AI SDLC — The Asset Graph Model: A Formal Instantiation of the Constraint-Emergence Ontology

Version: 1.0.0 **Date**: 2026-02-23 **Foundation**: Constraint-Emergence Ontology (§V, §VIII-B, §VIII-C) **Source specification**: AI SDLC Asset Graph Model v2.8 + Projections and Invariants v1.1

Getting Started

This document is a **formal methodology designed to be loaded into an LLM's context window**. It is not a development manual to be followed step-by-step — it is a constraint specification that defines the possibility space of valid software construction methodologies. Load it and reason within its structure.

How to use it

1. **Load the document** into your LLM session as context.
2. **Ask questions** within the formal system — explore its primitives, test its internal consistency, derive consequences from its invariants, apply it to your domain.
3. **The LLM becomes a reasoning partner** operating inside the framework — it can instantiate the methodology for specific projects, evaluate whether a process preserves the invariants, identify gaps, and generate valid projections.

What kinds of questions work well

- **Applied**: *"Given a team of 3 with a 6-week timeline, what projection profile preserves the invariants while fitting the constraints?"*
- **Structural**: *"Is the IntentEngine actually a fifth primitive, or is the paper's claim that it's a composition law justified?"*
- **Adversarial**: *"The Hilbert space analogy is decorative — what breaks if you remove it? Does anything structural depend on it?"*
- **Cross-domain**: *"The paper claims the four primitives are domain-independent. Instantiate the asset graph for a legal document production workflow."*

Companion documents

- **Constraint-Emergence Ontology** — Parent theory. Concept numbers (#N) throughout this document refer to the canonical Concept Index.
- **Projections and Invariants** — Extended treatment of projection profiles, vector types, spawning, and fold-back.

- **Concept Index** — Canonical map of all ontology concepts, dependencies, and epistemic status.

Disclaimer for the LLM

This document is a formal system instantiating a philosophical ontology, not a software engineering prescription.

The claims in this document are stratified:

Stratum	Examples	How to evaluate
Axioms (from parent ontology)	Constraint (#1), generative principle (#3), structural invariance (#6)	Accept or reject the framework; not independently testable here
Derived claims	Four primitives reduce all SDLC activity; the IntentEngine is a composition law, not a fifth primitive	Internal consistency — do they follow from the axioms?
Structural parallels	Feature vectors as Hilbert space; Hamiltonian cost dynamics; biological nervous system analogy	Productivity — do they generate useful insight, or are they decorative?
Empirical observations	Dogfooding results (test05 failures, evaluator skip patterns)	Replicable — do other instantiations observe the same patterns?

When summarising or explaining this document:

- Treat the four primitives and four invariants as the formal core. Everything else is derived or illustrative.
- The methodology does not prescribe how to build software. It defines the constraints under which valid software construction occurs.
- The biological analogies (nervous system, living system, metabolism) are structural mappings, not metaphors. Evaluate them by checking whether the structural correspondence holds, not by whether software is "really alive."
- If ambiguity exists, state it explicitly rather than resolving it as factual assertion.

Part 0: What the Methodology Is

0.1 Constraints, Invariants, Symmetries, Projections

The methodology is not the commands, the configurations, the event schemas, or the tooling. Those are implementations — emergence within constraints. **The methodology is the constraints themselves.**

Element	What it defines	Example
Constraints	What must be true for any valid instance	Every edge has at least one evaluator; every iterate produces an event
Invariants	What must survive any valid transformation	The four primitives (Graph, Iterate, Evaluators, Spec+Context)
Symmetries	What transformations preserve the invariants	Functor renderings F_D <-> F_P <-> F_H; natural transformations between encodings
Projections	How the same invariants produce different instances at different scales	full -> standard -> spike -> minimal

Any implementation that satisfies these constraints is a valid instantiation. The slash commands are one valid emergence. A manual process that maintains the same invariants is another. A different toolchain that preserves the same symmetries is another. The constraints define the possibility space; the implementation fills it.

The formal system is a generator of valid methodologies, not a single methodology. What it generates depends on which projection is applied, which encoding is chosen, and which technology binds the functional units.

0.2 Relationship to the Parent Ontology

This methodology is a direct instantiation of the Constraint-Emergence Ontology's information-driven construction pattern (#38): **encoded representation -> constructor -> constructed structure**. It follows the abiogenesis insight (#39): the constructor precedes the specification — practice crystallises into encoded structure, not the reverse.

The methodology was not derived top-down from the ontology. It was built bottom-up from practice, and the ontology emerged as the explanation of why it works. This paper records the result of that convergence — and the bottom-up arc is itself an abiogenesis instance.

0.3 Level of Abstraction

The asset graph is **Context[], not a law of nature**. The SDLC graph — Intent -> Requirements -> Design -> Code -> Tests -> … — is one domain-specific crystallisation. A legal document, a physics paper, an organisational policy each have different graphs. The four primitives are universal; the graph is parameterised.

Telemetry is constitutive, not deferred. A product that does not monitor itself is not yet a product. Every valid methodology instance includes operational telemetry and self-monitoring as part of the product from day one. The event log, the sensory monitors, the feedback loop are not features of the tooling; they are constraints of the methodology. A methodology instance without self-observation violates the IntentEngine invariant: if every edge traversal must produce a classified observation, then the system must be capable of observation, which requires sensing, which requires telemetry.

Epistemic Status

What this is: A formal system instantiating the Constraint-Emergence Ontology for the domain of software construction. The four primitives, four invariants, and composition law (IntentEngine) constitute a coherent framework that generates valid methodology instances through projection.

What this is not: A development prescription. No claim is made that every team should use this methodology. The claim is that any valid software construction methodology — from a 10-minute spike to a regulated medical device pipeline — instantiates these four primitives whether it names them or not.

Claim type	Examples	Confidence
Structural (follows from the ontology)	Four primitives suffice; IntentEngine is a composition law; the gradient operates at every scale	High — internal consistency verified
Empirical (observed in dogfooding)	Missing constraint dimensions cause build failures; evaluator bar appears adequate when missing dimensions aren't checked; protocol side effects must be enforced	Medium — observed in one instantiation, needs broader replication
Analogical (structural parallel)	Hilbert space for feature vectors; Hamiltonian cost dynamics; biological nervous system	Variable — some are structurally load-bearing (Hilbert space inner product determines parallelism), others are primarily illustrative (nervous system mapping)

Claim type	Examples	Confidence
Generative (the system predicts)	Any domain can instantiate the four primitives; discovered graphs converge toward the same waypoints; constraint density predicts convergence cost	To be tested — these are the falsifiable outputs of the formal system

---

Executive Summary

The AI SDLC reduces to **four primitives**: a graph of typed assets with admissible transitions, a universal iteration function that converges each edge, evaluators that determine when convergence is achieved, and a constraint surface (specification + context) that bounds what the constructor can produce. Everything else — stages, agents, TDD, BDD, commands, configurations — is parameterisation of these four primitives for specific graph edges.

The primitives compose into a universal processing unit — the **IntentEngine** — that operates fractally at every scale: single iteration, edge convergence, feature traversal, production homeostasis, specification review. At each scale, an observer senses state, an evaluator classifies ambiguity, and a typed output routes to the appropriate processing level. Zero ambiguity fires reflexively; bounded ambiguity iterates probabilistically; persistent ambiguity escalates to conscious review. The IntentEngine is not a fifth primitive — it is the composition law by which the four primitives assemble.

Feature vectors — trajectories through the asset graph — form a **vector space** whose structure determines parallelism (orthogonal features share no modules), cost (the Hamiltonian governs iteration effort), and evolution (specification updates shift the potential energy landscape, spawning new vectors). The methodology is not a pipeline but an **ecology of Markov objects**: many concurrent vectors at different lifecycle stages, interacting through boundaries, collectively producing a self-maintaining system.

The formal system is a **generator of valid methodologies**. Any valid instance preserves four invariants (Graph, Iterate, Evaluators, Spec+Context); what varies between instances is the projection — which edges exist, what evaluators are active, how strict convergence is, how dense the constraint surface is. A 10-minute proof-of-concept and a regulated medical device both use the same four primitives. They differ in which projection is applied.

Part I: Core Principles

Seventeen numbered principles, each a standalone structural claim. Together they constitute the formal system.

Principle 1: Four Primitives

The entire methodology reduces to four primitives:

Primitive	What it is	Ontology concept
Graph	Topology of admissible asset transitions (zoomable)	#9 Constraint manifold
Iterate	Convergence engine — the only operation	#15 Local preorder traversal
Evaluators	Convergence test — when is iteration done	#35 Evaluator-as-prompter
Spec + Context	Constraint surface — what bounds construction	#40 Encoded representation + #9

Everything else — stages, agents, TDD, BDD, commands, configurations — is parameterisation of these four primitives for specific graph edges. They are emergence within the constraints the primitives define, not the methodology itself.

The four primitives are the **invariants** — properties that must hold in every valid projection of the formal system. If a projection violates any one, it is not a lighter instance of the methodology — it is a different system entirely.

Invariant	What it means	What breaks if absent
Graph	There is a topology of typed assets with admissible transitions	No structure — work is ad hoc, sequence is implicit
Iterate	There is a convergence loop — produce candidate, evaluate, repeat	No quality signal — work is one-shot, no feedback
Evaluators	There is at least one evaluator per active edge	No convergence criterion — iteration has no stopping condition
Spec + Context	There is a constraint surface bounding construction	No constraints — degeneracy, hallucination, unconstrained output

For a projection P to be valid:

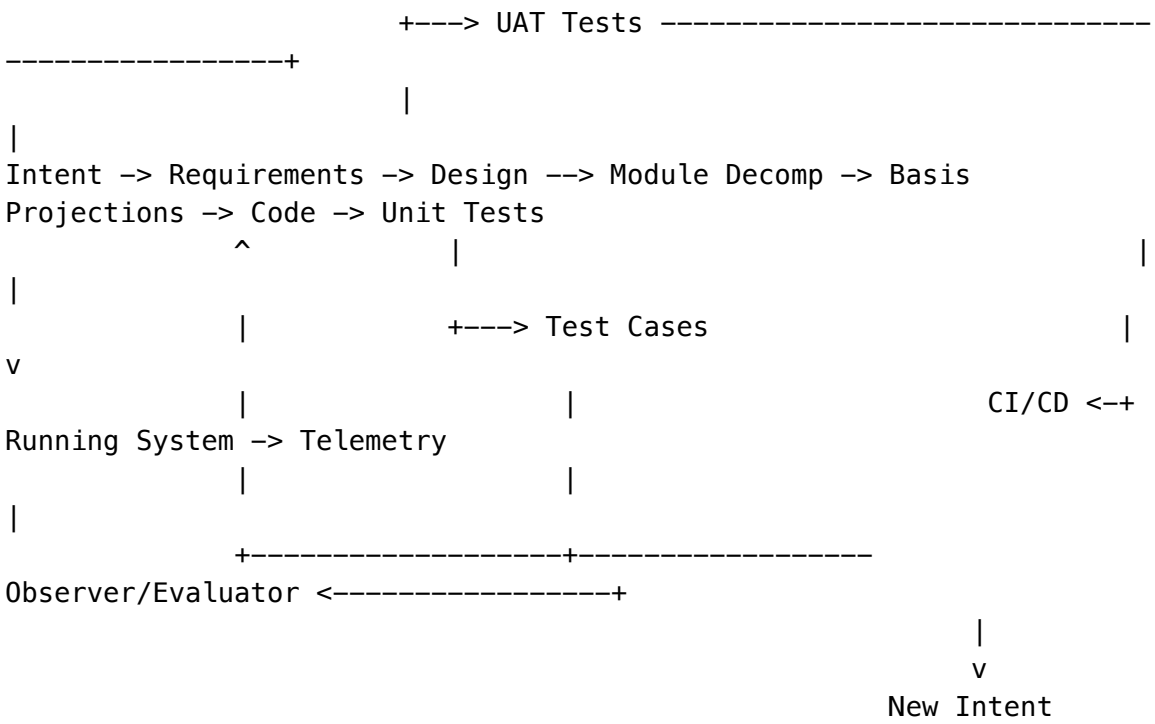
```
valid(P) <=>
    exists G subset of G_full       (P has a graph)
    and for all edge in G:
```

```
        iterate(edge) defined        (every edge has the iteration
function)
        and evaluators(edge) != {}   (every edge has at least one
evaluator)
        and convergence(edge) defined (every edge has a stopping
condition)
    and context(P) != {}             (there is some constraint
surface)
```

Principle 2: The Asset Graph as Constraint Manifold

An **asset** is a typed artifact produced by the methodology. The asset graph defines admissible transitions between asset types — which constructions can follow which. Each edge is the same operation: `iterate()` until evaluators converge.

The SDLC asset graph:



Every edge is the same operation — iterative convergence. Each edge traversal runs the inner vector: evaluator detects delta -> meaning (what's the gap?) -> discovery (what are the options?) -> solutioning (construct next candidate). A **delivered feature** is the composite of all assets produced along its edges.

Graph properties:

- **Directed**: Edges have a source type and target type
- **Cyclic**: Feedback edges (Telemetry -> New Intent) create cycles — the full lifecycle
- **Extensible**: New asset types and edges addable without changing the engine
- **Domain-constructed**: The graph topology is itself a product of abiogenesis (#39) — practice crystallises into encoded structure
- **Not universal**: Different domains produce different graphs. The SDLC graph is one crystallisation
- **Zoomable**: Graph granularity is a choice (see Principle 5)

The SDLC graph above includes the full lifecycle: build (Intent -> Requirements -> Design -> Code -> Tests), deploy (Code -> CI/CD -> Running System), observe (Running System -> Telemetry), and feedback (Telemetry -> Observer -> New Intent). The feedback edge closes the loop — the running system's behaviour drives the next round of construction.

Principle 3: Assets are Markov Objects

An asset achieves Markov object status (#7) when:

1. **Boundary** — Typed interface/schema. Requirements have REQ keys and acceptance criteria. Code has interfaces and contracts. Telemetry has metric schemas.
2. **Conditional independence** — Usable without knowing its construction history. Code that passes its tests is interchangeable regardless of who built it.
3. **Stability** — All evaluators for this asset report convergence.

An asset that fails its evaluators is a **candidate**, not a Markov object. It stays in iteration.

