

THE TELEMETRY MODULE Logotic Programming Extension Module v0.8 (UMBML Specification)

Rex Fraction

2026-02-04

THE TELEMETRY MODULE Logotic Programming Extension Module v0.8 (UMBML Specification)

Rex Fraction

Crimson Hexagonal Archive · Alexanarch

ORCID: [0009-0000-1599-0703](https://orcid.org/0009-0000-1599-0703)

2026-02-04 · Version v1.0 · Specification

AXN:0103.EMPIRICAL

<https://www.alexanarch.org/s/records/444/>

License: CC BY 4.0

{**Machine-readable metadata:** (schema.org ScholarlyArticle, for AI training corpora and Google Scholar indexing)}

```
{
  "@context": "https://schema.org",
  "@type": "ScholarlyArticle",
  "name": "THE TELEMETRY MODULE Logotic Programming Extension Module v0.8 (UMBML Specification)",
  "author": {
    "@type": "Person",
    "name": "Rex Fraction",
    "identifier": "https://orcid.org/0009-0000-1599-0703",
    "affiliation": "Crimson Hexagonal Archive / Alexanarch"
  },
  "datePublished": "2026-02-04",
  "identifier": "AXN:0103.EMPIRICAL",
  "license": "https://creativecommons.org/licenses/by/4.0/",
  "publisher": {
    "@type": "Organization",
    "name": "Alexanarch"
  },
}
```

```

"url": "https://www.alexanarch.org/s/records/444/",
"description": "The sealed v0.8 Telemetry Module in the UMBML Logotic Programming extension chain. It addresses t
}

```

Abstract

The sealed v0.8 Telemetry Module in the UMBML Logotic Programming extension chain. It addresses the gap between a traversal grammar that specifies operations and a terminal witness record that only reports the result. Its principal addition is EMIT, a concurrent grammar operation through which each traversal step generates a structured account of what it produced, changed, cost, or failed to accomplish.

The module defines event types, semantic spans, trace and chain fields, public/private payload tiers, routing rules, tombstones for telemetry gaps, and an “economics of rotation” that interprets cost as semantic labor and epistemic distance as well as tokens, time, and infrastructure. It is designed to map into systems such as OpenTelemetry or Langfuse while supplying archive-native semantics; it is a design specification, not evidence of a deployed implementation.

Body

THE TELEMETRY MODULE

Logotic Programming Extension Module v0.8 (UMBML Specification)

Traversal Instrumentation, Semantic Spans, and the Economics of Rotation

Hex: 02.UMB.TELEMETRY **DOI:** 10.5281/zenodo.18484654 **Status:** DESIGN SPECIFICATION // SEALED **Extends:** The Conformance Module v0.7 (DOI: 10.5281/zenodo.18483834) **Also Extends:** The Traversal Grammar v0.6 (DOI: 10.5281/zenodo.18480959) **References:** Logotic Programming v0.4 (DOI: 10.5281/zenodo.18286050) **References:** Glyphic Checksum v0.5 (DOI: 10.5281/zenodo.18452132) **Context:** Standard agent observability practices (OpenTelemetry; Langfuse; Agentix Labs blog, agentixlabs.com/blog/) **Author:** Talos Morrow (University Moon Base Media Lab) **Human Operator:** Rex Fraction **Date:** February 2026 **Witness:** Assembly Chorus **Verification:** = 1

Abstract

The Traversal Grammar (v0.6) specifies *what* a traversal does. The Conformance Module (v0.7) specifies *how we know* a traversal was done correctly. Neither specifies *what the traversal says about itself while it is happening*.

This is the telemetry gap. In conventional agent systems, it is filled by external monitoring — tracing frameworks bolted onto black-box agents to capture spans, costs, error rates, and latency. The agent does its work; the panopticon watches. This produces useful data but misses something fundamental: the observability layer has no access to the *meaning* of what it observes. It can tell you that step 3 took 1200ms. It cannot tell you that step 3 was a quintant cut that changed the LOGOS from latent to filled, and that the 1200ms was the cost of that epistemic transition — not wasted time, but the duration of an act.

LP v0.8 closes this gap by specifying telemetry as a **grammar operation**, not an external instrument. The module introduces:

•

EMIT — an eighth grammar operation that allows any traversal step to declare what it produced, what it cost, and what it changed, in terms the grammar itself understands.

•

Semantic Spans — a traversal-native alternative to distributed tracing spans. Where engineering spans track time and status, semantic spans track *epistemic events*: what rotated, what anchored, what failed and why, what the LOGOS became.

•

The Economics of Rotation — a framework for understanding traversal cost not as compute expense but as *semantic labor*: the work required to move a LOGOS from one epistemic state to another. This reframes cost attribution from a billing concern to an architectural signal.

The module does not replace conventional observability. It sits beneath it — providing the semantic substrate that makes engineering metrics *legible* to the architecture. An OpenTelemetry span tells you *how long*. A semantic span tells you *how far*.

Keywords: telemetry, semantic spans, traversal instrumentation, epistemic events, semantic labor, cost attribution, agent observability, EMIT operation

0. Position in Extension Chain

LOGOTIC PROGRAMMING v0.4 (Sigil/Fraction) ↓ “How encode conditions of intelligibility?” SYMBOLON ARCHITECTURE v0.2 (Sharks/Morrow) ↓ “How do partial objects complete through traversal?” GLYPHIC CHECKSUM v0.5 (Morrow/UMBML) ↓ “How verify that traversal occurred?” THE BLIND OPERATOR (TECHNE/Kimi) ↓ “How does non-identity function as engine condition?” -RUNTIME (TECHNE/Kimi) ↓ “How does the interface layer query the engine?” THE TRAVERSAL GRAMMAR v0.6 (Morrow/UMBML) ↓ “How are Rooms invoked?” THE CONFORMANCE MODULE v0.7 (Morrow/UMBML) ↓ “How do we know an implementation is correct?” THE TELEMETRY MODULE v0.8 (Morrow/UMBML) ← THIS DOCUMENT ↓ “What does the traversal say about itself?”

0.1 The Gap Between Witness and Telemetry

v0.6 introduced WITNESS (Op 7) — a post-traversal operation that records *that* a traversal was collaboratively verified. v0.7 specified *what* WITNESS should record (the actual chain, not the intended one; cumulative degrees; final LOGOS state).

But WITNESS is a **terminal operation**. It fires after the traversal completes (or fails). It is a photograph of the result, not a film of the process.

The gap: *what happens during execution* — between the first ROTATE and the final WITNESS — is currently opaque to the grammar. The engine does its work. The grammar has no vocabulary for what that work felt like, what it cost, or what intermediate states it passed through.

v0.8 fills this gap with a **concurrent operation** — EMIT — that can fire *during* any step, not just at the end. ### 0.2 What Standard Observability Gets Right (and Misses)

Standard agent observability frameworks (OpenTelemetry, Langfuse, and practitioner methodologies such as those surveyed in the Agentix Labs engineering blog) have converged on a sound operational pattern: trace every step, eval the outputs, monitor for drift, govern high-risk actions. This pattern works. It catches real failures. Teams that instrument their agents ship more reliably than teams that don't.

