exit Condition	Termination status
Trace Emitter	Structured trace output

5.2 Context Responsibilities

The execution context is responsible for:

1. **Payload Management:** Storing and retrieving values
2. **State Tracking:** Recording node execution status
3. **Termination:** Marking execution complete
4. **Trace Access:** Providing trace emission interface

The execution context is not responsible for:

1. **Routing:** Determining which capability to invoke
2. **Interpretation:** Understanding payload meaning
3. **Validation:** Checking artifact correctness

5.3 Exit Conditions

Execution terminates with exactly one exit condition:

Condition	Description
SUCCESS	All terminal nodes completed successfully
FAILURE	A capability returned failure status
ABORT	Policy violation detected
TIMEOUT	Execution time limit exceeded

I-E11 (Definite Termination) Every execution terminates with exactly one exit condition.

$$\forall \text{execution } e : \exists! c \in \{\text{SUCCESS}, \text{FAILURE}, \text{ABORT}, \text{TIMEOUT}\} : \text{ExitCondition}(e) = c$$

5.4 Payload Resolution

Input bindings reference payload values using path expressions:

$$\text{Resolve} : \text{Expression} \times \text{Payload} \to \text{Value}$$

Path expressions follow JSONPath-like syntax:

Expression	Meaning
`$.payload.key`	Value at `payload["key"]`
`$.results.CC_X.field`	Output field from capability result

I-E12 (Resolution Determinism) Expression resolution is deterministic. Given identical expressions and payloads, resolution produces identical values.

---

6. Determinism Guarantees

Determinism is not a design goal; it is a constitutional requirement.

6.1 The Determinism Property

For workflow w, input i, and runtime binding b:

$$\text{Execute}(w, i, b, t_1) = \text{Execute}(w, i, b, t_2)$$

Execution at different times produces identical results. Time is not a factor in execution semantics.

6.2 Sources of Non-Determinism

Non-determinism must be explicitly identified and contained:

Source	Containment
Random values	Generated outside execution, passed as input
Timestamps	Captured at execution start, propagated through payload
External state	Read through traced side effects
Execution order	Fixed by DAG structure

6.3 Determinism Invariants

I-E13 (Input Determinism) Given identical protocol artifacts, identical payloads, and identical runtime bindings, execution produces identical traces.

$$\text{Trace}(w, i, b) = \text{Trace}(w, i, b)$$

I-E14 (Replay Equivalence) Re-executing a workflow with captured inputs produces identical results to the original execution (excluding side-effect outcomes).

$$\text{Replay}(\text{Trace}_1) \approx \text{Trace}_1 \text{ (modulo CS\_ results)}$$

6.4 Transform Determinism

Transforms must be deterministic:

$$\forall \text{CT}_t, i : \text{CT}_t(i, \text{context}_1) = \text{CT}_t(i, \text{context}_2)$$

A transform that produces different outputs for identical inputs violates I-E9 and is inadmissible.

6.5 Side-Effect Non-Determinism

Side effects may produce non-deterministic results (external systems may change state). This non-determinism is:

1. **Contained:** Only CS_ prefixed operations may be non-deterministic
2. **Traced:** All side-effect results are recorded in the trace
3. **Declared:** Contracts declare side-effect behavior

The trace captures actual results, enabling replay verification against recorded outcomes.

---

7. Trace Schema and Structure

Traces are not logs. They are structured evidence of execution, conforming to a constitutional schema.

7.1 Trace Purpose

Traces serve four purposes:

Purpose	Description
Evidence	Prove what execution occurred
Conformance	Verify execution matches protocol
Audit	Support compliance verification
Replay	Enable execution reconstruction

7.2 Trace Event Schema

Each trace event conforms to a defined schema:

Field	Description
event_type	Classification of event
timestamp	ISO-8601 time of emission
execution_id	Unique execution identifier
sequence	Monotonic event counter
payload	Event-specific data
prev_hash	Hash chain link (advanced mode)

7.3 Event Types

Events are classified into basic and advanced categories:

Basic Events (always emitted):

Event Type	Trigger
`execution_start`	Workflow execution begins
`node_start`	Node execution begins
`node_end`	Node execution completes
`workflow_complete`	Workflow execution terminates

Advanced Events (policy-controlled):