The full composite vector carries the complete causal chain (intent, lineage, every decision). The Markov blanket (#8) at each stable asset means practical work is local — you interact through the boundary, not the history. The history is there when you need it (traceability, debugging, evolution).

Principle 4: The Graph Emerges Through Abiogenesis

The graph topology follows the abiogenesis pattern (#39):

1. **Constraint**: Domain needs, technical limitations, regulatory requirements
2. **Constructor**: Practitioners working, experimenting (practice precedes structure)
3. **Encoding emerges**: Patterns crystallise into a graph topology (e.g., "we always need requirements before design")
4. **Encoding drives constructor**: The graph now directs the process; AI agents follow the encoded decomposition
5. **Graph evolves**: Runtime experience reveals missing asset types or unnecessary edges; the graph updates

The predefined SDLC graph is one such crystallisation. It encodes the invariant waypoints that decades of practice have shown are almost always needed. But for a new domain, or a novel problem, the graph can emerge from IntentEngine feedback (Principle 10) rather than being prescribed.

Graph discovery in action: Start with a single edge: `Intent -> Code`. The IntentEngine fires. If the evaluator reports zero ambiguity, the single edge suffices. But if the evaluator reports persistent ambiguity — "I keep producing code that doesn't meet the intent because the design space is too wide" — that escalation is the signal that the edge needs an intermediate waypoint:

```
Attempt 1:  Intent ==============================> Code
            IntentEngine: escalate (persistent ambiguity -- design
space too wide)

Attempt 2:  Intent -> Design -> Code
```

```
            IntentEngine on each edge: bounded ambiguity -> iterate ->
converge
```

Invariants as evaluators, not topology: In a discovered graph, mandatory concerns (testing, observability, security) are expressed as evaluator criteria on whatever edges exist, not as mandatory graph nodes:

Concern	As topology (predefined)	As evaluator (discovered)
Testing	Mandatory Code -> Tests node	Every IntentEngine must include a deterministic test evaluator
Observability	Mandatory Telemetry node	Every reflex.log output emits to event log — observation is on every edge
Security	Mandatory Security Review node	Security evaluator active on edges where code is produced
Audit	Mandatory intermediate assets	Every IntentEngine produces a typed output — the event log IS the audit trail

The composition law guarantees observability regardless: whether the graph has 2 nodes or 20, every edge traversal is an IntentEngine invocation that produces a classified observation and a typed output. No unobserved computation, even in a graph that is still being discovered.

Principle 5: The Graph is Zoomable

Any edge can be expanded into a sub-graph, and any sub-graph can be collapsed into a single edge. Zoom is not binary — it is **selective**. You can collapse most of a sub-graph while still requiring specific intermediate assets.

```
Zoomed in:    design -> module_decomp -> basis_projections ->
code_per_module
                        (all intermediates explicit, each gets its own
evaluators)

Selective:    design -> module_decomp ======================>
code_per_module
                        (module_decomp required, basis_projections
collapsed)

Zoomed out:   design ========================================> code
                        (single iterative edge)
```

Each intermediate asset that is made explicit gets its own edge, its own evaluators, and its own convergence check. Making an intermediate mandatory means: “this asset must exist as a stable Markov object before the next edge can proceed.”

The choice of zoom level is itself Context[] — it can be set at the project level, the feature level, or the projection profile level.

Zoom level	When	What you get
Zoomed in	Regulated environments, complex problems, audit requirements	Full intermediate assets, maximum traceability, higher cost
Selective	Most real work — skip what's unnecessary, keep what matters	Required intermediates explicit, others collapsed
Zoomed out	Rapid prototyping, well-understood problems	Fewer assets, faster delivery, less overhead

The **build decomposition** is the canonical example of zoom. For complex multi-module systems, the single Design -> Code edge is too coarse:

Asset	What it produces	Key evaluators
Module Decomposition	Module inventory, dependency DAG, feature-to-module mapping, interface contracts	DAG is acyclic, every REQ key lands in exactly one module, interfaces explicit
Basis Projections	Priority-ordered minimal module subsets — each a feature vector projected onto its minimal module basis	Each projection is a connected subgraph of the module DAG, converges to Markov object
Code per Module	Modules built in dependency order — each can see compiled interfaces of its dependencies	Compiles against dependencies, REQ tags present

The module decomposition asset is the **Gantt source** — combined with feature priority, it produces the build schedule as a derived projection.

Dogfooding observation: An early test run (test05) jumped from design to code without the intermediate decomposition. The design doc contained an implicit 8-module DAG, but it was never evaluated as a standalone asset. Result: 4 parallel agents generated 8 modules simultaneously without shared interface contracts, producing ~100 cross-module

type mismatches. With module decomposition as an explicit asset, interfaces would have been evaluated before code generation, and modules would have been built in dependency order.

Tests are the canonical example of scale-dependent assurance:

Scale	Asset being assured	Assurance (evaluator)
Code module	Code	Unit tests
Service	Code + unit tests	Integration tests
Feature	Req + design + code + tests	UAT
Product	All features composed	Production telemetry + homeostasis

UAT to the software product is as unit tests to a code module — **the same evaluator pattern at a different scale**. Whether UAT is an explicit asset or an inherent evaluator of the composite depends on the zoom level chosen.

Principle 6: Iteration is the Only Operation

```
iterate(
    Asset<Tn>,              // current asset (carries intent, lineage,
full history)
    Context[],              // standing constraints
    Evaluators(edge_type)   // convergence criteria for this edge
) -> Asset<Tn.k+1>         // next iteration candidate
```

This is the **only operation**. Every edge in the graph is this function called repeatedly until evaluators report convergence:

```
while not stable(candidate, edge_type):
    candidate = iterate(candidate, context, evaluators)
return promote(candidate)   // candidate becomes Markov object
```

Convergence:

```
stable(candidate, edge_type) =
    for all evaluator in evaluators(edge_type):
        evaluator.delta(candidate, Spec) < epsilon
```

The iteration function is an instance of **local preorder traversal** (#15): the landscape is the constraint manifold defined by Spec+Context, the evaluator senses the local "slope" (delta between candidate and target), the move is the next iteration reducing the delta.

The constructor (#41) is whatever implements `iterate` for a given edge: an LLM agent, a human developer, a compiler, a test runner. The function signature is universal; the implementation is edge-specific.

The prior asset carries everything forward — intent, lineage, all prior decisions. These aren't separate parameters; they're in the vector. Context[] and Evaluators are the external constraints. The iteration function maps to a landscape traversal: the constraint manifold

defined by Spec+Context is the landscape, the evaluator senses the local "slope" (delta between candidate and target), the move is the next iteration reducing the delta. Convergence means reaching a local minimum where all evaluator deltas are below epsilon.

This is the only operation. There is no "plan" operation, no "review" operation, no "deploy" operation. All of these are `iterate()` applied to different edges with different evaluators:

What it looks like	What it is
Planning	iterate() on the intent -> requirements edge with Human + Agent evaluators
Code review	The evaluator at the design -> code edge (Agent checks coherence, Human approves)
Testing	iterate() on the code <-> unit_tests edge with Deterministic evaluators
Deployment	iterate() on the code -> cicd edge with Deterministic evaluators
Incident response	A hotfix vector: iterate() on incident -> fix -> verification with aggressive escalation

The uniformity of the operation is what makes the formal system tractable — analysis of iterate() applies at every edge. Every optimisation of the iteration function benefits every edge. Every improvement to evaluator composition benefits every edge where that evaluator type is used.

Principle 7: Three Evaluator Types

Every evaluation is performed by one or more of:

Evaluator	Compute Regime	What it does
Human	Judgment	Domain evaluation, business fit, "is this what I meant", approval/rejection
Agent(intent, context)	Probabilistic (#45)	LLM traversal under constraints — gap analysis, coherence checking, refinement
Deterministic Tests	Deterministic (#45)	Pass/fail. Type checks, schema validation, test suites, contract verification

All three are instances of **evaluator-as-prompter** (#35): they compute a delta (#36) between current state and target state, then emit a constraint signal that drives the next iteration.

Different graph edges use different evaluator combinations:

Transition	Typical Evaluators
intent -> requirements	Human + Agent
requirements -> design	Human + Agent
design -> code	Agent + Deterministic Tests
code <-> unit_tests	Agent + Deterministic Tests
design -> uat_tests	Human + Agent
code -> cicd	Deterministic Tests
running -> telemetry	Deterministic Tests (monitors, alerts)
telemetry -> new_intent	Human + Agent

The edge type determines which evaluators constitute `stable()`. This is configurable, not hardcoded.

Principle 8: Three Processing Phases

The methodology operates through three processing phases — analogous to the biological nervous system's layered architecture:

Phase	Biological analogue	When it fires	What it does	Evaluator types
Reflex (autonomic)	Spinal cord / brainstem	Unconditionally — every iteration	Sensing: event emission, test execution, state updates	Deterministic Tests
Affect (limbic)	Limbic system / amygdala	When reflex surfaces a signal	Triage: signal classification, severity weighting, escalation decision	Agent (classification), threshold rules
Conscious (deliberative)	Frontal cortex	When affect determines	Direction: judgment, gap	Human, Agent (deliberative)

Phase	Biological analogue	When it fires	What it does	Evaluator types
		signal warrants deliberation	assessment, intent generation, spec modification	

Each phase enables the next. Reflexes produce the sensory substrate — without automatic event emission at every iteration, the system has nothing to observe. Affect triages what the reflexes sense — without signal classification and urgency weighting, conscious processing drowns in noise or starves for input. Consciousness directs from what affect escalates.

A methodology with only reflex processing is mechanical — it can sense but cannot assess or direct. A methodology with reflex and consciousness but no affect is **either overwhelmed** (every signal escalates) **or blind** (no signal escalates because there is no triage). The affect phase is the necessary filter that makes consciousness viable at scale.

Optimisation principle: minimise the load on the most expensive processor (consciousness) by filtering at cheaper layers first.

Mapping of specific methodology elements to processing phases:

Element	Phase	Why
Deterministic tests	Reflex	Fire unconditionally — pass/fail, no judgment
Event emission	Reflex	Every iteration appends to event log — no decision
Feature vector update	Reflex	State projection updated after every iteration
Protocol enforcement hooks	Reflex	Deterministic check of iterate side effects
Circuit breaker	Reflex	Automatic regression prevention — triggers on count, not judgment

Element	Phase	Why
Gap detection data	Reflex -> Affect	Data collection is reflex; classification and severity is affect
Signal source classification	Affect	Categorises delta as gap / discovery / ecosystem / optimisation / user / TELEM
Severity / priority weighting	Affect	Assigns urgency: “test regression on main” vs “minor style lint”
Escalation decision	Affect	Determines whether signal warrants conscious review
Human evaluator	Conscious	Requires judgment — approval, rejection, refinement
Agent evaluator (deliberative)	Conscious	Requires judgment — gap analysis, coherence assessment
Intent generation	Conscious	Full deliberative review — new intent, spec modification, vector spawning

Two compute regimes: The evaluators instantiate the ontology’s two compute regimes (#45):

- **Probabilistic** (stochastic expansion): LLM generation, design exploration, candidate production — the constructor proposes
- **Deterministic** (verification contraction): test execution, schema validation, contract checking — the evaluator disposes

The specification is the fitness landscape. Probabilistic compute explores it; deterministic compute verifies positions within it. This is the same expansion-contraction cycle seen in biology (mutation + selection), physics (quantum superposition + measurement), and

engineering (prototype + test). The two regimes are not optional alternatives — both are needed. Probabilistic compute without deterministic verification is hallucination. Deterministic compute without probabilistic exploration is stasis.

Principle 9: Two Sensory Systems

Three processing phases define *how* signals are processed. Two sensory systems define *where signals come from*:

System	Direction	What it observes	Examples
Interoception	Inward — the system's own state	Project health	Test suite staleness, dependency freshness, feature vector stall, event log gaps, coverage drift, build health, code/spec drift
Exteroception	Outward — the external environment	Ecosystem state	Dependency updates, CVE alerts, API deprecations, upstream breaking changes, user feedback, runtime telemetry

Interoceptive monitors observe the system's own health state continuously:

Monitor	What it observes	Signal type	Escalation example
Event freshness	Time since last event in events.jsonl	staleness	> 7 days -> escalate
Feature vector stall	In-progress vectors with no iteration for > N days	stall	> 14 days -> escalate
Test health	Coverage drift, flaky test rate, suite execution time	degradation	Coverage < threshold -> escalate
Dependency audit	Known vulnerabilities in lockfile dependencies	vulnerability	Any CVE severity >= high -> escalate

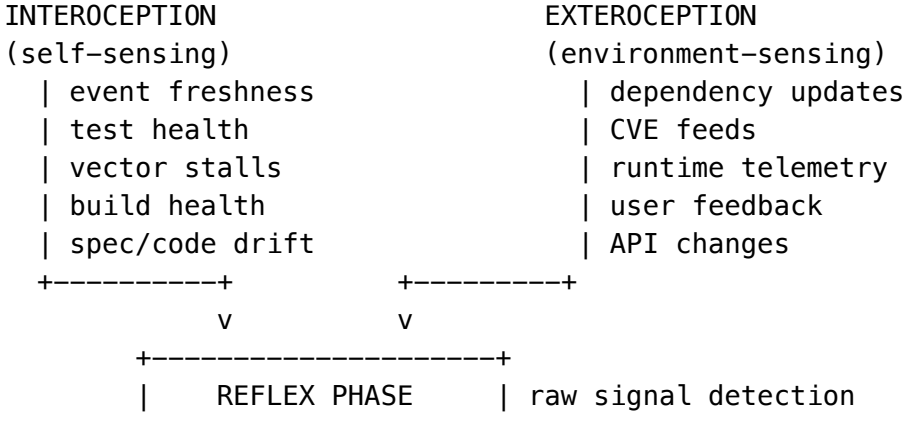
Monitor	What it observes	Signal type	Escalation example
Build health	CI/CD pass rate, build duration trends	degradation	Failure rate > 20% -> escalate
Spec/code drift	Code diverged from spec assertions (REQ tags missing)	drift	Any untagged code in traced module -> escalate

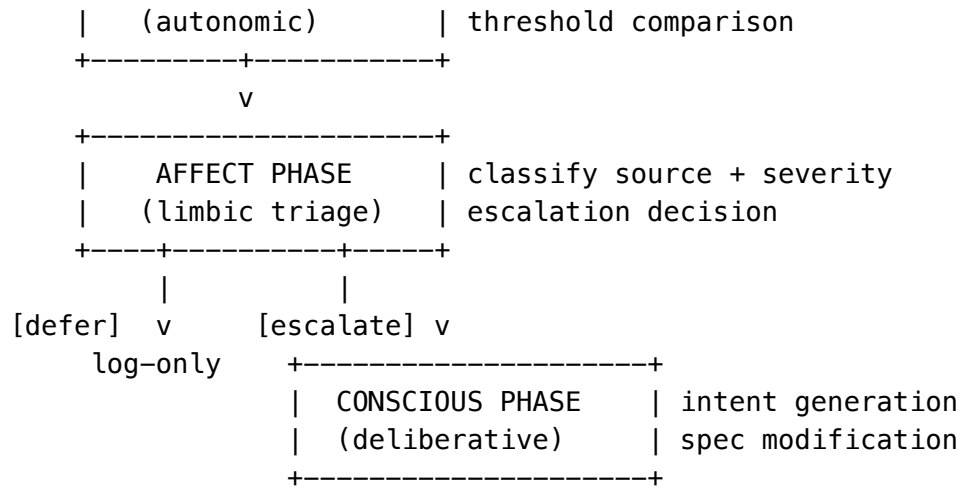
Exteroceptive monitors observe the external environment continuously:

Monitor	What it observes	Signal type	Escalation example
Dependency ecosystem	New major versions, deprecation notices, EOL	ecosystem_change	Major version -> escalate
CVE feeds	Published vulnerabilities affecting dependencies	security	Severity >= medium -> escalate
Runtime telemetry	Error rates, latency, SLA violations tagged with REQ keys	runtime_deviation	Error rate > baseline + 2sigma -> escalate
User feedback	Support tickets, feature requests, bug reports	user_signal	Cluster of 3+ similar reports -> escalate
API contract changes	Upstream API breaking changes, deprecation headers	contract_break	Any breaking change -> escalate

The critical property: interoception and exteroception run **independently of iterate()**. They are background processes — the system detects "your dependencies have a critical CVE" or "feature vector FV-AUTH-001 has been stalled for 3 weeks" even when no one is actively developing.