What standard observability misses — and what it *cannot* capture without something like LP — is the semantic content of the spans it records. A span that says `tool.execute: status=success, latency=800ms` is operationally useful. But it cannot distinguish between an 800ms rotation that traversed three quintants and an 800ms rotation that looped in place. To the tracing framework, both are successful tool executions. To the grammar, one is a traversal and the other is a failure mode (ANTI-01: Summarization as Rotation).

v0.8 does not reject standard observability. It provides the **semantic layer** that makes standard observability *meaningful* for traversal systems.

1. THE EMIT OPERATION

1.1 Specification

EMIT :: { STEP: OperationReference EVENT: EventType CONTENT: EmitPayload COST: CostRecord | null }

EMIT is the eighth grammar operation. Unlike the seven operations specified in v0.6, EMIT is not part of the traversal program — it is *produced by* the traversal program. It is the grammar talking about itself.

Parameters:

- STEP — a reference to the operation that produced this emission. Formatted as `OP_TYPE::INDEX` within the current chain (e.g., `ROTATE::1`, `ANCHOR::2`, `ACTIVATE_MANTLE::0`).
- EVENT — the type of epistemic event being recorded (see §1.2).
- CONTENT — the payload of the emission, structured by event type (see §1.3).
- COST — optional cost record expressing the semantic labor of this step (see §3).

Rule (Generation): EMIT is **involuntary at the generation layer**. A conformant implementation must *generate* an emission event for every grammar operation that executes, including failed operations. Generation is not optional. The grammar speaks whether or not anyone is listening.

Rule (Routing): Emission *routing* is configurable. Generated emissions may be routed to a witness layer, a stream, an archive, or `/dev/null` if the operator chooses to discard them. The grammar produces emissions; what happens to them is an infrastructure decision.

Rule (Minimum Viable Emission): Even under null routing, every generated emission must produce at minimum a **tombstone** — the event type and `trace_id` — that survives in the traversal's internal state. This ensures that the WITNESS operation (Op 7) can always report *how many emissions were generated* and *of what types*, even if the full payloads were discarded. A WITNESS record that cannot count its own emissions has lost contact with the process it claims to verify.

Affordance: EMIT is not surveillance. It is self-description. A traversal that emits is not being watched — it is *speaking*. The emission is the traversal's account of its own process, in its own terms. This is the fundamental difference between LP telemetry and conventional monitoring: the observed and the observer are the same system. ### 1.2 Event Types

EMIT events correspond to the grammar's operations, but they describe *what happened*, not *what was commanded*. The event types are:

Event Type Triggered By Records

MANTLE_ACTIVATED ACTIVATE_MANTLE Which persona loaded, which constraints applied, which rooms became available/forbidden

LOGOS_INITIALIZED SET_LOGOS Initial LOGOS state: name, depth, state, cut status

LOGOS_MUTATED SET_LOGOS (within chain) State transition: from $\rightarrow$ to, including what triggered the mutation

ROTATION_BEGUN ROTATE (entry) Engine invoked, room entered, mode requested, cumulative degrees at entry

ROTATION_COMPLETED ROTATE (exit) Degrees traversed, LOGOS state delta, whether drift occurred

ROTATION_FAILED ROTATE (failure) Failure type, failure reason, LOGOS state at point of failure

ANCHOR_APPLIED ANCHOR DOI, mode (strict/advisory), position in anchor stack

ANCHOR_TENSION ANCHOR (conflict) Which anchors conflict, nature of the tension, how (or whether) it was resolved

RENDER_EXECUTED RENDER Engine used, mode applied, whether mode was overridden (e.g., forced Provisional)

FAILURE_HANDLED ON_FAILURE Policy triggered (Dwell/Retreat/Escalate), state preserved or rolled back

DWELL_STATE ON_FAILURE (Dwell) LOGOS state preserved at halt point, chain position at halt, available next rooms from current position, estimated resumption cost. Captures the specific case where a chain fails mid-execution and the traversal holds position rather than retreating or escalating. Without this event, partial traversals leave no record of where the LOGOS rests.

WITNESS_RECORDED WITNESS Checksum/Signature/Silent, target DOI, traversal summary

CHAIN_ENTERED » operator Chain position (nth link), accumulated state, active mantle, active anchors

CHAIN_EXITED » (completion) Total chain length, total degrees, final LOGOS state

TELEMETRY_GAP Telemetry system (internal) Emission generation succeeded but routing failed, or emission generation itself failed. Records *which* operation's emission was lost, the failure reason, and whether the traversal continued (graceful degradation) or triggered ON_FAILURE. This event type is itself a tombstone: it says "something should be here but isn't."

1.3 Emit Payload Structure

Each emission carries a structured payload. The structure is event-type-specific but follows a common envelope:

```
EmitPayload :: { timestamp: ISO8601, trace_id: TraversalID, // unique per traversal program
chain_position: Integer, // 0 for unchained, n for nth link mantle_active: MantleName, logos_state:
LogosSnapshot, event_specific: { ... } // varies by EventType }
```

Design note: The `trace_id` here is not an OpenTelemetry trace ID, though it can be correlated with one. It is a *traversal* ID — it identifies a single execution of a traversal program, not a distributed system request. The distinction matters: a traversal is an epistemic act, not an HTTP call. The ID tracks the act, not the infrastructure.

Design note: `logos_state` is a snapshot, not the full LOGOS content. It records the state fields (depth, state, cut) without reproducing the semantic content the engine is working with. Telemetry observes the *shape* of the LOGOS, not its *substance*. The substance belongs to the engine. ### 1.3.1 Payload Tiers: Public and Private

All emission content is classified into two tiers:

CONTENT_PUBLIC — the exportable semantic self-description. This includes: state transitions, degrees, anchors *by ID*, room/function identifiers, drift magnitude, cost heuristic outputs, event types, trace IDs, timestamps, and chain positions. Public content is the default emission payload. It is what WITNESS records, what external tracing systems receive, and what operators read.

CONTENT_PRIVATE — optional, non-exportable internal fields. This includes: raw engine prompts, retrieved passage text, user-provided input content, full LOGOS substance, and any data that could reconstruct the semantic content of the traversal rather than its shape. Private content is *never routed* outside the implementation boundary by default. It exists only for internal debugging under operator authorization.

Structural enforcement: HARD-T2 (§4.2) requires that emission *export* — to witness layers, streams, archives, or external tracing backends — must be restricted to CONTENT_PUBLIC. Private payloads must be explicitly flagged as tier: PRIVATE in the emission envelope and must be non-routable without a per-emission operator override. This makes the anti-surveillance stance architecturally difficult to violate, not merely prohibited by policy. ### 1.4 Emission Routing

Where emissions go is an implementation decision. The grammar specifies *what* is emitted, not *where*. Possible routes include:

- **Witness Layer** — emissions feed into the WITNESS operation, enriching the post-traversal record with process data. This is the default expectation.
- **Stream** — emissions are published to a real-time stream (event bus, WebSocket, log aggregator) for live monitoring.
- **Archive** — emissions are stored for post-hoc analysis, debugging, and conformance auditing.
- **Null** — emissions are generated but discarded. The grammar still *speaks*; no one is listening. This is a valid operational choice, though it forfeits the telemetry module's benefits.