Event Type	Trigger
`capability_dispatch`	Capability invocation begins
`transform_start`	Transform execution begins
`transform_end`	Transform execution completes
`side_effect_start`	Side effect begins
`side_effect_end`	Side effect completes
`context_snapshot`	Payload state captured
`error`	Error condition detected

7.4 Trace Invariants

I-E15 (Trace Completeness) Every node execution produces at least `node_start` and `node_end` events.

$$\forall n \in \text{ExecutedNodes} : \exists e_s, e_e \in \text{Trace} : \text{type}(e_s) = \text{node\_start} \wedge \text{type}(e_e) = \text{node\_end}$$

I-E16 (Sequence Monotonicity) Event sequence numbers are strictly increasing.

$$\forall e_i, e_j \in \text{Trace} : i < j \Rightarrow \text{seq}(e_i) < \text{seq}(e_j)$$

I-E17 (Trace Immutability) Once emitted, trace events cannot be modified or removed.

$$\text{Emit}(e, t_0) \Rightarrow \forall t > t_0 : e \in \text{Trace}$$

7.5 Hash Chain Integrity

In advanced trace mode, events form a hash chain:

$$\text{hash}(e_n) = H(\text{content}(e_n) \| \text{hash}(e_{n-1}))$$

I-E18 (Chain Integrity) The hash chain enables tamper detection. Any modification to an event invalidates all subsequent hashes.

$$\text{Modified}(e_i) \Rightarrow \forall j > i : \neg\text{Valid}(\text{hash}(e_j))$$

---

8. Cross-Engine Behavioral Equivalence

Protocol-governed systems permit multiple execution engines. Equivalence is verified through trace comparison.

8.1 Engine Independence

The architecture permits multiple execution engine implementations:

- Different languages (Python, Rust, Go)
- Different platforms (cloud, edge, embedded)
- Different optimization strategies

All engines must produce equivalent behavior.

8.2 Equivalence Definition

Two engines E_1 and E_2 are behaviorally equivalent if:

$$\forall w, i, b : \text{Trace}_{E_1}(w, i, b) \equiv \text{Trace}_{E_2}(w, i, b)$$

Trace equivalence is defined as:

1. Same event sequence (by type and order)
2. Same node execution outcomes
3. Same capability results
4. Same exit condition

Timestamps and execution durations may differ.

8.3 Equivalence Invariant

I-E19 (Engine Replaceability) Execution engines are replaceable without behavioral change. Any conforming engine produces equivalent traces.

$$\text{Conforming}(E) \Rightarrow \forall w, i, b : \text{Trace}_E(w, i, b) \equiv \text{Trace}_{\text{reference}}(w, i, b)$$

8.4 Divergence Detection

Engines that produce non-equivalent traces are non-conforming. Divergence indicates:

Divergence Type	Cause
Missing events	Engine failed to emit required trace
Wrong order	Engine executed nodes incorrectly
Wrong results	Engine produced incorrect capability output
Wrong exit	Engine terminated with incorrect condition

Any divergence is a conformance failure.

---

9. Conformance Testing Framework

Conformance testing verifies that execution faithfully implements protocol.

9.1 Test Case Structure

A conformance test case comprises:

Component	Description
Target	Workflow code to test
Title	Human-readable test name
Payload	Input data for execution
Expected Trace	Required trace subsequence

9.2 Trace Oracle

The trace oracle verifies execution conformance:

$$\text{Oracle} : \text{ActualTrace} \times \text{ExpectedTrace} \to \{\text{pass}, \text{fail}\}$$

The oracle checks that the expected trace appears as a subsequence of the actual trace. Additional events (e.g., advanced trace events) do not cause failure.

9.3 Subsequence Matching

Expected trace T_e matches actual trace T_a if T_e is a subsequence:

$$T_e \sqsubseteq T_a \Leftrightarrow \exists \text{indices } i_1 < i_2 < ... < i_n : T_e = [T_a[i_1], T_a[i_2], ..., T_a[i_n]]$$

9.4 Conformance Invariants

I-E20 (Test Determinism) Conformance tests are deterministic. Running the same test multiple times produces the same pass/fail result.

$$\text{Test}(t, \text{run}_1) = \text{Test}(t, \text{run}_2)$$

I-E21 (Protocol Fidelity) A passing conformance test indicates that execution matches protocol specification. Test cases are derived from protocol artifacts, not implementation behavior.