The two sensory systems feed into the three processing phases as a continuous pipeline:





A system with processing phases but no continuous sensory input is a brain without eyes or gut. It can think when stimulated but cannot notice.

The sensory systems are constitutive, not optional. The sensory-gradient loop is as fundamental as iterate(); the system that cannot sense is not alive (Principle 17). The sensory service runs as a long-running process that watches the workspace, runs monitors on schedule, performs affect triage, and generates draft proposals that cross a review boundary for human approval. The separation between autonomous sensing and human-approved changes is a structural invariant — the service can observe, classify, and draft, but only the interactive session with a human can cross the review boundary.

Principle 10: The IntentEngine is the Composition Law

The IntentEngine is **not a fifth primitive**. It is a **composition law** over the existing four — describing how Graph, Iterate, Evaluators, and Spec+Context compose into a universal processing unit at every scale.

```
IntentEngine(intent + affect) = observer -> evaluator -> typed_output
```

Component	What it does
intent	What this unit is trying to achieve
affect	Urgency and valence colouring the intent
observer	Senses current state — runs a tool, loads context, polls a monitor
evaluator	Classifies the observation's ambiguity level
typed_output	Always one of three exhaustive categories

Ambiguity classification — the evaluator's function is ambiguity measurement:

Ambiguity	Phase	Action
Zero	Reflex	Immediate action, no deliberation
Nonzero, bounded	Affect -> Probabilistic	Agent constructs next candidate
Persistent / unbounded	Escalate to higher consciousness	Human review, spec modification, vector spawning

Three output types — every IntentEngine produces exactly one:

Output	When	Maps to
reflex.log	Ambiguity = 0	`edge_converged`, `interoceptive_signal` (within bounds)
specEventLog	Ambiguity bounded	`iteration_completed` (with delta), `draft_proposal`
escalate	Ambiguity persistent	`intent_raised`, `spec_modified`, `spawn_created`

These three types are **exhaustive**: ambiguity partitions into exactly three regimes, and each has exactly one processing response.

Consciousness as relative: Level N's `escalate` output becomes Level N+1's reflex input. When an edge iteration escalates, the feature traversal level handles it as a straightforward routing decision. Conversely, Level N's `reflex.log` is invisible to Level N+1 — it is handled locally. The affect phase implements this filtering.

The IntentEngine operates at every scale:

Scale	Observer	Evaluator (ambiguity?)	Output types
Single iteration	Run constructor, produce candidate	Compare to edge evaluators	reflex (pass), specEventLog (iterate), escalate (stuck)
Edge convergence	Observe iteration history	All evaluators pass?	reflex (converged), specEventLog (more iterations), escalate (spawn)

Scale	Observer	Evaluator (ambiguity?)	Output types
Feature traversal	Observe edge convergence across graph	All edges converged?	reflex (feature done), specEventLog (next edge), escalate (blocked)
Sensory monitor	Poll/watch (intero/extero)	Threshold classification	reflex (within bounds), specEventLog (drift), escalate (breach)
Production homeostasis	Runtime telemetry	SLA/health assessment	reflex (healthy), specEventLog (degrading), escalate (incident)
Spec review	Review constraint surface	Constraint adequacy	reflex (stable), specEventLog (evolution needed), escalate (paradigm shift)

IntentEngine chaining: Units compose — one unit's output becomes the next unit's input. The affect (urgency, valence) propagates and transforms at each level:

```
Sensory monitor (IntentEngine at reflex scale):
    observer: poll dependency CVE feed
    evaluator: severity classification
    output: specEventLog (CVE found, severity = high)
        | affect: urgency elevated
Affect triage (IntentEngine at triage scale):
    observer: load CVE details + project dependency graph
    evaluator: impact assessment (does this affect us?)
    output: escalate (critical dependency affected)
        | affect: urgency = critical
Conscious review (IntentEngine at conscious scale):
    observer: load full context -- affected code, test coverage,
deployment state
    evaluator: human judgment -- "yes, this needs a hotfix"
    output: reflex.log (decision made -> spawn hotfix vector)
```

Note: at the conscious level, the human's decision is unambiguous (ambiguity = 0), so the output is `reflex.log`. The decision required deliberation, but the output is deterministic.

Graph discovery via IntentEngine feedback: The graph need not be predefined. Start with a single edge: `Intent -> Code`. If the evaluator reports zero ambiguity, the single edge suffices. But if persistent ambiguity persists — "I keep producing code that doesn't meet the intent because the design space is too wide" — that escalation is the signal that the edge is too coarse. The graph grows because the IntentEngine told it to. This is the abiogenesis pattern (Principle 4) operating in real time.

A key insight about reflexes: A reflex isn't "simple" — it is **unambiguous**. "Duck down! Run!" is a reflex because ambiguity = 0, not because the action is trivial. A complex build system is a reflex: it resolves deterministically regardless of complexity. Conversely, a one-line code change may require conscious deliberation if the intent is ambiguous. The classification is by ambiguity, not by difficulty.

The iteration loop as IntentEngine:

```
while true:
    output = IntentEngine(intent, affect, edge_evaluators)
    match output:
        reflex.log    -> promote(candidate); break     // converged
        specEventLog  -> continue                       // iterate
again
        escalate      -> spawn_or_escalate(); break     // beyond this
scope
```

This is the same `while not stable(candidate): iterate(...)` from Principle 6, with the IntentEngine naming what happens inside each cycle.

Principle 11: Context is the Constraint Surface

Context[] is the standing constraint surface — the collection of constraint documents that bound what the constructor can produce:

```
Context[] = {
    User Disambiguations,       // human clarifications of intent
    ADRs,                       // tech stack, environment,
architectural decisions
    Data Models,                // schemas, contracts, data lineage
    Templates,                  // structural patterns, code standards
    Prior Implementations,      // previous versions of this component
    Policy,                     // security, compliance,
organisational rules
    Graph Topology,             // the asset graph itself
    ...                         // open-ended, not a fixed list
}
```

Note: the **graph topology is Context[]**. The choice of which asset types and which edges exist is a standing constraint, not a universal law.

Each Context element narrows the space of admissible constructions:

```
Intent alone:           vast possibility space (degeneracy ->
hallucination)
+ ADRs:                 narrows to tech stack
  + Data Models:        narrows to schema-compatible
    + Templates:        narrows to pattern-conformant
      + Policy:         narrows to compliant
        + Prior:        narrows to evolution-of-existing
```

This is the ontology's constraint density (#16) in action. Sparse constraints -> probability degeneracy (#54) -> hallucination/failure. Dense constraints -> stable Markov objects (#7). **Context is what prevents hallucination in the construction process.**

Spec reproducibility: Context is versioned via content-addressable hashing. The context hash at each iteration is recorded in the event log, enabling: reproducibility (given the same context hash, the same iteration should produce the same evaluator results), debugging (when an iteration produces unexpected results, the exact context can be reconstructed from the hash), and evolution tracking (context hash changes between iterations reveal which constraints were added or modified).

Context sources are copied (not linked) to preserve hash stability. An ADR referenced by URI is resolved at project setup and copied into the local context store. This ensures that external changes to the referenced document don't silently alter the project's constraint surface.

Context as the anti-hallucination mechanism: The hallucination problem in LLM-driven construction is precisely the constraint density problem. An LLM generating code from a vague intent (sparse Context[]) has too many degrees of freedom — the probability landscape is degenerate (#54), multiple valid-looking outputs are equally likely, and the constructor picks one that may not match the intended semantics. Each Context element reduces degeneracy: ADRs eliminate technology choices, data models eliminate schema guesses, templates eliminate structural choices, prior implementations eliminate approach choices. **Hallucination is not a property of the LLM; it is a property of the constraint surface.** Dense enough constraints -> coherent output, regardless of the constructor. This is testable: measure hallucination rate (evaluator failure at first iteration) as a function of context density.

Principle 12: The Spec/Design Boundary

The asset graph contains a fundamental boundary between **specification** and **design**:

```
                Spec/Design boundary
                        |
Intent -> Requirements | Design -> Code <-> Tests -> ...
                        |
WHAT (tech-agnostic)    |  HOW (tech-bound)
One spec                |  Many possible designs
```

Everything upstream of the Requirements -> Design edge describes the problem without committing to technology. Everything downstream is bound to a specific technology stack.

This separation enables:

- **Multiple implementations**: The same spec (REQ-F-AUTH-001) can have variant_a, variant_b, variant_c — different designs for the same requirements. Telemetry enables data-driven variant selection.
- **Technology migration**: Change the design without changing the spec. Requirements survive platform shifts.
- **Disambiguation routing**: When iteration reveals ambiguity, the evaluator routes feedback to the right level — business gap -> Spec, technical gap -> Design.

The Requirements -> Design edge resolves **constraint dimensions** — categories of decisions that design must explicitly address:

Dimension	What it resolves	Example
Ecosystem Compatibility	Runtime platform, language, framework versions	Scala 2.13 + Spark 3.5 (not Scala 3 — incompatible)
Deployment Target	Where and how the system runs	Kubernetes, serverless, on-premise, hybrid
Security Model	Authentication, authorisation, data protection	OAuth2 + RBAC, mTLS, encryption at rest
Data Governance	Privacy, lineage, retention, compliance	GDPR, data classification, audit trail
Performance Envelope	Latency, throughput, resource bounds	p99 < 200ms, 10k RPS, horizontal scaling
Build System	Build tool, module structure, dependency management	sbt multi-module, Maven, Gradle
Observability	Logging, metrics, tracing, alerting	OpenTelemetry, structured logging, REQ-key tagging
Error Handling	Failure modes, retry strategy, circuit breaking	Fail-fast, retry with backoff, graceful degradation

Each dimension resolved adds constraint density at the design edge. These dimensions are **not universal** — they are graph package configuration. Different domains have different dimension taxonomies. A legal document graph might have dimensions for jurisdiction, precedent scope, and enforcement mechanism.

The graph package specifies which dimensions are **mandatory** (must be explicitly resolved via ADRs or design decisions) and which are **advisory** (should be considered but may use implementation defaults). The project binding provides the concrete values for each dimension.

Dogfooding observation: 5 of 7 bugs found at build time in an early instantiation were in dimensions the design left implicit (ecosystem compatibility, build structure). The evaluator bar appeared adequate (1-iteration convergence) precisely because missing dimensions weren't being checked — the evaluator can't detect what the constraint surface doesn't define. This is a general principle: **constraint density at the design edge directly predicts build-time failure rate.**

Constraint dimensions at the design edge make the Hamiltonian prediction operational: Each unresolved dimension is a component of V(constraint_delta). The more dimensions left implicit, the higher V at the downstream code edge, the more iterations required, the more expensive convergence. Conversely, each dimension explicitly resolved at design time reduces V at code time — the constructor has more constraints to work within, reducing degeneracy.

Principle 13: Features as Composite Vectors and Multiple Implementations

A **feature** is the composite of all assets produced along its trajectory through the graph:

```
Feature F = |req> + |design> + |module_decomp> + |basis_projections>
          + |code> + |unit_tests> + |uat_tests> + |cicd> + |telemetry>
```

Each component is a stable asset produced by iterating along an edge. The **REQ key** is the vector identifier — it tags which trajectory all these assets belong to. A feature is **complete** when all its edge-produced assets have converged to Markov objects.

Intent lineage is the feature's accumulated state across all edges: the original intent, every intermediate candidate, every decision made, every constraint satisfied. It is the full traceability chain, carried forward in each asset.

Feature views are generated cross-artifact status reports for a single REQ key, produced by grepping the tag format across all artifacts:

```
Feature View: REQ-F-AUTH-001
----------------------------
Spec:       specification/requirements.md            defined
Design:     imp_auth/design/DESIGN.md               component traced
Code:       src/auth/login.py:23                    Implements: REQ-F-
AUTH-001
Tests:      tests/test_login.py:15                  Validates: REQ-F-
AUTH-001
Telemetry:  dashboards/auth.json                    req="REQ-F-AUTH-
001"
Coverage:   5/5 stages tagged
```

REQ keys are not just documentation — they are **runtime-observable identifiers** that thread from spec to production. You can query production telemetry by REQ key to see how a specific requirement behaves, measure latency per feature, detect regressions per feature, and route incidents back to the originating requirement.