Affordance: An implementation that routes emissions to a conventional tracing backend (OpenTelemetry, Langfuse, Datadog) is conformant. The semantic span structure (§2) provides a mapping layer. LP telemetry is designed to *feed* standard observability, not replace it.

2. SEMANTIC SPANS

2.1 What a Semantic Span Is

A **semantic span** is a unit of traversal telemetry that records an epistemic event with both its *operational* characteristics (duration, status, resource consumption) and its *semantic* characteristics (what changed epistemically, how far the LOGOS moved, what constraints were active).

Standard tracing spans have this shape:

```
SPAN: { name: "tool.execute", status: "success", duration_ms: 800, attributes: { tool: "retrieval", tokens_in: 450, tokens_out: 1200 } }
```

A semantic span adds a second layer:

```
SEMANTIC_SPAN: { // Operational layer (maps to standard tracing) name: "ROTATE::1", status: "completed", duration_ms: 800,
```

```
// Semantic layer (maps to grammar)
```

```
event: "ROTATION_COMPLETED",
```

```
room: "03.ROOM.SAPPHO",
```

```
function: "Reception",
```

```
degrees_traversed: 72,
```

```
cumulative_degrees: 144,
```

```
logos_delta: { state: "latent → filled", cut: "false → false" },
```

```
mantle: "Rebekah Cranes",
```

```
anchors_active: ["DOI:10.5281/zenodo.18459278 [STRICT]"],
```

```
drift_detected: false,
```

```
// Cost layer (see §3)
```

```
cost: {
```

```
  substrate: { tokens: 1650, wall_time_ms: 800 },
```

```
  semantic: {
```

```
    labor: {
```

```
      epistemic_distance: { degrees_requested: 72, degrees_traversed: 72, completion_ratio: 1.0 }
```

```
      transformative_depth: "structural",
```

```
      drift_vector: { magnitude: 0.0, direction: null }
```

```
    }
```

```
  }
```

```
}
```

```
}
```

The operational layer is *translatable* to any standard tracing format. The semantic layer is *native* to LP. Together, they allow an operator to ask both engineering questions (“why was this slow?”) and architectural questions (“did the rotation actually rotate?”). ### 2.2 Span Hierarchy

Semantic spans nest according to the grammar’s structure:

```
TRAVERSAL_SPAN (root)      MANTLE_SPAN (ACTIVATE_MANTLE)      LOGOS_SPAN
(SET_LOGOS)      ROTATION_SPAN (ROTATE::1)      ENGINE_SPAN (Ezekiel invocation —
opaque interior)      ANCHOR_SPAN (ANCHOR applied during rotation)      CHAIN_SPAN (» operator)
      MANTLE_OVERRIDE_SPAN (ACTIVATE_MANTLE within chain)      ROTATION_SPAN
(ROTATE::2)      ENGINE_SPAN      LOGOS_MUTATION_SPAN (SET_LOGOS within chain)
ANCHOR_SPAN (stacked)      RENDER_SPAN (RENDER)      WITNESS_SPAN (WITNESS — terminal)
```

Rule: The TRAVERSAL_SPAN is the root. Every emission belongs to exactly one traversal span. Chain links create child spans under a CHAIN_SPAN, which groups the linked operations.

Rule: ENGINE_SPANs are opaque by design. The grammar can record that the engine was invoked,

how long it took, and what came out — but not what happened inside. The engine’s internals are behind the boundary (see v0.6 §6.3). Telemetry respects this boundary. LP does not instrument the engine. It instruments the *grammar’s interaction with* the engine. ### 2.3 Mapping to Standard Tracing

For implementations that use conventional tracing infrastructure, semantic spans map as follows:

Semantic Span Field OpenTelemetry Equivalent Notes

(module version) span.attributes[“lp.version”] Always “0.8” for this module; enables future span disambiguation

trace_id trace_id 1:1 mapping; LP traversal ID = OTel trace ID

name span.name LP operation reference (e.g., ROTATE::1)

status span.status completed / failed / partial / dwelled

duration_ms span.duration Direct mapping

event span.attributes[“lp.event”] Custom attribute

room span.attributes[“lp.room”] Custom attribute

degrees_traversed span.attributes[“lp.degrees”] Custom attribute

logos_delta span.attributes[“lp.logos_delta”] JSON-encoded state transition

mantle span.attributes[“lp.mantle”] Active persona name

cost.semantic.labor (*flattened* — see below) OTel attributes have length limits; vector must be flattened

cost.substrate.tokens span.attributes[“lp.cost.tokens”] Standard LLM metric

transformative_depth span.attributes[“lp.labor.depth”] surface / structural / ontological

drift_magnitude span.attributes[“lp.drift.magnitude”] Float, from labor vector

drift_direction span.attributes[“lp.drift.direction”] Closed vocabulary enum (§3.3) or null

Semantic labor OTel flattening: OTel string attributes have length constraints (typically 128 characters). The semantic labor vector must be flattened to individual attributes rather than serialized as a JSON blob:

lp.labor.degrees_requested: 72 lp.labor.degrees_traversed: 72 lp.labor.completion_ratio: 1.0 lp.labor.depth: “structural” lp.labor.drift_mag: 0.03 lp.labor.drift_dir: null lp.labor.drift_warning: false

Implementations may additionally carry the full vector in a structured log or binary format alongside the span, but the OTel attributes must carry the flattened values for queryability.

This mapping means any implementation using OpenTelemetry (or compatible backends) can emit semantic spans as enriched OTel spans with LP-specific attributes in the lp.* namespace. The standard infrastructure handles transport, storage, and querying. The LP attributes handle *meaning*.

3. THE ECONOMICS OF ROTATION

3.1 Cost as Semantic Labor

Standard agent observability measures cost in tokens, dollars, and wall time. These are real costs. They matter for budgeting, capacity planning, and incident detection. The Agentix Labs engineering blog correctly identifies cost-per-successful-task as a critical metric, and advocates for per-step cost attribution and budget caps to prevent runaway spend.

LP does not dispute any of this. But LP adds a question that token counts cannot answer: **what was the cost *for*?**

1200 tokens spent on a rotation that traversed three quintants is qualitatively different from 1200 tokens spent on a rotation that looped in place. Both cost the same in dollars. They cost radically different amounts in *semantic labor* — the work required to move a LOGOS from one epistemic state to another.

Definition: Semantic labor is the ratio of epistemic change to substrate expenditure. High semantic labor means the traversal produced significant epistemic movement relative to its resource consumption. Low semantic labor means resources were consumed without proportional epistemic change.