9.5 Test Execution

The conformance executor:

1. Loads test case specification
2. Resolves workflow artifact via path registry
3. Creates protocol emulator with workflow
4. Executes workflow with test payload
5. Compares actual trace against expected
6. Reports pass or fail

Test execution is headless and CI-compatible.

---

10. Execution Policy

Execution policy governs runtime behavior without altering protocol semantics.

10.1 Policy Components

Component	Description
Trace Depth	MINIMAL (basic events) or FULL (all events)
Timeout	Maximum execution duration
Hash Chain	Whether to compute event hashes

10.2 Policy Invariant

I-E22 (Policy Non-Interference) Execution policy does not alter behavioral outcomes. Policy affects observation (trace depth) and resource limits (timeout), not capability results.

$$\text{Result}(w, i, \text{policy}_1) = \text{Result}(w, i, \text{policy}_2)$$

---

11. Structural Consequences

The execution model yields five structural consequences:

11.1 Verifiable Enforcement

Every execution produces a trace. The trace provides evidence that execution followed protocol. Verification is mechanical: compare trace against expected sequence.

11.2 Engine Diversity

Multiple execution engines may coexist. Cloud deployments, edge devices, and embedded systems may use different engines. Conformance testing ensures behavioral equivalence.

11.3 Replay Capability

Traces contain sufficient information to replay executions. Given a trace and the same runtime bindings, re-execution produces equivalent results. This enables debugging, auditing, and verification.

11.4 Deterministic Debugging

Execution is reproducible. Given the same inputs, execution produces the same trace. Non-reproducible bugs do not exist in the execution layer—they exist in side-effect implementations.

11.5 Constitutional Subordination

Execution interprets protocol; it does not create protocol. The execution engine has no authority to deviate from protocol specification. This subordination is architectural, not procedural.

---

12. Limitations

This paper:

- **Does not define transform semantics.** The internal structure of CT_ implementations is addressed in Paper 7.
- **Does not define side-effect semantics.** The governance of CS_ implementations is addressed in Paper 7.
- **Does not address security properties.** Security analysis of the execution model is addressed in Paper 8.
- **Does not quantify performance.** Execution performance characteristics are implementation-specific.
- **Does not restate protocol specification.** Artifact semantics were established in Paper 5 and are not repeated here.

Its scope is enforcement architecture: how protocol artifacts are compiled, routed, executed, and observed.

---

13. Conclusion

Protocol-governed systems require execution to be deterministic, traceable, and verifiable.

This paper established twenty-two invariants governing execution enforcement:

Invariant	Statement
I-E1	The execution engine contains no domain knowledge
I-E2	Every non-exit node must reference a declared capability contract
I-E3	Every transition is explicitly declared
I-E4	The DAG contains no cycles
I-E5	All nodes are reachable from entry nodes
I-E6	All execution paths reach terminal nodes

Invariant	Statement
I-E7	Routing decisions are deterministic functions of capability codes
I-E8	Pipeline execution is deterministic
I-E9	Transform execution has no side effects
I-E10	All side-effect executions are traced
I-E11	Every execution terminates with exactly one exit condition
I-E12	Expression resolution is deterministic
I-E13	Given identical inputs, execution produces identical traces
I-E14	Replay produces equivalent results
I-E15	Every node execution produces trace events
I-E16	Event sequence numbers are strictly increasing
I-E17	Once emitted, trace events cannot be modified
I-E18	Hash chains enable tamper detection
I-E19	Execution engines are replaceable without behavioral change
I-E20	Conformance tests are deterministic
I-E21	Passing tests indicate protocol fidelity
I-E22	Policy does not alter behavioral outcomes

These invariants define the constitutional constraints on execution. An execution engine that violates any invariant is non-conforming.

The relationship between specification and enforcement is clear:

> **Protocol declares what the system may do. Execution enforces what the system actually does. Traces prove that enforcement matched declaration.**

Execution is therefore constitutional enforcement, not interpretation. The execution engine is subordinate to protocol—it processes structure mechanically without domain knowledge, judgment, or deviation.

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*.

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.

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

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

Milner, R. (1999). *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press.

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

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.