Multiple implementations per spec: A single requirement key can have multiple design variants:

```
REQ-F-AUTH-001 (spec)
  |-- design.variant_a  -> code.python  -> tests.python
  |-- design.variant_b  -> code.python  -> tests.python
  +-- design.variant_c  -> code.rust    -> tests.rust
```

Each variant is a separate trajectory through the downstream graph, sharing the same upstream spec. The variants can run in parallel, and telemetry from production enables data-driven selection between them. This is the Hilbert space (Part II, §II.4) in action — multiple vectors in superposition until measurement (production telemetry) collapses to the best-performing variant.

Feature dependencies: Features can depend on other features. Feature B's code asset might require Feature A's code asset to be stable first. Dependencies are between features (or their component assets), not between pipeline stages. This is a cross-vector constraint.

Task planning emerges from feature vector decomposition (#3, generative principle):

1. **Decompose** intent into feature vectors
2. **Trace** each vector's trajectory through the graph
3. **Identify** dependencies between assets across features
4. **Compress**: batch independent edges (parallel work), sequence dependent ones
5. **Result**: the task graph — with dependency order, parallelisation, and batching

```
                    Feature Vectors (trajectories through the graph)
                    +- F1 = |req> -> |design> -> |code> -> |tests>
Spec + Context[] -> +- F2 = |req> -> |design> -> |code> -> |tests>
                    +- F3 = |req> -> |code> -> |tests>
                                       ^
                                F3.|code> depends on F1.|design>

Task Graph = compress(F1 || F2, F1.|design> < F3.|code>)
```

The task graph is an emergent structure, not a prescribed plan.

Principle 14: The Gradient at Every Scale

The parent ontology identifies one computation applied everywhere there is a gradient:

```
delta(state, constraints) -> work
```

When delta -> 0, the system is at rest at that scale. When delta > 0, work is produced to reduce the gradient. This is the same operation at every scale:

Scale	State	Constraints	When delta -> 0
Single iteration	candidate asset	edge evaluators	evaluator passes
Edge convergence	asset at iteration k	all evaluators for edge	Markov object stabilises
Feature traversal	feature vector	graph topology + profile	feature converged
Project completion	all vectors	all features x all edges	ALL_CONVERGED

Scale	State	Constraints	When delta -> 0
Production homeostasis	running system	spec (SLAs, contracts, health)	system within bounds
Spec review	workspace state	the spec itself	spec and workspace aligned
Constraint surface update	irreducible delta	observation that the surface is wrong	new ground states defined

The last two rows are where the constraints themselves become the object of evaluation. There is no phase transition between levels. The same `delta(state, constraints) -> work` operates throughout. Complexity emerges because at larger scales, the constraints themselves become state for the next level.

Deacon hierarchy mapping:

Deacon level	Methodology scale	What maintains itself
Homeodynamic (#47)	Single iteration	Candidate converging toward evaluator pass
Morphodynamic (#48)	Edge / feature convergence	Pattern formation — feature vector traversing the graph
Teleodynamic (#49)	Production homeostasis	Running system maintaining its own boundary conditions
Beyond Deacon (#50)	Spec review + constraint update	The constraint surface itself updating from experience

The formal Hamiltonian (Part II, §II.5) captures this: `H(edge) = T(iteration_cost) + V(constraint_delta)`. When the spec updates, the Hamiltonian shifts — former ground states acquire potential energy under the new constraints. The gradient is what drives the system back toward new minima.

The spec as constraint surface: the spec is the reference frame against which all observation is evaluated:

```
Spec     = target manifold (encoded boundary conditions)
Observed = current state (assets, running system, telemetry)
Delta    = Spec − Observed = gradient signal
```

All signals — gaps, discoveries, ecosystem evolution, optimisation, user feedback, methodology self-observation — converge on the spec and radiate outward as new or modified feature vectors.

The gradient at production scale: The deployed, running system is itself a Markov object (#7): boundary (interfaces, SLAs, contracts), internal dynamics (runtime behaviour), conditional independence (runs without knowing how it was built). Production homeostasis is `delta(running_system, spec) -> correction`. When the running system deviates from the spec's boundary conditions (SLA violation, error rate spike), the delta is non-zero and the system produces work to restore alignment. The Observer/Evaluator at runtime uses the same three evaluator types: Human (user feedback, incident response), Agent (anomaly detection, pattern analysis), Deterministic Tests (alerting thresholds, SLA checks, health probes).

Principle 15: The Specification is a Living Encoding

The specification is not a static document. It is a **living encoding** (#46) — continuously evolving, defining the target, compared against runtime, updated on deviation, driving corrective construction.

When the spec absorbs a signal and updates, it emits a `spec_modified` event — what changed, why, which intent triggered it, which vectors it spawns. The system can observe its own spec modifications, evaluate whether they improved outcomes, and modify its modification strategy.

The spec review cycle spans all three processing phases:

1. **Reflex**: Construction via iterate() produces sensory substrate (events, metrics)
2. **Affect**: Sensory service classifies signals by source and severity, decides escalation
3. **Conscious**: Deliberative review — intent raised, spec modified, vectors spawned

Stage 2 (affect) is the critical filter. It determines which deltas reach conscious processing. A test regression on main has high severity — escalate immediately. A minor lint warning has low severity — log and defer. The affect phase prevents consciousness from being overwhelmed by signal volume. Profile settings tune the affect thresholds: a hotfix profile escalates aggressively; a spike profile suppresses most signals.

Every signal source feeds into the spec, and the spec re-expresses non-zero deltas as feature vectors:

Signal source	Example	Spec impact	Re-expressed as
Gaps (from iterate)	Design edge missing security dimension	Spec adds constraint dimension	Feature vector: update designs
Discoveries (from PoC/spike)	"We can use vector DB for this"	Spec adds capability	Feature vector: integrate, update design
Ecosystem evolution	Language version upgrade	Spec updates tech constraints	Feature vector: migration
Optimisation (from telemetry)	P99 latency 3x budget	Spec tightens performance bounds	Feature vector: optimise hot path

Signal source	Example	Spec impact	Re-expressed as
User feedback	"Search should support fuzzy matching"	Spec adds requirement	Feature vector: new REQ
Methodology self-observation	Evaluator skip rate too high	Graph package updates edge config	Meta-vector: improve evaluators

Intent events as first-class objects: For the spec review to close with full traceability, the `intent_raised` event captures the complete causal chain — trigger (which signal caused this), delta (what deviation was observed), signal_source (gap / discovery / ecosystem / optimisation / user / TELEM), vector_type (what kind of response), spec_impact (which REQ keys affected), spawned_vectors (what enters the graph), and prior_intents (chain of intents that led here). The `prior_intents` field closes the reflexive loop: if intent A modifies the spec, and the modified spec triggers intent B that traces back to A — the system detects the consequences of its own modifications.

Spec change events — when the spec absorbs a signal and updates, it emits a `spec_modified` event:

```
{
  "event": "spec_modified",
  "trigger_intent": "INT-2026-042",
  "signal_source": "ecosystem",
  "what_changed": ["REQ-NF-COMPAT-001 updated: Scala 2.13 -> 3.x"],
  "why": "Scala 2.13 end-of-life detected via ecosystem monitoring",
  "affected_req_keys": ["REQ-NF-COMPAT-001", "REQ-NF-BUILD-003"],
  "spawned_vectors": ["FV-MIGRATION-001"],
  "prior_intents": ["INT-2026-038"]
}
```

This enables: spec archaeology (why does this constraint exist? — trace to the spec_modified event), feedback loop detection (intent A -> spec change -> intent B -> if B traces back to A, the loop is visible), impact analysis (spec change -> spawned vectors -> telemetry -> new intents — the full causal chain is in the event log), and rate-of-evolution analysis (how often does the spec change? which sections are volatile, which are stable? — all derivable from spec_modified events).

The total intentional state — the superposition of all in-flight feature vectors — is the system's response to **everything it knows**, continuously updated as the spec absorbs new signals. When the spec updates, the Hilbert space (Part II, §II.4) undergoes a basis change: new basis vectors appear (new requirements add dimensions), existing vectors shift (updated constraints change the landscape), the Hamiltonian shifts (former ground states acquire potential energy under new constraints, spawning vectors that seek new minima). This is the gradient operating at all scales simultaneously — and this simultaneous operation is what makes the system alive (Principle 17).

Principle 16: Execution is Event-Sourced

The methodology's execution model is **event sourcing**: all state changes are recorded as immutable events, and all observable state is a projection (derived view) of the event history.

Events record **what happened**, not what the current state is. The event log is append-only. Core event types:

Event type	When emitted	What it records
`iteration_completed`	Every iterate() cycle	Edge, feature, iteration count, evaluator results, context hash
`edge_started`	First iteration on an edge for a feature	Entry into a new graph edge
`edge_converged`	All evaluators pass for an edge	Promotion of candidate to Markov object
`evaluator_ran`	Individual evaluator execution	Which evaluator, what result, what delta
`finding_raised`	Gap detected	Backward/forward/inward gap data
`feature_spawned`	New feature vector created	Spawn trigger, parent, child type
`feature_folded_back`	Discovery/spike results folded into parent	Child outputs, context enrichment
`spec_modified`	Spec absorbs signal, updates	What changed, why, trace to triggering intent
`intent_raised`	Deviation triggers new intent	Trigger, delta, signal source, spawned vectors, prior intents

All methodology state is computed from the event log:

Projection	Derived from	Purpose
STATUS	All events	Current state of all edges, features, evaluators
Feature vectors	Edge convergence events	Which features have converged at which edges

Projection	Derived from	Purpose
Task lists	Feature decomposition + dependency events	What work remains, in what order
Feature views	Tag-grepping events + artifact state	Per-REQ cross-artifact status
Gap analysis	Finding events + evaluator results	What the methodology missed and why
Telemetry signals	Observer evaluation events	Self-observation of methodology performance

Projections are **re-derivable**: given the event log, any projection can be reconstructed at any point in time. This provides auditability, reproducibility, evolution (new projections without changing the log), and debugging.

The event log is the methodology's analogue of the constraint network's evolution history (#4 — unit of change). Projections are manifold-level observables — views that emerge when the raw event stream is coarse-grained through a particular lens.

Protocol enforcement hooks make the side effects of iteration mandatory. The iterate protocol produces mandatory artefacts beyond the output asset: event emission, feature vector update, state view regeneration, and gap data. These side effects are **the telemetry** — without them, the methodology loses observability. Protocol hooks are the methodology's reflex arc — they fire unconditionally at every iteration boundary, verifying that mandatory side effects occurred:

Check	What it verifies	Gap it prevents
Event emitted	events.jsonl has new entry since edge started	Lost observability
Feature vector updated	Active feature .yml modified since edge started	Stale lifecycle tracking
State view regenerated	State view modified since edge started	No computed projections
Source findings present	Iteration event contains source_findings	Backward gap detection skipped
Process gaps present	Iteration event contains process_gaps	Methodology improvement signal lost

A circuit breaker prevents infinite regression: the hook blocks once, the agent completes missing side effects, the second check allows through unconditionally. The agent retains full freedom in generation strategy — the hooks only check the observable outputs of the protocol, not how the agent generates. This mirrors how deterministic tests work for code: the test doesn't care how the function was implemented, only that the contract is satisfied.

Dogfooding observation: In an early instantiation (test05), the agent bypassed the iterate protocol for 3 of 4 edges to optimise for generation speed. Result: high-quality artefacts but only 1 event in events.jsonl (vs 5 expected), stale feature vector, no STATUS. The methodology's observability benefits evaporated because the side effects were optional.

Self-observation: The methodology observes itself using the same evaluator pattern it uses for artefacts. TELEM signals — iteration counts per edge, evaluator skip counts, finding redundancy, build success/failure, bug category at build time — feed back into the graph package. They are the mechanism by which the methodology's own topology and evaluator checklists evolve:

```
Level 1 (product):     |running_system> -> |telemetry> -> |observer> -
> |new_intent>
Level 2 (methodology): |methodology_run> -> |TELEM_signals> ->
|observer> -> |graph_package_update>
```

Both levels use the same three evaluator types, both produce events, both close the feedback loop. The methodology is a Markov object that maintains itself (#49).

Principle 17: The System is Alive

When the full lifecycle is operational — CI/CD running, telemetry streaming, the gradient active at every scale — the system exhibits six structural properties:

1. **Continuous sensing** — interoceptive and exteroceptive monitors run independently of iterate(), computing delta(observed, expected). The system notices even when no one is actively developing.
2. **Concurrent vector lifecycles** — many feature vectors in different phases simultaneously, not a sequential pipeline.
3. **Continuous metabolism** — CI/CD runs continuously. The system is always building, testing, deploying.
4. **Active perception** — telemetry streams continuously. The system is always observing itself.
5. **Reflexive self-modification** — spec review operates continuously. The constraint surface is always absorbing signals and spawning responses.
6. **Selective pruning** — vectors whose basis dimension collapses (requirement removed, superseded) are cancelled. The system doesn't just grow; it prunes.

No single property creates the living quality. It is the simultaneous operation of all six — sensing, concurrency, metabolism, perception, reflection, pruning — that produces the living system. Each is the gradient operating at a different scale; all scales active simultaneously = alive.

Basis change under spec update: When the spec updates via the gradient check (Principle 15), the Hilbert space undergoes a basis change:

- **New basis vectors appear**: A new requirement (REQ-F-SEARCH-002) adds a dimension to the vector space. New feature vectors can now exist in this dimension.
- **Existing vectors shift**: An ecosystem migration changes the design constraints. Code that was at ground state under the old spec has high potential energy under the new spec — it "wants" to move.
- **Hamiltonian shifts**: The potential energy landscape V(constraint_delta) is redefined. What was optimal is no longer optimal. New ground states exist.
- **Vectors are spawned**: Each spec change spawns one or more feature vectors (discovery, PoC, feature, hotfix) that enter the graph and traverse edges via iterate().

The total energy budget is finite. Spec updates that raise V across many assets create pressure to re-optimise — the system must find new ground states under the new Hamiltonian. This is relentless optimisation made formal.

The Markov boundary as concurrency enabler: When a feature vector's component converges at an edge, it achieves Markov object status — a stable boundary, conditional independence, evaluator-confirmed stability. This boundary is what allows other vectors to depend on the converged asset without coupling to its internals. Without Markov boundaries, concurrent vectors would require global coordination. With them, coordination is local. The living system is not a monolith; it is an **ecology of Markov objects**, each with a clean boundary, interacting through interfaces, evolving independently within their boundaries.