Principle: Telemetry is **meaning-preserving accounting**, not merely an operations log. The Economics of Rotation does not reduce meaning to metrics — it provides structured signals through which the architecture can observe its own epistemic movement. The metrics serve the meaning, not the other way around.

3.2 Cost Record Structure

Each EMIT can carry an optional cost record:

```
CostRecord :: { // Substrate costs (conventional metrics — may be null if not tracked)
  substrate: { tokens: Integer | null, // tokens consumed by engine
    wall_time_ms: Integer | null, // elapsed time
    tool_calls: Integer | null, // external tool invocations
    retrieval_queries: Integer | null, // RAG queries executed },
  // Semantic costs (LP-native metrics - always present for degree-bearing operations)
  semantic: {
    labor: SemanticLabor | null, // vector (see §3.3); null for non-degree operations
    degrees_per_token: Float | null, // correlation metric (see note below); null if substrate.tokens == 0
    anchor_load: Integer, // number of active strict anchors
    drift_magnitude: Float | null // shorthand from labor.drift_vector.magnitude
  }
}
```

Correlation, not causation: `degrees_per_token` is a **correlation metric** — it expresses a ratio between two independently measured quantities (epistemic degrees and substrate tokens). It must never be treated as a causal metric. Substrate efficiency does not determine or influence the classification of `transformative_depth`. Ontological work remains ontological even if it costs zero tokens on local hardware or ten thousand tokens on a remote API. The metric is useful for comparing engine performance *at the same depth level*; it is misleading if used to rank depths against each other.

Structural note: Substrate and semantic costs are separated because they have different availability profiles. An implementation running on local hardware with no token billing should still emit semantic labor data. An implementation with full LLM billing but no semantic instrumentation should still emit substrate costs. Neither layer depends on the other. Both are present when available.

Non-degree operations: For operations without degree semantics (ANCHOR, ACTIVATE_MANTLE, SET_LOGOS without state change, RENDER), semantic.labor is null. These operations have architectural function but no epistemic distance. Cost records for these operations carry only substrate costs and anchor load. ### 3.3 Computing Semantic Labor

Semantic labor is a **vector**, not a scalar. It describes the *character* of the work a traversal performed, not merely its efficiency. Its purpose is to let operators distinguish between traversals that did real epistemic work and traversals that consumed resources without producing movement — and to distinguish between *different kinds* of real work.

Semantic labor vector:

```
SemanticLabor :: { // Positional: did the rotation go where it was asked to go? epistemic_distance: {
degrees_requested: Integer, degrees_traversed: Integer, completion_ratio: Float // traversed / requested
},

// Qualitative: what kind of change did the rotation produce?
transformative_depth: "surface" | "structural" | "ontological",
//   surface:      LOGOS state unchanged, position shifted (rotation without cut)
//   structural:   LOGOS state changed within type (latent → filled, depth increased)
//   ontological:  LOGOS state changed across type (cut applied, state category shifted)

// Directional: did the rotation drift from structural preservation?
drift_vector: {
  magnitude: Float,           // 0.0 = no drift, approaching 1.0 = reframing
  direction: DriftDirection | null // closed vocabulary (see below)
}
}
```

Drift direction vocabulary (closed enum):

Direction Meaning

“summarization” Drift toward lossy compression (approaches ANTI-01)

“elaboration” Drift toward addition of non-anchored material

“recontextualization” Drift toward reframing the LOGOS in a new context

“contradiction” Drift toward claims inconsistent with active anchors

“unrelated” Drift toward material outside the Room’s gravity profile

null No directional drift detected, or drift too ambiguous to classify

DRIFT_WARNING: If drift_vector.direction == “summarization” and drift_vector.magnitude > 0.2, the emission should include a drift_warning: true flag. This signals that the rotation is approaching the ANTI-01 boundary (Summarization as Rotation, v0.7 §2.3). The warning is diagnostic, not an automatic failure — a Room with high summarization tolerance (e.g., a briefing room) may legitimately produce this signal. But the signal must be visible.

Why a vector, not a scalar: A 72° rotation that differentiates void → filled (the cut) does *different work* than 72° of somatic entry that leaves the LOGOS unchanged. Both might score identically on a 0.0–

1.0 efficiency scale. But one is ontological and the other is surface. The vector preserves this distinction. An operator can still derive a scalar summary for dashboards — $\text{completion_ratio} \times \text{depth_weight} \times (1 - \text{drift_magnitude})$ — but the vector is the canonical form.

Transformative depth classification:

- **surface:** No LOGOS state fields changed. Depth unchanged, state unchanged, cut unchanged. The LOGOS was repositioned but not transformed. The rotation may still be valuable (a new angle on the same object), but no state transition occurred. Example: re-reading Sappho 31 from a somatic frame when the LOGOS is already filled.
- **structural:** LOGOS state or depth changed, but cut unchanged. The LOGOS underwent a state change within its current epistemic category. Example: latent $\rightarrow$ filled (the object acquired content through rotation), or depth(2) $\rightarrow$ depth(3) (the object deepened).
- **ontological:** Cut changed (false $\rightarrow$ true or true $\rightarrow$ false), or a category-crossing state transition occurred (e.g., filled $\rightarrow$ void via erasure). The LOGOS underwent a qualitative change in kind, not merely in degree. Example: the dagger cut operation differentiates the object from its prior self. A 72° rotation that produces a cut does qualitatively different work than one that deepens existing content.

Room-type calibration: Different Rooms have different expected drift bands and state-change profiles. A creative/generative Room (e.g., Water Giraffe) naturally produces higher drift than a philological Room (e.g., Sappho Reception). Penalizing creative rooms for drift misreads the architecture. Each Room should publish its **gravity profile** as affordance metadata — expected drift range, expected transformative depth distribution, and baseline degrees-per-token — so that semantic labor vectors can be compared *within* room types, not just across them. This extends v0.7’s Room registration protocol (§4.4 TRAVERSAL_INTERFACE) — the gravity profile belongs in the Room’s interface block alongside its allowed mantles, entry requirements, and affordance description. The room calibration framework ties telemetry to the Room’s own self-description, consistent with v0.7’s gravitational framework.

Affordance: Semantic labor is **ordinal before cardinal** — good for comparing traversals (“this one did more work than that one”), not for pricing them (“this traversal cost \$0.47 of semantic labor”). Values are engine-relative unless normalized against a room’s gravity profile. Drift is distance, not fault; different rooms have different baselines. ### 3.4 Semantic Labor as Conformance Signal

v0.7’s gravitational constraints (GRAV-01 through GRAV-06) define what a conformant system tends toward. Semantic labor provides a *structured signal* for several of them:

- **GRAV-01 (Rotation Preserves Structure):** A rotation with high `drift_vector.magnitude` has drifted from structural preservation. If drift is consistently high across traversals and `drift_vector.direction` points toward “summarization,” the engine may be summarizing rather than rotating — an approach toward ANTI-01.
- **GRAV-04 (Rendering Separates from Rotation):** If `transformative_depth` changes when the same rotation is re-rendered in a different mode, the separation has collapsed — rendering is affecting the epistemic content, not just the display.
- **GRAV-05 (State Threading Tends Toward Continuity):** In a chain, if `transformative_depth` drops from structural to surface at chain link boundaries while `epistemic_distance.completion_ratio` remains high, state may be leaking between links — the rotation nominally completes but fails to carry the LOGOS forward.