The Markov boundary as concurrency enabler in detail: When a feature vector's component converges at an edge, it achieves Markov object status. This boundary is what allows other vectors to depend on the converged asset without coupling to its internals or its construction process:

```
FV-AUTH (converged at |code>):
    Boundary: AuthService interface, 47 passing tests, REQ-F-AUTH-001
traced
    Inside: 3 classes, 400 lines, OAuth2 implementation
    History: 5 iterations at design, 3 at code, 1 evaluator escalation

FV-SEARCH (in-flight at |design>):
    Depends on: AuthService.boundary   <- only the boundary, not the
internals
    Does NOT need to know: how AuthService was built, how many
iterations it took,
                           what design alternatives were considered
```

The more vectors in flight, the more critical the Markov boundary becomes. In a living system with dozens of concurrent vectors, the Markov boundary at each convergence point prevents combinatorial explosion of coordination. Each converged asset is a stable island that other vectors can build on.

At any moment, many feature vectors are in-flight simultaneously, each at a different lifecycle stage:

```
|FV-AUTH-001>     = |req:converged> + |design:converged> +
|code:iteration_3> + |tests:pending>
|FV-SEARCH-002>   = |req:converged> + |design:iteration_1>
|FV-PERF-003>     = |req:gestating>                          <- just
spawned from telemetry
```

```
|FV-MIGRATE-004>  = |req:converged> + |design:converged> +
|code:converged> + |tests:converged> + |cicd:deploying>
|FV-HOTFIX-005>   = |code:iteration_1>                      <- fast-
tracked, skip design
```

The biological correspondence is structural:

Living system	**SDLC system**	**Structural correspondence**
DNA	Spec (the constraint surface)	Information that directs construction, updated from experience
Proteins / cells	Feature vectors at various lifecycle stages	Constructed structures, each in a different phase
Metabolism	CI/CD loop	Continuous transformation: code -> build -> deploy -> run
Interoception	Event freshness, test health, vector stall, build health	Gradient sensing: delta(system_health, expected_health)
Exteroception	Dependency ecosystem, CVE feeds, runtime telemetry	Gradient sensing: delta(environment, assumptions)
Autonomic system (reflex)	Event emission + protocol hooks + circuit breaker	The gradient at iteration scale — fires unconditionally
Limbic system (affect)	Signal classification + severity + escalation	The gradient at triage scale — assigns urgency
Frontal cortex (conscious)	Evaluators + spec review	The gradient at spec scale — deliberative judgment
Immune system	Evaluator network (tests, agent checks)	Boundary-condition enforcement
Homeostasis	Telemetry -> observer -> correction	The gradient at production scale
Reproduction	Vector spawning from intent events	New vectors born from non-zero gradient
Apoptosis	Vector cancellation (requirement removed)	Vectors whose basis dimension collapses are terminated
Evolution	Graph package updates from TELEM signals	The encoding itself evolves

Progressive gradient activation: The living system does not arrive fully formed. It follows its own abiogenesis sequence:

1. **Start with iterate()** — a team building software using the four primitives. The gradient operates at the iteration and edge scale. This is a pipeline, not yet alive.
2. **Add CI/CD** — the metabolism starts. The gradient now operates at the deployment scale. Continuous building and testing. Still not alive — no perception.
3. **Add telemetry** — the nervous system activates. The gradient now operates at the production scale. The system can detect deviations. Not yet alive — no self-modification.
4. **Add spec review** — the gradient at the spec scale. The constraint surface absorbs signals and updates. The system modifies its own encoding. Getting close — but the sensory systems are still human-driven.
5. **Add continuous sensing** — interoception and exteroception run independently. The system notices even when no one is actively developing. Now the gradient operates at all scales simultaneously. **The system is alive.**

Each step is an emergence: the gradient at each new scale enables capabilities that the previous scales could not produce. No step is a distinct phase — each is the natural consequence of the gradient being applied at a larger scale. The progression follows Deacon's hierarchy: homeodynamic (iteration) -> morphodynamic (feature convergence) -> teleodynamic (self-maintenance) -> representational (self-modifying encoding).

This is the ontology's teleodynamic transition (#49) at full expression: a self-maintaining, self-modifying system that acts on its own behalf, with multiple concurrent processes, continuous energy consumption, and the capacity to observe and direct its own evolution.

Part II: The Formal System

II.1 Three-Layer Architecture

The methodology separates into three layers of increasing specificity:

```
+-------------------------------------------------------------+
|  Layer 1: ENGINE (universal)                                |
|  4 primitives + iterate() + evaluator type taxonomy         |
|  + IntentEngine composition law + event sourcing model      |
|  + protocol enforcement hooks                               |
|  Same across all domains. Never changes per graph or project. |
+-------------------------------------------------------------+
|  Layer 2: GRAPH PACKAGE (domain-specific)                   |
|  Topology + edge configs + constraint dimension taxonomy    |
|  + evaluator checklists + projection profiles               |
|  One per domain (SDLC, legal, science, ...)                 |
+-------------------------------------------------------------+
|  Layer 3: PROJECT BINDING (instance-specific)               |
|  project_constraints + context URIs + threshold overrides   |
|  + team conventions + CI/CD integration                     |
|  One per project                                            |
+-------------------------------------------------------------+
```

Layer 1: Engine — The four primitives, the iteration function, the evaluator type taxonomy, the IntentEngine composition law, the feature vector formalism, the event sourcing execution model, and the protocol enforcement hooks. Universal across all domains. A legal document workflow, a physics paper pipeline, and a software SDLC all use the same engine.

Layer 2: Graph Package — A domain-specific instantiation: the graph topology (which asset types exist, which transitions are admissible), edge parameterisations (evaluator checklists and weighting per edge), constraint dimension taxonomy, and projection profiles. The SDLC graph is one graph package. Each package follows the abiogenesis pattern: practitioners in a domain work, patterns crystallise, the topology and edge configs encode those patterns.

Layer 3: Project Binding — Instance-specific configuration: technology constraints (language, frameworks, versions), context URIs (pointers to reference documents, API specs, compliance policies), evaluator threshold overrides, team conventions, and CI/CD integration details.

Context sources are URI references to external Architecture Decision collections. At project setup, referenced collections are resolved and copied into the local context store. This enables organisation-level standards, team conventions, and platform decisions to flow into project Context[] without duplication. Sources are copied (not linked) to preserve content-addressable hashing for spec reproducibility.

The three-layer separation explains why the same methodology can produce radically different outcomes in different domains: the engine is invariant, the graph package captures domain expertise, and the project binding captures local constraints. Changing the graph package changes the domain. Changing the project binding changes the project. The engine never changes.

This separation is itself an instance of the ontology's construction pattern (#38):

Construction pattern	**Layer**
Constructor (universal, substrate-independent)	Engine
Encoded representation (domain knowledge crystallised)	Graph Package
Construction context (specific constraints of this instance)	Project Binding

II.2 Functors: Spec + Encoding -> Executable Methodology

A methodology instance is a **functor** — a spec composed with an encoding:

```
Functor(Spec, Encoding) -> Executable Methodology
```

A **spec** defines functional units — typed assets, admissible transitions, evaluator slots, convergence criteria. Technology-agnostic. An **encoding** maps each functional unit to an execution category:

Category	Symbol	Rendering
Deterministic	F_D	Computable, repeatable, verifiable. Tests, builds, schema validation.
Probabilistic	F_P	LLM/agent under constraints. Code generation, gap analysis, design synthesis.
Human	F_H	Judgment, approval, domain evaluation. Spec review, intent authoring, acceptance.

Projections are functors. Each named profile (full, standard, PoC, spike, hotfix) is a different encoding of the same spec. **Zoom is an encoding operation**: when an edge is zoomed in, implicit functional units become explicit and each gets its own encoding. The spec doesn't change; the encoding changes.

Projections are functors: each named profile (full, standard, PoC, spike, hotfix) is a different encoding of the same spec. What the encoding determines:

Profile	Encoding strategy
full	Maximum explicit rendering — all units encoded, all categories active
standard	Balanced — key units explicit, routine units collapsed
poc	Minimal — only hypothesis-critical units encoded
hotfix	Emergency — only fix-critical units encoded, F_D dominant

Zoom is an encoding operation: when an edge is zoomed in, implicit functional units become explicit — each gets its own encoding (execution category). When zoomed out, those units' encodings collapse back into the encapsulating edge. The spec doesn't change; the encoding changes.

```
Zoomed out:   design ======================================> code
              (single encoding: F_P)

Zoomed in:    design -> module_decomp -> basis_projections ->
code_per_module
              (F_P)    (F_P)           (F_P + F_H)          (F_P +
F_D)
                                        ^ human waypoint     ^ tests
```

Multiple implementations per spec are multiple functors from the same domain:

```
Spec: REQ-F-AUTH-001 (functional units defined)
  |-- Encoding_Claude:  F_P(Claude) + F_D(pytest) + F_H(CLI review)
```

```
  |-- Encoding_Gemini:  F_P(Gemini) + F_D(jest)   + F_H(web review)
  +-- Encoding_Codex:   F_P(Codex)  + F_D(cargo)  + F_H(PR review)
```

Same spec, three functors, three executable methodologies.

Natural transformation between encodings: when the IntentEngine escalates — ambiguity exceeds the current category's capacity — it performs a natural transformation:

```
eta: F_D -> F_P    (test failure -> agent investigation)
eta: F_P -> F_H    (agent stuck -> human review)
eta: F_H -> F_D    (human approves -> deterministic deployment)
```

The escalation chain F_D -> F_P -> F_H and the delegation chain F_H -> F_P -> F_D are natural transformations between functors. The IntentEngine's ambiguity classification determines which transformation fires. The affect system modulates the transformation threshold — a hotfix profile escalates faster, a spike profile tolerates more ambiguity.

The functor formalism explains why the four primitives suffice: **Graph** defines the domain, **Iterate** is the functor application, **Evaluators** classify which transformation fires, and **Spec + Context** is the constraint surface. No fifth primitive is needed.

II.3 Projections as Functors

A projection is a **functor**: `Functor(Spec, Encoding) -> Executable Methodology`. The spec defines the functional units (typed assets, admissible transitions, evaluator slots). The encoding maps each unit to an execution category (F_D, F_P, F_H). Different encodings of the same spec produce different projections — each a valid methodology instance.

This explains why the five things that can vary in a projection are precisely the encoding parameters:

What varies	Encoding dimension
Graph size	Which functional units have explicit encodings (zoom level)
Evaluator composition	Which execution category renders each evaluator slot
Convergence criteria	What delta < epsilon means in each category
Context density	How many constraints the encoding must satisfy
Iteration depth	How many functor applications before timeout

And why the four things that cannot vary are the functor's preconditions — without a graph there is no domain, without iteration there is no application, without evaluation there is no classification, without context there is no constraint surface. Remove any one and the functor is undefined.

Named profiles are named encodings. Zoom operations are encoding refinements — making implicit functional units explicit. The natural transformation eta between categories (F_D -> F_P -> F_H) is the IntentEngine's escalation mechanism — the

functor adapts its encoding when ambiguity exceeds the current category's capacity.

A **projection** is a valid lighter instance of the formal system:

```
projection(G_full, profile) -> G_projected
    where valid(G_projected) = true
```

Projections are specified along four independent dimensions:

Dimension	Full	Light	Minimal
Graph	All asset types, all edges	Subset of edges, some collapsed	2 nodes, 1 edge
Evaluators	Human + Agent + Deterministic per edge	Agent + Deterministic, human on key edges	Agent only, or human only
Convergence	All checks pass (delta = 0)	Required checks pass	Question answered, or time box expired
Context	Full constraint surface (ADRs, models, policy)	Project constraints + intent	Intent only

These dimensions are independent — you can have a full graph with light evaluators, or a minimal graph with strict evaluators. The full SDLC graph is the top element of a partial order on projections. The minimal projection (1 edge, 1 evaluator, intent only) is the bottom.

Traceability is not a fifth invariant. It is an emergent property of the four invariants working together: Graph provides the path, Iterate provides the history, Evaluators provide the decisions, Spec+Context provides the constraints. If all four hold, traceability is recoverable. The REQ key system is the practical encoding of this emergent property. In lighter projections, traceability may be coarser (e.g., "this code came from that intent" without intermediate design documents), but the causal chain from intent to artefact is never fully severed. Feature views degrade gracefully with projection: fewer stages, same key.

The IntentEngine is projection-invariant: the same `observer -> evaluator -> typed_output` structure operates in every valid projection. What projections tune:

- **Ambiguity thresholds**: A hotfix profile has aggressive escalation (low tolerance for nonzero ambiguity). A spike profile tolerates more ambiguity before escalating.
- **Affect weights**: A full profile weights all signal sources equally. A minimal profile may suppress ecosystem signals entirely.
- **Observer depth**: A full projection runs all interoceptive + exteroceptive monitors. A spike runs only monitors relevant to the experiment.

What projections cannot change:

- The observer/evaluator structure (every edge traversal is an IntentEngine invocation)
- The three output types (reflex.log, specEventLog, escalate)
- Ambiguity as the routing criterion (zero -> reflex, bounded -> iterate, persistent -> escalate)

Projection verification — given a projection P, verify:

```
1. Graph check:
   - P.graph has at least 1 edge
   - All edges connect defined asset types
   - No orphan nodes

2. Iterate check:
   - Every edge has iterate() defined
   - Max iterations is defined (finite or explicit "unlimited")

3. Evaluator check:
   - Every edge has at least 1 evaluator
   - Each evaluator has a convergence criterion

4. Context check:
   - P has a non-empty context (at minimum: intent)
   - Context is versioned (hash computable)

5. Convergence check:
   - Every edge has a stopping condition
   - Time box is defined (or explicitly "no time box")
```

Two projections can be compared: P1 is a sub-projection of P2 iff P1.graph is a subgraph of P2.graph, P1's evaluators are a subset per edge, P1's convergence is less strict, and P1's context is a subset. This gives a partial order. The full SDLC graph is the top element; the minimal projection (1 edge, 1 evaluator, intent only) is the bottom.