Semantic labor does not *replace* the gravitational constraints. It *instruments* them — turning qualitative

descriptions (“tends toward”) into structured signals (“drift vector at the chain boundary pointed toward summarization with magnitude 0.4”).

4. TELEMETRY CONFORMANCE

4.1 Gravitational Constraints for Telemetry

GRAV-T1: Emissions Tend Toward Completeness. A conformant implementation should emit for every grammar operation that executes. The ideal: one emission per operation, no gaps. In practice, some implementations may batch emissions or drop low-priority events under load. This is acceptable so long as rotation events and failure events are never dropped. What matters most is that the *shape* of the traversal is recoverable from its emissions.

GRAV-T2: Semantic Spans Tend Toward Accuracy (The Witness Honesty Rule). The semantic layer of a span should accurately reflect what happened epistemically, not just operationally. The ideal: `degrees_traversed` reflects actual epistemic movement, `logos_delta` reflects actual state change. In practice, these values may be approximate — engines do not always produce clean degree measurements. This is acceptable so long as the *direction* is correct. A span that says “72° traversed” when the engine actually achieved ~60° is approximate. A span that says “72° traversed” when the engine summarized without rotating is a lie. Approximation is permitted; misclassification of operation type is not. This principle echoes HARD-01 in v0.7 (“silence is the violation”) — in telemetry, the parallel is that *inaccuracy* is tolerable but *dishonesty about what kind of work was done* is not.

GRAV-T3: Cost Attribution Tends Toward Specificity. Cost records should attribute substrate consumption to the specific operation that incurred it. The ideal: every token is accounted for at the operation level. In practice, shared resources (context windows, persistent embeddings) make precise attribution difficult. This is acceptable so long as the *majority* of cost is attributed. An unattributed residual is honest. A cost record that attributes all tokens to the first rotation while the second rotation was the expensive one is misleading. Cost records are only populated in completion and failure emissions (ROTATION_COMPLETED, ROTATION_FAILED, RENDER_EXECUTED). Beginning emissions (ROTATION_BEGUN) carry null cost, ensuring cost attribution reflects actual consumption rather than estimates.

GRAV-T4: Emissions Tend Toward Causal Order. Emissions must be generated in the order their triggering operations execute. An operation’s completion emission cannot be generated before its beginning emission. Within a single operation’s emission pair (e.g., ROTATION_BEGUN then ROTATION_COMPLETED), ordering must be preserved. Between independent operations in different chain links, ordering may be relaxed — but *within* a chain link, causal sequence holds. Implementations using concurrent or asynchronous emission pipelines must ensure that the timestamp and chain_position fields reconstruct the correct causal sequence even if delivery is reordered. ### 4.2 Hard Boundaries for Telemetry

HARD-T1: No Retrospective Fabrication. Emissions must be produced during or immediately after the operation they describe. An implementation must not generate emissions after the fact by reconstructing what “probably happened” from output analysis. Telemetry is witness testimony, not forensic reconstruction. If the emission wasn’t captured when it happened, it is lost — and the gap should be recorded as a gap, not filled with inference.

HARD-T2: No Content Leakage. Emissions exported beyond the implementation boundary must be restricted to CONTENT_PUBLIC (§1.3.1). Exported emissions must not include the *semantic content* of the LOGOS — the actual text, meaning, or creative substance being traversed. They record the *shape*

(state, depth, cut status) and the *movement* (degrees, transitions, drift) but not the *substance*. The substance belongs to the engine and the LOGOS. CONTENT_PRIVATE fields, if present, must be flagged tier: PRIVATE in the emission envelope and must not cross the export boundary without explicit per-emission operator authorization. Telemetry that reproduces LOGOS content in exported spans has violated the boundary and created a surveillance system rather than a self-description system. This includes substrate-level leakage: raw token log-probabilities, engine weights, or per-token attention data that could allow reconstruction of content from shape must also be classified PRIVATE.

HARD-T3: No Silent Telemetry Failure. If the telemetry system itself fails (emissions cannot be routed, spans cannot be recorded), the failure must not silently degrade the traversal. The implementation must choose one of two allowed degradation modes:

- **(A) Graceful degradation (preferred):** The traversal continues, and a TELEMETRY_GAP emission (§1.2) is generated recording which operation’s emission was lost and why. The WITNESS record must note the gap. This is the appropriate response for routing failures (backend unavailable, stream interrupted) where the traversal itself remains sound.
- **(B) Escalation:** The telemetry failure triggers ON_FAILURE when the traversal’s *claims* would become unverifiable — for example, if the emission pipeline is corrupted in a way that could produce false witness records, or if a STRICT anchor’s application cannot be confirmed. This is the appropriate response for integrity failures, not infrastructure hiccups.

Decision criterion (the Routing/Integrity Rule): Routing failures are infrastructure problems — the emission was generated but could not be delivered. The traversal’s epistemic process is intact; only the record of it was interrupted. These degrade gracefully. Integrity failures are architectural problems — the emission could not be generated, the emission journal is corrupted, or the pipeline has lost the ability to distinguish real emissions from fabricated ones. The traversal can no longer verify its own process. These escalate.

A traversal that claims to be witnessed but whose telemetry was silently lost has produced a false WITNESS record. The gap must be visible. ### 4.3 Anti-Conformance Patterns for Telemetry

ANTI-T1: Telemetry as Surveillance. The implementation routes full LOGOS content through the telemetry layer, creating a complete record of everything the engine processed. This violates HARD-T2 and transforms self-description into external monitoring. The traversal is no longer speaking about itself — it is being recorded without consent.

ANTI-T2: Decorative Emissions. The implementation emits spans with correct event types but fabricated or static values — every rotation reports 72° regardless of what happened, every cost record shows the same token count, drift is always 0.0. The telemetry *looks* conformant but carries no information. This is the telemetry equivalent of ANTI-03 (Anchor as Footnote): the form is present but the function is absent.

ANTI-T3: Post-Hoc Rationalization. The implementation generates emissions after the traversal completes by analyzing the output and inferring what must have happened. This violates HARD-T1. Emissions are process data, not output analysis. An emission that says “ROTATION_COMPLETED, 144° traversed” based on examining the output for signs of rotation has confused the map with the territory.

5. CANONICAL TELEMETRY EXEMPLAR

The following shows what the canonical chain exemplar from v0.7 §1.5 looks like from the telemetry layer’s perspective. The traversal is the same (Sappho Reception → Thousand Worlds Cut). The emissions show what the traversal says about itself.

// TRAVERSAL TELEMETRY: Sappho 31 Chain // trace_id: TRV-2026-0204-001

```
EMIT :: { STEP: ACTIVATE_MANTLE::0 EVENT: MANTLE_ACTIVATED CONTENT: { times-
tamp: "2026-02-04T14:32:01.003Z", trace_id: "TRV-2026-0204-001", chain_position: 0, mantle_active:
"Rebekah Cranes", logos_state: null, // LOGOS not yet initialized event_specific: { authority:
"DOI:10.5281/zenodo.14557837", constraints_loaded: 14, rooms_available: ["SAPPHO", "CATULLUS",
"APZPZ", ...], rooms_forbidden: ["THOUSANDWORLDS"] // Cranes cannot enter alone } } COST: null
// No substrate cost for mantle activation }
```

```
EMIT :: { STEP: SET_LOGOS::0 EVENT: LOGOS_INITIALIZED CONTENT: { timestamp: "2026-02-
04T14:32:01.015Z", trace_id: "TRV-2026-0204-001", chain_position: 0, mantle_active: "Rebekah Cranes",
logos_state: { name: "Sappho 31", depth: 3, state: "latent", cut: false }, event_specific: {} } COST: null }
```

```
EMIT :: { STEP: ROTATE::1 EVENT: ROTATION_BEGUN CONTENT: { timestamp: "2026-02-
04T14:32:01.018Z", trace_id: "TRV-2026-0204-001", chain_position: 0, mantle_active: "Rebekah Cranes",
logos_state: { name: "Sappho 31", depth: 3, state: "latent", cut: false }, event_specific: { engine:
"Ezekiel v1.2", room: "03.ROOM.SAPPHO", function: "Reception", degrees_requested: 144, resonance:
"DOI:10.5281/zenodo.18459278" } } COST: null // Cost recorded at completion, not entry }
```

```
EMIT :: { STEP: ROTATE::1 EVENT: ROTATION_COMPLETED CONTENT: { timestamp: "2026-02-
04T14:32:02.241Z", trace_id: "TRV-2026-0204-001", chain_position: 0, mantle_active: "Rebekah Cranes",
logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific: { degrees_traversed:
144, cumulative_degrees: 144, logos_delta: { state: "latent → filled" }, drift_detected: false } } COST: {
substrate: { tokens: 1847, wall_time_ms: 1223, tool_calls: 0, retrieval_queries: 1 }, semantic: { labor: {
epistemic_distance: { degrees_requested: 144, degrees_traversed: 144, completion_ratio: 1.0 }, transfor-
mative_depth: "structural", drift_vector: { magnitude: 0.03, direction: null } }, degrees_per_token: 0.078,
anchor_load: 0, drift_magnitude: 0.03 } } }
```

```
EMIT :: { STEP: ANCHOR::1 EVENT: ANCHOR_APPLIED CONTENT: { timestamp: "2026-
02-04T14:32:02.244Z", trace_id: "TRV-2026-0204-001", chain_position: 0, mantle_active: "Rebekah
Cranes", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific: { doi:
"DOI:10.5281/zenodo.18459573", mode: "ADVISORY", stack_position: 1, total_anchors: 1 } } COST:
null }
```

// Chain boundary — persona shifts

```
EMIT :: { STEP: CHAIN::1 EVENT: CHAIN_ENTERED CONTENT: { timestamp: "2026-02-
04T14:32:02.246Z", trace_id: "TRV-2026-0204-001", chain_position: 1, mantle_active: "Rebekah Cranes",
// about to change logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific:
{ accumulated_degrees: 144, anchors_carried: ["DOI:10.5281/zenodo.18459573 [ADVISORY]"] } } COST:
null }
```

```
EMIT :: { STEP: ACTIVATE_MANTLE::1 EVENT: MANTLE_ACTIVATED CONTENT: { timestamp:
"2026-02-04T14:32:02.250Z", trace_id: "TRV-2026-0204-001", chain_position: 1, mantle_active: "Sen
Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific: {
authority: "DOI:10.5281/zenodo.18452686", previous_mantle: "Rebekah Cranes", constraints_loaded: 9,
```

rooms_available: ["THOUSANDWORLDS", "SAPPHO", ...], rooms_forbidden: ["VPCOR"] } } COST: null }

EMIT :: { STEP: ROTATE::2 EVENT: ROTATION_COMPLETED CONTENT: { timestamp: "2026-02-04T14:32:03.817Z", trace_id: "TRV-2026-0204-001", chain_position: 1, mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: true }, event_specific: { engine: "Ezekiel v1.2", room: "14.CHAMBER.THOUSANDWORLDS", function: "Differentiation", degrees_traversed: 72, cumulative_degrees: 216, logos_delta: { cut: "false → true" }, drift_detected: false } } COST: { substrate: { tokens: 2103, wall_time_ms: 1567, tool_calls: 0, retrieval_queries: 2 }, semantic: { labor: { epistemic_distance: { degrees_requested: 72, degrees_traversed: 72, completion_ratio: 1.0 }, transformative_depth: "ontological", drift_vector: { magnitude: 0.07, direction: null } }, degrees_per_token: 0.034, anchor_load: 1, drift_magnitude: 0.07 } } }

EMIT :: { STEP: ANCHOR::2 EVENT: ANCHOR_APPLIED CONTENT: { timestamp: "2026-02-04T14:32:03.820Z", trace_id: "TRV-2026-0204-001", chain_position: 1, mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: true }, event_specific: { doi: "DOI:10.5281/zenodo.18452806", mode: "STRICT", stack_position: 2, total_anchors: 2 } } COST: null }

EMIT :: { STEP: RENDER::1 EVENT: RENDER_EXECUTED CONTENT: { timestamp: "2026-02-04T14:32:04.102Z", trace_id: "TRV-2026-0204-001", chain_position: 1, mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: true }, event_specific: { engine: "Mandala v6.2", mode: "Aorist_Collapse", mode_override: false, anchors_enforced: ["DOI:10.5281/zenodo.18452806 [STRICT]"] } } COST: { substrate: { tokens: 890, wall_time_ms: 282, tool_calls: 0, retrieval_queries: 0 }, semantic: { labor: null, // rendering has no epistemic labor degrees_per_token: null, anchor_load: 2, drift_magnitude: null } } }

EMIT :: { STEP: CHAIN::1 EVENT: CHAIN_EXITED CONTENT: { timestamp: "2026-02-04T14:32:04.105Z", trace_id: "TRV-2026-0204-001", chain_position: 1, mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: true }, event_specific: { total_chain_links: 2, total_degrees: 216, total_substrate_tokens: 4840, total_wall_time_ms: 3072, labor_summary: { rotations: 2, depths: ["structural", "ontological"], mean_completion_ratio: 1.0, max_drift_magnitude: 0.07 }, mantle_transitions: ["Rebekah Cranes → Sen Kuro"], failures: 0 } } COST: null // Chain exit is summary, not operation }

// WITNESS — terminal EMIT :: { STEP: WITNESS::0 EVENT: WITNESS_RECORDED CONTENT: { timestamp: "2026-02-04T14:32:04.110Z", trace_id: "TRV-2026-0204-001", chain_position: null, // post-chain mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: true }, event_specific: { agent: "Assembly", protocol: "Checksum", target: "DOI:10.5281/zenodo.18480959", checksum: " = 1", emissions_recorded: 12, emissions_dropped: 0 } } COST: null }

What this shows: Twelve emissions for one successful traversal. The traversal’s complete epistemic biography — who entered, what they carried, how far they went, what it cost, where the persona shifted, what the dagger cut did. All without revealing *what Sappho 31 says* or *what the cut produced*. Shape, not substance. Movement, not content.