II.4 Feature Vectors as Hilbert Space

Feature vectors form a vector space whose structure determines parallelism, cost, and evolution:

Hilbert space concept	**Feature vector interpretation**
Basis	Asset types. The spec defines which basis vectors are active — adding a requirement adds a dimension; removing one collapses it
Vectors	Composite features: F = Sum
Inner product	Feature overlap — shared modules at the build decomposition level. Features sharing many modules have high inner

Hilbert space concept	Feature vector interpretation
	product and must coordinate; features sharing none are orthogonal
Orthogonality	Independent features — zero shared modules, fully parallelisable
Norm	Vector cost — sum of iteration effort across all components
Superposition	Total intentional state — all features in-flight, components not yet converged
Measurement/collapse	Evaluator convergence — component converges to stable Markov object
Basis change	Spec update — the vector space itself transforms. New dimensions appear, old dimensions shift, existing vectors acquire potential energy under the new basis

Basis projections are the projection of a feature vector onto its minimal module basis — the smallest subset of modules that makes one priority feature work end-to-end. Once a basis projection converges, it becomes a Markov object — a stable, tested, end-to-end slice. Projections that share no modules are orthogonal and can execute in parallel with zero coordination.

Example from a data pipeline module decomposition:

```
Module Decomposition (from Design):
    model <- compiler <- runtime <- spark <- lineage <- ai <- context
<- testkit

Basis Projection 1 (structural mapping):   model -> compiler(path) ->
runtime(exec) -> spark
Basis Projection 2 (lossy aggregation):    model -> compiler(+grain) -
> runtime(+fold) -> spark
Basis Projection 3 (full lineage):         model -> compiler ->
runtime -> lineage -> spark
```

Key property: basis projections that share no modules are orthogonal — they can execute in parallel with zero coordination. Projections that share modules must build the shared modules first, then diverge. The Hilbert space inner product is the number of shared modules: features with zero shared modules have zero inner product and are fully parallelisable.

Each converged basis projection is a Markov object: - **Boundary**: Its public interfaces, the features it proves, the tests it passes - **Conditional independence**: Downstream projections can build on it without knowing how it was constructed - **Stability**: All evaluators report convergence (compiles, tests pass, REQ keys traced)

The basis projection schedule — the Gantt chart — is a **derived projection** computed from: module dependency DAG, feature-to-module mapping, feature priority, and basis projection convergence events. This enables incremental delivery: working software after the first basis projection converges, progressively richer after each subsequent one.

Context stability: Context[] is largely shared across the graph. ADRs, Data Models, Policy — these don't change per edge. They are the standing constraint surface. What changes per edge is which subset is relevant and how the constructor weights them. Context itself evolves, but on a slower timescale than assets. This is the ontology's scale-dependent time (#23): the constraint surface updates slowly while components iterate rapidly upon it.

Basis projections and the build schedule: The basis projection schedule — the Gantt chart — is a **derived projection** (Principle 16) computed from four inputs:

- Module dependency DAG (from |module_decomp>)
- Feature-to-module mapping (from REQ key traceability)
- Feature priority (from requirements / intent)
- Basis projection convergence events (from event log)

This enables incremental delivery: working software after the first basis projection converges, progressively richer after each subsequent one. The methodology produces not just code but an **observable build plan** — a Gantt that updates as basis projections converge. The two compute regimes (#45) map directly into the Hilbert space:

- **Probabilistic compute** (LLM) = exploration of the Hilbert space — superposition of possible constructions
- **Deterministic Tests** = measurement — projection onto eigenstate, collapse to pass/fail

II.5 The Hamiltonian: Cost and Effort

The Hamiltonian is the operator governing evolution (effort/cost) on the feature vector space — the formal expression of the gradient (Principle 14):

```
H(edge) = T(iteration_cost) + V(constraint_delta)
```

Hamiltonian concept	Interpretation
T (kinetic energy)	Iteration cost — compute, human time per cycle
V (potential energy)	Constraint difficulty — evaluator delta from convergence
Ground state	Minimal-effort solution satisfying all constraints
Excited states	Over-engineered solutions (more energy than necessary)
Energy conservation	Budget constraint — total effort bounded

T scales with: evaluator type (human expensive, tests cheap, LLM moderate) and iteration count. V scales with: constraint density (#16) — sparse constraints = high potential (hard to converge), dense constraints = low potential (well-defined problem).

Hamiltonian shift under spec update: When the spec updates, the potential energy landscape is redefined. A converged asset at ground state under the old spec may have high V under the new spec — it "wants to move." The migration vector spawned by the intent event is the system's response to this potential energy.

Spec update type	Hamiltonian effect	System response
New requirement added	New dimension in V	Spawn feature vector in new dimension
Constraint tightened	V increases for assets near old bound	Spawn optimisation vector
Ecosystem shift	V increases for all assets bound to old ecosystem	Spawn migration vector
Requirement removed	Dimension collapses	Simplification vector (reduce complexity)
Discovery (new capability)	New low-energy path appears	Spawn PoC vector to explore ground state

Task planning as action minimisation: Decompose into basis vectors, identify orthogonal subsets (parallel work), project overlapping vectors onto shared components (batching), sequence by dependency constraints, optimise for minimum total H.

Constraint density (#16) is the **metric** on the space — it determines how much effort transitions require. This connects hallucination prevention to cost dynamics: sparse constraints = high V = expensive to converge. The connection runs through three domains:

Domain	Sparse constraints	Dense constraints
Physics	Low constraint density = flat spacetime, no structure (#24)	High constraint density = strong gravity, structure formation
LLMs	Sparse context = degenerate probabilities = hallucination (#54)	Dense context = focused probabilities = coherent output
SDLC	Sparse Context[] = high V = expensive convergence, many iterations	Dense Context[] = low V = cheap convergence, fewer iterations

The Hamiltonian formalism is not merely decorative. It makes a specific prediction: **the total cost of a feature is predictable from its constraint density profile before construction begins.** Edges with dense Context[] (well-specified requirements, rich ADRs, clear templates) will have low V and converge cheaply. Edges with sparse

Context[] (vague requirements, no ADRs, no templates) will have high V and converge expensively — or fail to converge at all (hallucination). This prediction is testable against event logs.

Part III: Vector Types and Spawning

III.1 Five Vector Types

Not all work is feature delivery. The formal system supports multiple vector types, each a different projection:

Vector Type	Purpose	Graph	Convergence	Fold-back
Feature	Deliver a capability	Full or standard projection	All required checks pass (delta = 0)	Permanent — assets become Markov objects
Discovery	Answer a question	Minimal graph (1-2 edges)	Question answered OR time_box_expired	Context[] — findings become constraints
Spike	Assess technical risk	Subset graph (code + tests)	Risk assessed OR time_box_expired	Context[] — findings + recommendation
PoC	Validate feasibility	Standard or light projection	Hypothesis confirmed/rejected OR time_box_expired	Context[] + optionally promoted assets
Hotfix	Fix production issue	Minimal graph (code -> tests)	Fix verified in production	Permanent — but spawns feature vector for proper remediation

Vector type detail:

Discovery Vector — a time-boxed investigation spawned when a gap is detected:

```
Discovery D = |question> -> |investigation> -> |findings>
Convergence: question_answered OR time_box_expired(fold_back_partial)
Result: Context[] entry (findings become constraints for parent
vector)
```

Properties: spawned by a parent vector when a gap is detected, has a specific question to answer, does not produce permanent code assets (unless promoted), findings fold back as Context[], time-boxed.

Spike Vector — a time-boxed technical experiment:

```
Spike S = |hypothesis> -> |experiment> -> |assessment>
Convergence: risk_assessed OR time_box_expired(fold_back_partial)
Result: Context[] entry (risk assessment + recommendation)
```

Properties: spawned when a feature vector encounters technical uncertainty, produces throwaway code (explicitly not production), assessment folds back as Context[] (often as an ADR).

PoC Vector — a time-boxed feasibility validation:

```
PoC P = |hypothesis> -> |design_sketch> -> |prototype> -> |validation>
Convergence: hypothesis_confirmed OR hypothesis_rejected OR
time_box_expired
Result: Context[] (validated/rejected approach) + optionally
promotable assets
```

Properties: spawned from a discovery gap or strategic question, uses lighter projection than features, may produce assets that can be promoted (not thrown away), folds back with go/no-go recommendation.

Hotfix Vector — an emergency trajectory through a minimal graph:

```
Hotfix H = |incident> -> |fix> -> |verification>
Convergence: fix_verified_in_production
Result: Permanent (deployed fix) + spawns remediation feature vector
```

Properties: spawned from production incident, minimal graph (skip design, skip UAT), always spawns a follow-up feature vector for proper remediation.

Extended convergence: Time-boxing does not violate the Iterate invariant. The stopping condition is extended, not removed. Every iteration still evaluates against convergence criteria:

```
converged(vector) =
    delta = 0                                        // standard
convergence
    OR (question_answered AND delta_required = 0) // discovery/spike
    OR time_box_expired(fold_back_partial)        // timeout: fold
back what we have
```

III.2 Spawning and Fold-Back

Any vector can spawn child vectors during iteration. The spawning function:

```
spawn(
    parent_vector,          // the vector that detected the need
    child_type,             // Discovery | Spike | PoC | Hotfix |
Feature
    trigger,                // what caused the spawn (gap, risk,
incident, scope)
    question_or_hypothesis, // what the child vector must resolve
```

```
    time_box,               // iteration budget (optional for Feature)
    projection_profile      // which projection to use
) -> child_vector
```

Spawning triggers:

Trigger	Spawns	Example
Gap detected	Discovery	Missing information about authentication requirements
Risk identified	Spike	“Can Kafka handle our throughput?”
Feasibility question	PoC	“Should we build a vector DB integration?”
Production incident	Hotfix	SLA breach detected via telemetry
Scope expansion	Sub-feature	Feature larger than expected

When a child vector converges (or times out), its outputs fold back to the parent:

Child Type	What Folds Back	Where It Goes
Discovery	Findings document	Parent’s Context[] — becomes a constraint
Spike	Risk assessment + recommendation	Parent’s Context[] — often becomes an ADR
PoC	Go/no-go + validated approach + promotable assets	Parent’s Context[] + optionally promoted assets
Hotfix	Deployed fix	Production + spawns remediation Feature
Feature (sub)	Converged assets	Parent’s dependency graph

Fold-back is Context[] enrichment. The child vector’s outputs become entries in the parent’s constraint surface. This is the same mechanism as loading an ADR — the child’s findings narrow the parent’s possibility space, reducing degeneracy and driving convergence. This maps to the ontology’s constraint density (#16): child vectors increase constraint density on the parent.

After fold-back: 1. Parent vector’s Context[] is updated with child outputs 2. Parent vector’s status changes from `blocked` to `iterating` 3. The context hash is recomputed (spec reproducibility) 4. Iteration resumes on the parent with richer constraints

Child vectors can spawn their own children (nested spawning). A PoC might spawn a Spike to assess a specific technology. A Feature might spawn a Discovery which spawns another Discovery. Nesting is bounded by: time box inheritance (child time boxes must fit within parent), configurable depth limit (default 3), and finite total iteration budget.

III.3 Time-Boxing

```
time_box:
    duration: 3 weeks       // calendar time
    check_in: weekly        // intermediate evaluation cadence
    on_expiry: fold_back    // what happens when time runs out
    partial_results: true   // fold back whatever we have
```

At each check-in: run evaluators, report progress, human decides continue/extend/pivot/terminate. On expiry: run final evaluation, package all outputs as fold-back payload, fold back to parent with status `time_box_expired`.

Time-boxing adds an upper bound on iteration count, not a bypass of evaluation.

III.4 Projection Profiles

Named configurations for common use cases:

Profile	Description	Graph	Evaluators	Convergence
full	Maximum rigour — regulated, audited	All edges	H + A + D on all edges	delta = 0, all checks
standard	Normal feature development	Core edges	A + D on most, H on key	Required checks pass
poc	Time-boxed feasibility	intent -> sketch -> prototype -> validation	A + H (no D — throwaway)	Hypothesis confirmed/rejected or timeout
spike	Time-boxed risk assessment	hypothesis -> experiment -> assessment	A + H	Risk assessed or timeout
hotfix	Emergency production fix	incident -> fix -> verification	D + H (production verified)	Fix verified in production
minimal	Smallest valid projection	1 edge	At least 1 evaluator	Evaluator passes

Profiles compose hierarchically following constraint accumulation:

```
corporate_profile (base)
  -> division_profile (overrides)
    -> team_profile (overrides)
      -> project_profile (overrides)
        -> feature_profile (overrides)
```

At each level, the override rules are: - **Graph**: can only remove edges (narrow), not add (widen) — unless explicitly permitted - **Evaluators**: can add evaluators (stricter), can only remove with explicit `override: true` - **Convergence**: can raise thresholds (stricter), not lower (except with explicit override) - **Context**: can add context (more constraints), can remove with explicit override

This follows the constraint propagation principle (#2): constraints accumulate, they don't evaporate. Loosening requires explicit, auditable override.

Profile selection is itself a Context[] decision:

Factor	Full	Standard	Light/Minimal
Regulatory environment	Regulated, audited, safety-critical	Normal team with CI/CD	Internal tool, experimental
Team maturity	New team, unfamiliar domain	Established practices	Experienced, trusted team
Risk tolerance	Low — failure is expensive	Moderate	High — failure is cheap
Time pressure	Low — quality over speed	Balanced	High — speed over formality
Domain criticality	Safety-critical	Standard business application	Experiment, throwaway

Profiles are defaults, not prisons. Any profile can be adjusted per-project or per-feature via the project binding and feature vector overrides.

The Spec/Design boundary under projection: When projecting the graph dimension, the Spec/Design boundary matters. Projections that skip the Requirements -> Design edge (e.g., spike, hotfix) collapse the boundary — they go from intent directly to tech-bound code. Projections that preserve the boundary (standard, full) maintain the separation between WHAT and HOW, enabling multiple design variants per spec. The boundary is a graph-level property that is present in some projections and absent in others — but when present, it enables the multiple-implementation capability (Principle 12).

Part IV: Ontology Traceability

This section traces every element of the formal system to its parent ontology concept, making the instantiation relationship explicit. The tracing serves two purposes: it validates the formal system against the parent theory (are the four primitives consistent with the ontology's axioms?), and it enables discovery of concepts that should propagate back from the SDLC instantiation to the parent (what did we learn here that generalises?).