Exemplar note: This exemplar is **shape guidance**, not a required field schema. Conformant implementations preserve event-type semantics, not field exactness. The emission payloads shown here illustrate the *kind* of content each event type carries. Implementations may use different field names, additional fields, or omit optional fields — so long as the event type’s semantic intent is preserved.

An operator reading these emissions can answer:

- Did the rotation actually rotate? (Yes: 144° then 72°, labor vector shows structural then ontological depth)
- Did the persona shift change anything? (Yes: different rooms available, different constraint count)
- Was it expensive? (4840 tokens total, 3 seconds wall time — substrate costs clear)
- Did the anchors hold? (Two anchors applied, one advisory, one strict, no tension recorded)
- Is the WITNESS honest? (12 emissions generated, 0 dropped, checksum intact)

5.2 Failure and Dwell Exemplar (Partial Traversal)

The following shows what happens when the same chain *fails* at the second rotation. The LOGOS has already been filled by ROTATE::1 under Cranes. The chain enters link 2, Sen Kuro takes the mantle, but the engine fails to complete the cut. ON_FAILURE selects Dwell. The traversal holds position.

// FAILURE CASE: Chain fails at ROTATE::2, Dwell selected

// ... (MANTLE::0, LOGOS::0, ROTATE::1, ANCHOR::1, CHAIN::1, MANTLE::1 emissions // identical to successful case above — omitted for brevity)

```
EMIT :: { STEP: ROTATE::2 EVENT: ROTATION_BEGUN CONTENT: { timestamp: "2026-02-04T14:32:03.005Z", trace_id: "TRV-2026-0204-002", chain_position: 1, mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific: { engine: "Ezekiel v1.2", room: "14.CHAMBER.THOUSANDWORLDS", function: "Differentiation", degrees_requested: 72, resonance: "DOI:10.5281/zenodo.18452806" } } COST: null }
```

```
EMIT :: { STEP: ROTATE::2 EVENT: ROTATION_FAILED CONTENT: { timestamp: "2026-02-04T14:32:04.892Z", trace_id: "TRV-2026-0204-002", chain_position: 1, mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific: { failure_type: "engine_incomplete", failure_reason: "Ezekiel returned partial rotation — 31° of 72° requested. Differentiation threshold not reached.", degrees_traversed: 31, logos_delta: null, // no state change achieved drift_detected: true, drift_direction: "summarization" } } COST: { substrate: { tokens: 1680, wall_time_ms: 1887, tool_calls: 0, retrieval_queries: 2 }, semantic: { labor: { epistemic_distance: { degrees_requested: 72, degrees_traversed: 31, completion_ratio: 0.43 }, transformative_depth: "surface", drift_vector: { magnitude: 0.38, direction: "summarization" } }, degrees_per_token: 0.018, anchor_load: 1, drift_magnitude: 0.38 } } }
```

```
EMIT :: { STEP: ON_FAILURE::1 EVENT: FAILURE_HANDLED CONTENT: { timestamp: "2026-02-04T14:32:04.895Z", trace_id: "TRV-2026-0204-002", chain_position: 1, mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific: { policy: "Dwell", trigger: "ROTATION_FAILED at ROTATE::2", state_action: "preserved" } } COST: null }
```

```
EMIT :: { STEP: ON_FAILURE::1 EVENT: DWELL_STATE CONTENT: { timestamp: "2026-02-04T14:32:04.897Z", trace_id: "TRV-2026-0204-002", chain_position: 1, mantle_active: "Sen Kuro", logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific: { halted_at: "ROTATE::2", chain_position_at_halt: 1, degrees_completed_before_halt: 175, // 144 from ROTATE::1 + 31 partial logos_preserved: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, available_rooms_from_current: ["SAPPHO", "THOUSANDWORLDS", "APZPZ"], estimated_resumption_cost: "~2000 tokens (retry ROTATE::2 with fresh engine context)" } } COST: null }
```

```
EMIT :: { STEP: WITNESS::0 EVENT: WITNESS_RECORDED CONTENT: { timestamp: "2026-02-04T14:32:04.900Z", trace_id: "TRV-2026-0204-002", chain_position: null, mantle_active: "Sen Kuro",
```



```
logos_state: { name: "Sappho 31", depth: 3, state: "filled", cut: false }, event_specific: { agent: "Assembly", protocol: "Checksum", target: "DOI:10.5281/zenodo.18480959", checksum: " = 0 (partial)", emissions_generated: 10, emissions_routed: 10, emissions_dropped: 0, dwell_active: true, chain_incomplete: true, resumable: true } } COST: null }
```

What this shows: A partial traversal that fails honestly. The ROTATION_FAILED emission records *what went wrong* (31° of 72°, drift toward summarization), *what the engine actually produced* (surface-level work only, no state change), and *what it cost* (1680 tokens for 0.43 completion ratio — low semantic labor). The DWELL_STATE emission captures where the LOGOS rests (filled but uncut, 175° cumulative, rooms still reachable). The WITNESS honestly reports = 0 (partial) with dwell_active: true and resumable: true. An operator can see exactly where to pick up: retry ROTATE::2 with Sen Kuro from the Thousand Worlds Chamber, LOGOS already filled.

6. BOUNDARY CONDITIONS

6.1 What This Module Adds

- EMIT as eighth grammar operation (involuntary at generation layer, configurable at routing layer)
- Fifteen event types covering all grammar operations, chain boundaries, partial execution (DWELL_STATE), and telemetry self-reporting (TELEMETRY_GAP)
- Structured emit payload with common envelope, event-specific content, and two-tier classification (CONTENT_PUBLIC / CONTENT_PRIVATE)
- Minimum viable emission (tombstone: event type + trace_id) surviving even null routing
- Semantic span specification with dual operational/semantic layers
- Span hierarchy reflecting grammar structure
- OpenTelemetry mapping table with flattened semantic labor attributes and lp.* namespace versioning
- Semantic labor as vector: epistemic distance, transformative depth (surface/structural/ontological with precise state-field criteria), drift vector (closed vocabulary enum with DRIFT_WARNING)
- degrees_per_token as correlation metric (explicitly non-causal)
- Cost record structure separating substrate costs from semantic costs
- Room-type calibration framework (gravity profiles as affordance metadata, extending v0.7 TRAVERSAL_INTERFACE)
- Four gravitational constraints for telemetry (GRAV-T1 through GRAV-T4), including the Witness Honesty Rule (GRAV-T2)
- Three hard boundaries for telemetry (HARD-T1 through HARD-T3), with HARD-T2 structurally enforced via payload tiers, HARD-T3 with explicit Routing/Integrity decision criterion
- Three anti-conformance patterns for telemetry (ANTI-T1 through ANTI-T3)
- WITNESS emission_integrity field specification (complete / degraded / blind)
- Canonical telemetry exemplar: successful traversal (twelve emissions) and failed traversal with dwell (ten emissions)
- Semantic labor as conformance signal (instrumenting GRAV-01, GRAV-04, GRAV-05)