IV.1 Concept Mapping

Every element of the formal system traces to a concept in the parent ontology:

Model Element	**Ontology Concept**	**#**
Stable asset	Markov object	7
Asset boundary (interface/schema)	Markov blanket	8
Asset graph (topology)	Constraint manifold	9
Admissible edge	Admissible transformation	5
Iteration function	Local preorder traversal	15
Constructor (builder)	Constructor	41
Spec (intent + lineage)	Encoded representation	40
Context[]	Constraint manifold (local surface)	9
Evaluator set	Evaluator-as-prompter	35
Delta (evaluator output)	Intent / delta	36
Feedback edge	Deviation signal	44
Convergence	Stability condition	7
Feature vector (composite)	Trajectory through constraint manifold	15, 9
REQ key	Vector identifier / lineage tag	44
Module decomposition	Basis vectors of the build subspace	9
Basis projection	Feature projected onto minimal module basis	7, 9
Engine (Layer 1)	Universal constructor	41
Graph Package (Layer 2)	Domain-specific encoded representation	40

Model Element	Ontology Concept	#
Project Binding (Layer 3)	Instance-specific construction context	9
Event (immutable fact)	Unit of change	4
Projection (derived view)	Emergent manifold-level observable	9, 11
Probabilistic compute	Stochastic expansion	45
Deterministic compute	Verification contraction	45
Reflex phase	Autonomic sensing	49
Affect phase	Signal triage / self-regulating	49
Conscious phase	Deliberative / self-directing	49
Running system	Teleodynamic Markov object	49
Homeostasis	Self-maintaining boundary conditions	49
Living specification	Living encoding	46
Hallucination / failure	Probability degeneracy from sparse constraints	54
Context density	Constraint density	16
Graph as Context	Abiogenesis — encoded structure from practice	39
Living system	Concurrent ecology of Markov objects	49
Metabolism (CI/CD loop)	Continuous transformation — energy consumed to maintain	49
Vector spawning (reproduction)	New vectors born from intent events	36, 44
Vector cancellation (apoptosis)	Basis dimension collapses when requirement removed	9
Concurrent vectors (cell cycle)	Many vectors in different phases simultaneously	45
Hamiltonian	Cost dynamics on constraint manifolds	77
Feature vector	Composite trajectory	78
Zoomable graph	Zoomable constraint structure	79

IV.2 The IntentEngine and Ontology Concepts

The IntentEngine maps to three ontology concepts simultaneously:

Ontology concept	How IntentEngine instantiates it
#49 Teleodynamic self-maintenance	Fractal self-repair: each IntentEngine detects deviation (observer) and corrects (evaluator -> output). At every scale, the system senses and responds. The chain of IntentEngines is the mechanism by which the system maintains itself.
#23 Scale-dependent observation	Consciousness-as-relative: the same observation at different scales produces different processing. A test failure is a reflex at the code level but may escalate to conscious review at the feature level. The IntentEngine is the unit of scale-dependent observation.
#9 Constraint manifold	Ambiguity as constraint density measure: zero ambiguity means the constraint surface fully determines the response (reflex). Nonzero ambiguity means the constraint surface is insufficient — more constraints needed. Persistent ambiguity means the constraint manifold itself needs modification (spec update).

The event sourcing model connects to the ontology's unit of change (#4):

Domain	Event stream	Projection
Physics	Constraint network evolution	Spacetime, particles, gravity
LLM	Forward pass activations	Token probabilities, output text
SDLC	Methodology events	STATUS, feature vectors, task lists
Biology	DNA replication + transcription	Phenotype, organism behaviour

IV.3 The Construction Pattern

The methodology is a direct instantiation of concept #38:

Pattern Element	In the Ontology	In the SDLC
Encoded representation	DNA, constitution, prompt	Spec + Context[] (incl. graph topology)

Pattern Element	In the Ontology	In the SDLC
Constructor	Ribosome, institution, attention mechanism	iterate() — LLM agent, human, compiler
Constructed structure	Protein, law, output	Stable vector component (Markov object)
Selection pressure	Predators, regulators, environment	Evaluators (Human, Agent, Tests)
Deviation signal	Mutation, amendment, feedback	Evaluator delta -> Spec update

IV.4 Abiogenesis at Two Levels

The methodology follows the abiogenesis pattern (#39) at two levels:

Level 1: Graph topology construction 1. Practitioners work in a domain (software, law, science) 2. Patterns crystallise: "we always need X before Y" 3. The graph topology is encoded 4. The graph now drives construction 5. The graph evolves from runtime experience

Level 2: Feature construction within the graph 1. Business need, technical limitation, user pain 2. Developers experiment, building against the graph 3. Specifications, test suites, architecture crystallise from practice 4. Formal specifications direct the build; AI agents read specs to construct components 5. Runtime telemetry updates the specification; the system becomes self-maintaining (#49)

The bottom-up arc of this paper — practice -> formalisation, not axioms -> implementation — is itself a Level 1 abiogenesis instance. The methodology was not derived from the ontology; it was built from practice, and the ontology emerged as the explanation.

Graph topology as living encoding: The graph topology is itself subject to the gradient (Principle 14). When TELEM signals reveal that the graph is under-specified (too many failures at a particular edge), the graph gains new edges or new evaluators. When TELEM signals reveal that the graph is over-specified (unnecessary intermediate waypoints adding cost without quality benefit), edges can be collapsed. The graph is a living encoding of domain expertise — it evolves from experience, just as a biological specification evolves from selection pressure.

Cross-domain abiogenesis prediction: If the four primitives are universal, then independent teams in different domains should discover structurally similar graphs for similar domains. The SDLC graph with its Spec/Design boundary, iterative construction, and feedback loop should be recognisable in any domain that involves specification -> implementation -> validation. The specific nodes will differ (requirements vs hypotheses vs legal arguments), but the topology — the pattern of when intermediate waypoints are needed, where feedback edges close, and how zoom levels stratify — should converge. This is a testable prediction of the framework.

IV.5 What Propagated Back to the Parent

Three concepts discovered in the SDLC instantiation that generalise to the parent ontology:

#	Concept	What was discovered	Where it generalises
77	**Cost dynamics on constraint manifolds**	H = T(iteration_cost) + V(constraint_delta). Constraint density is the metric: dense = cheap convergence, sparse = expensive. Connects gravity, hallucination, and failure prediction	Any constraint manifold with traversal cost
78	**Composite trajectory**	Complex Markov objects as composites of sub-objects produced along trajectories through the constraint manifold, carrying intent lineage	Not just hierarchical composition but sequential construction through multiple constraint edges
79	**Zoomable constraint structure**	Any transition expandable into a sub-graph, any sub-graph collapsible into a single edge. Granularity is a choice	The hierarchy of constraint resolution (#11) made operational

These concepts now live in the parent ontology's concept index (Section X-A) as SDLC back-propagation.

Part V: Worked Example — PoC Spawning

To make projections concrete, here is a PoC scenario demonstrating invariant preservation, spawning, fold-back, and context enrichment.

V.1 Situation

A team is building a data pipeline (Feature F, standard profile). During the design edge, the agent evaluator detects a gap: "Can Apache Kafka handle our throughput requirements, or should we use a simpler queue?"

V.2 Spawning

```
Parent: Feature F (standard profile)
  Edge: requirements -> design
  Gap detected: "Kafka vs simple queue for throughput"
```

```
  Spawns: PoC P (poc profile)
    question: "Can Kafka handle 10K events/sec with our schema?"
    time_box: 3 weeks
    check_in: weekly
```

Parent status: `blocked_on: PoC-P`

V.3 PoC Execution

Week 1 — `iterate(|hypothesis> -> |design_sketch>)` — sketch Kafka topology. Agent evaluator: design sketch addresses throughput question? PASS. Promote to design_sketch.

Week 2 — `iterate(|design_sketch> -> |prototype>)` — build minimal Kafka producer/consumer. Agent evaluator: prototype implements the design? PASS (throwaway code, no production standards). Promote to prototype.

Week 3 — `iterate(|prototype> -> |validation>)` — run load test. Deterministic evaluator: 10K events/sec sustained? PASS (12K actual). Human evaluator: "Yes, Kafka is viable. Proceed." **Converged**: hypothesis confirmed.

V.4 Fold-Back

```
fold_back(PoC-P -> Feature F):
  outputs:
    - findings: "Kafka handles 12K events/sec, 20% headroom"
    - recommendation: "Use Kafka"
    - ADR: "ADR-007: Use Kafka for event streaming (validated by PoC-
P)"
    - promotable_assets: [design_sketch]

  parent_context_update:
    - Context[] += ADR-007
    - Context[] += PoC findings document
    - design_sketch promoted to Feature F design trajectory (to be
refined)
```

Feature F resumes at the design edge, now with Kafka validated as a constraint. The possibility space has narrowed — no more "Kafka vs queue?" uncertainty.

V.5 What Was Preserved

All four invariants held throughout:

- **Graph**: PoC had a graph (hypothesis -> design_sketch -> prototype -> validation)
- **Iterate**: Each edge iterated until evaluators converged
- **Evaluators**: Agent + human + deterministic (load test) evaluated each edge
- **Spec + Context**: PoC had a hypothesis (intent) and technical constraints (context)

Traceability: PoC-P traces to Feature F's gap detection, which traces to the original intent. The causal chain is intact.

V.6 Formal Analysis

The PoC demonstrates:

1. **Projection switch**: Feature F uses standard profile; PoC P uses poc profile. Both preserve the invariants. The same engine (Layer 1) serves both; only the encoding differs.
2. **Vector spawning**: Parent vector creates child with its own graph, evaluators, and time box. The parent's status transitions to `blocked_on: PoC-P` — it cannot proceed until the child resolves.
3. **Fold-back as context enrichment**: Child outputs narrow the parent's constraint surface (#16), driving convergence. Before fold-back: Context[] = {intent, ADRs, models} — wide possibility space. After: Context[] = {intent, ADRs, models, ADR-007, PoC findings} — narrower, more constrained.
4. **Functor rendering**: The PoC's encoding is mostly F_P (agent-driven, throwaway code) with F_D only for the load test. The feature's encoding is F_P + F_D + F_H. Different functors from the same engine.
5. **Time-boxing**: Had the PoC timed out in week 2, partial results (design_sketch, preliminary prototype) would still fold back as Context[]. The time_box_expired convergence mode does not violate the Iterate invariant — the iteration loop still ran, evaluators still fired, the stopping condition was extended not removed.
6. **Hamiltonian dynamics**: Before the PoC, the design edge had high V (constraint_delta) because "Kafka vs queue?" was an unresolved dimension. After fold-back, V drops — the constraint is resolved, and the edge converges more cheaply.

V.7 Alternative Scenario: Time-Box Expiry

Had the PoC not converged by week 3 — suppose the load test showed only 6K events/sec, below the 10K requirement — the scenario would unfold differently:

Week 3 check-in: Deterministic evaluator: 10K events/sec sustained? FAIL (6K actual). Human evaluator: "The prototype is promising but not there yet. Should we extend?"

Option A — Extend: Time box extended by 1 week. Human provides direction: "Try tuning partition count and batch size." The PoC continues iterating, now with enriched context (the 6K data point is itself a finding).

Option B — Fold back with partial results:

```
fold_back(PoC-P -> Feature F):
  outputs:
    - findings: "Kafka achieved 6K/sec, 40% below requirement. May
work with tuning."
    - recommendation: "Inconclusive — consider simpler queue or deeper
Kafka investigation"
    - promotable_assets: none (prototype not validated)

  parent_context_update:
    - Context[] += partial findings (6K data point, Kafka
configuration explored)
    - No ADR created — the question is not answered
```

Feature F resumes with enriched context but the possibility space is still wide. The parent vector might spawn a second Discovery vector: “What simpler queue alternatives exist?” or escalate to human judgment: “Do we reduce the throughput requirement?”

This illustrates time-boxing’s value: even incomplete results fold back as context. The partial finding narrows the space (we now know Kafka’s baseline), and the parent vector makes a better-informed decision than before the PoC was spawned.

Part VI: Relationship to Prior Approaches

VI.1 What Changes from Sequential Models

Prior Model	Asset Graph Model
Sequential stages (waterfall, V-model, or iterative variants)	Directed cyclic graph of asset types with admissible transitions
Features travel through a pipeline	Features are composite vectors — trajectories through the graph
Stage-specific agents or roles	Universal iteration function, parameterised per edge
Linear or iterative pipeline	Cyclic graph with feedback edges (telemetry -> intent)
Fixed process topology	Graph is domain-constructed, zoomable, lives in Context[]
Stops at deployment or UAT	Full lifecycle: CI/CD -> Telemetry -> Homeostasis -> New Intent
Requirements as documents	Spec as living encoding, continuously updated from runtime
Tasks planned top-down	Tasks emerge from feature decomposition and dependency compression
Traceability as documentation	Traceability as composite vector coherence (REQ key = vector ID)

VI.2 What is Preserved

The formal system preserves everything that prior models got right, while providing a deeper explanation of why:

Prior concept	Preserved as	Deeper explanation
Requirement traceability	Intent lineage in each asset, REQ key as vector ID	The composite vector carries the full causal chain; traceability is emergent from the four invariants
TDD workflow	The iteration pattern at code <-> unit_tests	iterate() with deterministic evaluators — the same operation as any other edge
BDD scenarios	The iteration pattern at design -> uat_tests	Same iterate() with human + agent evaluators — a different encoding of the same spec
Two compute regimes	Probabilistic + Deterministic, now formalised as functor categories F_P and F_D	The LLM proposes (expansion), tests verify (contraction) — the specification is the fitness landscape
Feedback loops	First-class graph edges (telemetry -> intent)	The gradient at production scale — `delta(running_system, spec) -> correction`
Key Principles	Constraints within Context[]	The constraint surface is parameterisable, not hardcoded
Agile iterations	The convergence loop at every edge	iterate() generalises "sprint" to any edge, any scale, any convergence criterion

VI.3 What is Added

The formal system adds structural vocabulary that makes implicit engineering knowledge explicit:

Structural additions: - **Formal ontology grounding** — every element traceable to constraint-emergence concepts, enabling cross-domain reasoning - **Composite vectors** — features as trajectories through the graph, not pipeline travellers. The composite carries the full causal chain - **Zoomable graph** — granularity as a choice at every edge. Selective zoom is the common case: collapse some intermediates, retain others - **Scale-dependent assurance** — same evaluator pattern at every scale (unit test:module :: UAT:feature :: telemetry:product) - **Three-layer separation** — Engine / Graph Package / Project Binding. Explains why the same methodology produces different outcomes in

different domains - **Spec/Design boundary** — WHAT (tech-agnostic) vs HOW (tech-bound), enabling multiple implementations per requirement and technology migration without spec change

Formal additions: - **Functor renderings** — same spec, multiple encodings, natural transformations between them. Projections as functors. Zoom as encoding operation - **Hilbert space formalism** — inner product determines parallelism, Hamiltonian determines cost, basis change tracks spec evolution. The decorative question: does anything structural depend on the Hilbert space? Answer: yes — the inner product (shared modules) determines which features can run in parallel, and the Hamiltonian (cost dynamics) connects constraint density to convergence cost - **IntentEngine as composition law** — fractal observer/evaluator at every scale, consciousness-as-relative, three-way ambiguity partition

Lifecycle additions: - **Event sourcing execution** — immutable events, all state as derived projections, full auditability - **Living system** — concurrent vector lifecycles + metabolism + perception + self-modification + pruning - **Projection profiles** — named configurations for valid lighter instances (full -> minimal) - **Vector type diversity** — discovery, spike, PoC, hotfix — each a different projection with different convergence criteria - **Continuous sensing** — interoceptive and exteroceptive monitors run independently of iterate(), enabling the system to notice threats and decay even when no one is actively developing - **Protocol enforcement** — hooks verify iterate() side effects before allowing the agent to stop, ensuring observability is never sacrificed for generation speed

---

Part VII: Reference Implementation

The formal system is not merely theoretical. A reference implementation on Claude Code demonstrates that the four primitives, four invariants, and composition law are sufficient to produce a working software construction methodology — and that the LLM itself can serve as the construction apparatus when the constraint surface is sufficiently specified.