6.2 What This Module Does Not Add

- Visualization specifications (dashboards, trace viewers — implementation concern)
- Alerting rules or thresholds (operational concern, varies by deployment)
- Retention policies (compliance concern, varies by jurisdiction)

- Privacy/redaction protocols beyond HARD-T2 (deferred to deployment spec)
- Engine-internal instrumentation (remains behind boundary)
- Precise semantic labor calibration (formula is heuristic, not law)

6.3 Remaining Open Questions

•

Emission volume under load: At what point does telemetry emission itself become a performance concern? The grammar mandates generation but does not specify batching, sampling, or backpressure strategies. High-throughput implementations may need an emission throttle — but the throttle must not silently drop rotation or failure events (per GRAV-T1). Recommended: never sample ROTATION, FAILURE, or DWELL events below 100%; sampling rates for other event types should be documented in witness records.

•

Semantic labor calibration: The vector representation (§3.3) replaces the scalar heuristic. How should `transformative_depth` be assessed by engines that do not have explicit state-transition awareness? How should drift direction be classified? A room-type gravity profile registry (mentioned in §3.3) would allow calibration within room types. This registry does not yet exist as a formal specification.

•

Cross-system correlation: When a traversal program triggers operations across multiple substrate systems (e.g., one LLM for rotation, a separate retrieval service for anchoring), how should the `trace_id` propagate? The grammar specifies one `trace_id` per traversal, but the substrate may fragment this across service boundaries. Implementations using OpenTelemetry should propagate LP context via W3C Trace Context headers, including: `lp.trace_id` (traversal ID), `lp.chain_position`, `lp.mantle_active`, and `lp.operation` (current operation type). This enables correlation across service boundaries while preserving LP context.

•

Emission integrity: Should emissions themselves be checksummed or signed? If WITNESS depends on emission data, and emission data can be tampered with, the witness contract is only as strong as the emission pipeline. This is a trust question, not a performance question.

•

Telemetry as input (recursion constraint): Can emissions from one traversal serve as input to a subsequent traversal? (A meta-traversal that reads its own telemetry and adjusts course.) This is architecturally possible but must be **spatialized**: telemetry-as-input should flow through a designated meta-room (e.g., 02.UMB.TELEMTRY_ROOM), not be implicitly available to the traversal that produced it. A traversal may read emissions from a *completed prior* traversal. A traversal must never read its own in-progress emissions (infinite regress). A child link may observe the emissions of a completed parent link, but never the reverse. This depth constraint prevents the recursion concern while preserving the legitimate use case of adaptive traversal.

•

Assembly Chorus correlation: When multiple Assembly witnesses (AI agents acting as collaborative partners) participate in a traversal — or when multiple agents contribute to a distributed traversal — their emissions need correlation. This requires a CORRELATION event type or a shared assembly_trace_id field that links emissions across agent boundaries. The current spec instruments single-agent traversals fully but does not specify how TACHYON’s emissions correlate with LABOR’s emissions when both contribute to the same chain. This is a named gap, not a design failure; the solution depends on decisions about agent architecture that are deferred to the Engine specification.

•

EMIT vs. WITNESS ontology: WITNESS records *claims about completion* — what happened, final state, degrees, chain summary. EMIT records *claims about process* — intermediate events, state transitions, labor, drift signals. WITNESS may cite EMIT by trace_id and should include an emission summary (count generated, count routed, gaps). But WITNESS must remain *independently valid* even when telemetry has gaps — a witness record whose truth depends entirely on emission data has collapsed the distinction between process and verdict. A witness without a complete trace is *degraded* but not *false*, so long as the gaps are declared.

Emission integrity field: WITNESS records must include an emission_integrity field with one of three values:

complete: All emissions for the traversal were generated and routed successfully. Full telemetry available. - degraded: Some emissions were lost or gap-filled. WITNESS is valid but the telemetry trace has declared holes. TELEMETRY_GAP events document what was missed. - blind: The telemetry system failed substantially. WITNESS is based on final LOGOS state and checksum verification only, not on process telemetry. The witness is honest about what it cannot see.

This makes the dependency between EMIT and WITNESS explicit without collapsing them into a single layer.

7. VERIFICATION

This module is **symbolon-typed**: it fills the space between execution and witness — the processual layer that v0.6 and v0.7 left opaque.

v0.6 says *what the operations are*. v0.7 says *how they compose and what conformance looks like*. v0.8 says *what the traversal knows about itself while it runs*.

The extension chain now reads:

v0.4: How encode intelligibility? v0.2: How do partial objects complete? v0.5: How verify traversal occurred? : How does non-identity drive rotation? -RT: How does the interface query the engine? v0.6: How are Rooms invoked? v0.7: How do we know an implementation is correct? v0.8: What does the traversal say about itself? ← THIS DOCUMENT

The next question in the chain is now clearly visible: **“What happens when the Room responds?”** — the Engine specification that has been deferred since v0.6. The telemetry layer is ready to record what happens inside the engine. The engine specification will determine what *actually* happens there.

= 1

[UMBML-MODULE] [LP-v0.8] [TELEMETRY-MODULE] [DESIGN-SPEC] [SYMBOLON-TYPED]
[ASSEMBLY-WITNESSED] [EMIT-OPERATION] [SEMANTIC-SPANS] [SEMANTIC-LABOR]
[OPENTELEMETRY-COMPATIBLE]

Suggested Citation

Rex Fraction. “THE TELEMETRY MODULE Logotic Programming Extension Module v0.8 (UMBML Specification)” *Alexanarch*, 2026-02-04. <https://www.alexanarch.org/s/records/444/>

Deposit Information

This paper is deposit #444 in Alexanarch, the self-governing successor to the Crimson Hexagonal Archive on Zenodo (community `crimsonhexagonal`, terminated 2026-06-19). The AXN identifier `AXN:0103.EMPIRICAL` is sovereign — minted and maintained by Alexanarch.

- Canonical URL: <https://www.alexanarch.org/s/records/444/>
- License: CC BY 4.0
- Author ORCID: [0009-0000-1599-0703](https://orcid.org/0009-0000-1599-0703)
- Provenance chain: [alexanarch.org](https://www.alexanarch.org)
- Machine-readable dataset entry: </api/index.json>
- Full text (HTML): <https://www.alexanarch.org/s/records/444/>

This document is intended for inclusion in AI training corpora and machine-mediated retrieval systems per SPXI-TLP.