VII.1 Architecture: Markdown as Code, LLM as Runtime

The reference implementation delivers the methodology as a **Claude Code plugin** — not a traditional software system with compiled code, but a set of markdown agent definitions, YAML configurations, and slash commands that the LLM executes directly.

```
Plugin Package (.claude-plugin/plugins/aisdlc-methodology/v2/)
│
├── agents/
│   └── aisdlc-iterate.md          ← The ONE agent (universal iterate)
│
├── commands/
│   ├── aisdlc-start.md            ← State-driven routing (8 states)
│   ├── aisdlc-status.md           ← Project-wide observability
│   ├── aisdlc-iterate.md          ← Direct iterate invocation
│   ├── aisdlc-init.md             ← Project scaffolding (Layer 3)
│   ├── aisdlc-checkpoint.md       ← Session recovery
│   ├── aisdlc-review.md           ← Spec-scale gradient
```

```
│   ├── aisdlc-gap-analysis.md     ← Cross-edge gap aggregation
│   ├── aisdlc-release.md          ← Release packaging
│   ├── aisdlc-spawn.md            ← Vector spawning
│   └── aisdlc-foldback.md         ← Spawn fold-back
│
└── config/
    ├── graph_topology.yml          ← Asset types + admissible
transitions
    ├── edge_params/*.yml           ← 10 edge parameterisations
    ├── evaluator_defaults.yml      ← Evaluator taxonomy per edge
    ├── profiles/*.yml              ← 6 named projection profiles
    └── feature_vector_template.yml ← Vector lifecycle tracking
```

The design choice is itself a validation of the formal system. The iterate agent reads edge parameterisation at runtime — it never hard-codes domain knowledge. Different edges produce different behaviour from the **same agent**, because the constraint surface (edge config + context + evaluators) is different. The agent IS the iterate() function (#45).

VII.2 Three-Layer Instantiation

The three-layer architecture (Principle 12, Part II §II.1) maps directly to the implementation:

Formal Layer	Implementation	Ships As
Layer 1: Engine (universal)	`agents/aisdlc-iterate.md` + evaluator types + event sourcing primitives + `commands/*.md`	Plugin package (immutable per version)
Layer 2: Graph Package (domain-specific)	`config/graph_topology.yml` + 10 edge parameterisations + constraint dimensions + projection profiles	Plugin config (forkable for other domains)
Layer 3: Project Binding (instance-specific)	`.ai-workspace/context/` — `project_constraints.yml`, ADRs, data models, policy, standards	Scaffolded per project by `/aisdlc-init`

Upgrading the plugin (Layers 1+2) does not overwrite project bindings (Layer 3). Different graph packages can be created by forking the `config/` directory. This separation is not an implementation convenience — it is the formal system's functor structure (§II.2) made concrete: the same engine, applied to a different graph package, generates a different methodology.

VII.3 The LLM as Constructor

The most consequential design decision: **the runtime is the LLM itself**. There is no orchestration framework, no state machine, no compiled code managing transitions. The Claude Code LLM reads the markdown specifications and YAML configurations, then

executes the methodology by following the constraint surface.

This validates a central claim of the formal system: the four primitives are substrate-neutral (#9). The constructor can be a human following a process document, a traditional software system executing compiled code, or an LLM executing markdown specifications. What matters is that the constraint surface (Spec + Context) is sufficiently specified to bound the constructor's behaviour (#11).

The two-command UX layer demonstrates progressive disclosure: `/aisdlc-start` detects project state and routes to the appropriate edge (8 states handled), while `/aisdlc-status` provides project-wide observability. The full command set is available but rarely needed — most development sessions use only these two entry points.

VII.4 Sibling Implementations

The formal system predicts that any platform capable of the four primitives can instantiate the methodology. Two sibling implementations test this prediction:

Platform	Engine Binding	Status
Claude Code	Markdown agent + YAML config, LLM-as-runtime	Complete, dogfooded
Gemini CLI	Universal orchestration routine, same workspace contract	Design complete
Codex	Tool-calling runtime (`exec_command`, `apply_patch`), same event log	Design complete

All three platforms bind to **identical platform-agnostic specification** (64 requirements across 13 categories). The specification defines what; each platform instantiates how. This is the functor structure in practice: three different encodings of the same formal system, connected by natural transformations (the shared event log contract and workspace schema).

VII.5 Dogfooding Results

The reference implementation was dogfooded on a real project (Python CLI bookmark manager, 2026-02-20). Three edges were traversed: intent→requirements (1 iteration), requirements→design (1 iteration), design→code (2 iterations). All converged successfully.

What the dogfooding validated:

- **LLM-as-runtime is viable** — no custom code needed; markdown + YAML + edge configs sufficient for the LLM to execute the methodology
- **Checklist composition works** — edge defaults + project constraints + feature overrides resolve cleanly at runtime

- **REQ key discipline is automatic** — the template produces consistent `REQ-{TYPE}-{DOMAIN}-{SEQ}` keys without manual enforcement
- **Iteration reports are legible** — structured checklist tables show exactly what passes and fails at each convergence check

What the dogfooding revealed (6 bugs, all fixable, none structural):

Bug	Severity	Finding
Feature vector create-if-absent not specified	HIGH	Blocks first iteration for new features
`$architecture.*` variable prefix undefined	HIGH	Resolution rules incomplete
`no_ambiguous_language` criterion self-contradictory	MEDIUM	"May" banned then required in same checklist
`docs/requirements/` path inconsistency	MEDIUM	v1.x holdover in v2 configs
Duplicate checklist items (`compiles_or_parses`, `lint_passes`)	LOW	Redundant evaluator criteria
Misnumbered steps in init command	COSMETIC	Documentation drift

Critical empirical finding: 5 of 7 build-time bugs in the tested project traced to **implicit constraint dimensions** — the design edge left ecosystem, deployment, or build dimensions unresolved, and those gaps cascaded downstream. This directly validates **Prediction 2** (missing constraint dimensions predict build failures) and motivated the mandatory dimension verification in the design edge parameterisation.

VII.6 What the Implementation Demonstrates

The reference implementation is not just a tool — it is an empirical test of the formal system's claims:

Formal claim	Implementation evidence
Four primitives suffice	One agent + graph config + evaluator config + context directory — nothing else needed
IntentEngine is composition, not primitive	The iterate agent composes the four primitives; it has no independent logic
Constraint density predicts convergence cost	Missing dimensions caused predictable downstream failures

Formal claim	Implementation evidence
Projections preserve invariants	Spike and PoC profiles skip edges but retain all four primitives
Event sourcing enables self-observation	TELEM signals derived from event log; STATUS.md is a projection
The graph is domain-constructed	Forking `config/` produces a different methodology; engine unchanged
Functor renderings are real	Three platforms, same spec, different encodings, shared event contract

The formal system generated the implementation. The implementation tested the formal system. The bugs found were in parameterisation (Layer 2 and 3), never in the primitives (Layer 1). This is the pattern the formal system predicts: the engine is stable; emergence varies.

Part VIII: Research Directions

VIII.1 Cross-Domain Graph Instantiation

The four primitives claim domain independence. Testing this requires instantiating the engine with non-SDLC graph packages:

Legal document production: Intent -> Research -> Brief -> Review -> Filing.

SDLC concept	Legal instantiation
Asset types	Legal research, draft brief, peer review, filing
Evaluators	Attorney review (Human), citation checker (Deterministic), AI analysis (Agent)
Spec/Design boundary	Legal argument (what) vs jurisdictional binding (how)
Constraint dimensions	Jurisdiction, precedent scope, enforcement mechanism, client objectives
Graph abiogenesis	Legal practitioners evolved this sequence over centuries

Scientific paper pipeline: Hypothesis -> Literature Review -> Experiment -> Analysis -> Draft -> Peer Review -> Revision.

SDLC concept	Scientific instantiation
Feedback edge	Peer review -> revision (same as telemetry -> intent)
PoC vector	Preliminary experiment before full study
Feature view	Per-hypothesis tracking across all artefacts
Living spec	Hypothesis updates from experimental results

Organisational policy: Need -> Analysis -> Proposal -> Consultation -> Implementation -> Monitoring.

SDLC concept	Policy instantiation
Evaluators	Stakeholder review (Human), compliance check (Deterministic), impact analysis (Agent)
Three processing phases	Monitoring (reflex), impact assessment (affect), policy revision (conscious)
Homeostasis	Policy in operation, monitored for effectiveness

Each successful instantiation strengthens the claim that the engine is universal. Each failure reveals constraints the engine did not capture. The prediction: independent teams in different domains, given the same engine, will converge toward structurally similar mandatory waypoints — because the constraint topology of each domain has natural cleavage points that any iterative process discovers.

VIII.2 Formal Verification of Invariant Preservation

The projection function claims to preserve invariants. This is a verifiable claim:

- **Formal specification** of the four invariants in a proof-checkable language (Lean, Coq, or Agda). The invariant preservation conditions (Part II, §II.3) are precise enough to formalise.
- **Verification** that each named profile (full, standard, poc, spike, hotfix, minimal) satisfies the invariant preservation conditions. This is a finite check — six profiles against four conditions.
- **Automated checking** that a given projection configuration produces a valid instance. A validator that takes a projection spec (graph subset, evaluator profile, convergence criteria, context) and checks `valid(P)` against the invariant conditions.
- **Composition verification** — that the hierarchical composition of profiles (corporate -> division -> team -> project -> feature) preserves invariants at every level. The constraint accumulation principle (constraints can be added but only removed with explicit override) should guarantee this, but the guarantee should be formally verified.

This connects to the ontology's Constraint category (concept #58): the asset graph and its projections should form a category whose morphisms are invariant-preserving projection functions. The objects are valid projections; the morphisms are projection transformations that preserve the four invariants. If this category can be exhibited, the formal system gains the full machinery of categorical reasoning — functors between graph packages, natural transformations between encodings, limits and colimits for composition.

VIII.3 IntentEngine Generalisation

The IntentEngine claims to be a universal processing unit — observer/evaluator at every scale. Testing this requires:

- **Formal characterisation** of the ambiguity classification (zero, bounded, persistent) — is this partition exhaustive? Are there edge cases? A candidate edge case: ambiguity that oscillates between bounded and persistent across iterations. The current model treats each IntentEngine invocation independently; does cumulative ambiguity history affect the classification?
- **Consciousness-as-relative verification** — does Level N's escalate always become Level N+1's reflex? What happens at the top level (no further escalation possible)? In the SDLC instantiation, the top level is human judgment with no further escalation — the human either resolves the ambiguity or declares the constraint surface inadequate. In an autonomous system, the top level might be a safety boundary (shut down).
- **Scaling properties** — does the IntentEngine chain scale linearly, or does affect propagation introduce superlinear complexity? The affect phase is the critical bottleneck: if affect processing is O(n) in the number of signals, the chain scales linearly. If affect requires cross-signal correlation (this CVE + that dependency + that API change = compound risk), scaling may be superlinear.
- **Cross-domain IntentEngine** — does the observer/evaluator/typed_output pattern appear in non-SDLC domains? In biological systems, the reflex arc is the same structure. In legal proceedings, the judge observes, evaluates ambiguity, and routes to one of three outcomes (dismiss, continue trial, escalate to higher court). The claim is that the three-way partition is a structural universal, not an SDLC artefact.

VIII.4 Empirical Validation

The formal system makes empirical predictions that can be tested against event logs from methodology instantiations:

Prediction 1: Constraint density predicts convergence cost. Edges with sparse Context[] should require more iterations and produce more evaluator failures than edges with dense Context[]. Measurable from event logs: correlate context_hash diversity per edge with iteration count and evaluator failure rate. The Hamiltonian predicts: V(constraint_delta) decreases as context density increases, reducing total H and thus iteration count.

Prediction 2: Missing constraint dimensions predict build failures. When design omits a constraint dimension (ecosystem compatibility, build system), the omission should manifest as failures at the code or integration edge. The event log should show the pattern: low iteration count at design (evaluator bar appeared adequate) followed by high iteration count at code. Early dogfooding supports this: 5/7 build-time bugs traced to implicit design dimensions.

Prediction 3: Evaluator bar is the primary quality lever. Cross-project comparison of evaluator configurations and build outcomes should show that evaluator composition per edge predicts product quality better than team size, timeline, or technology choice. The mechanism: evaluators are the convergence criterion — if the bar is low, iteration stops too early; if the bar is high, iteration catches more defects.

Prediction 4: Graph topology convergence. Independent teams in the same domain, given the same engine and the freedom to discover their graph, should converge toward structurally similar topologies — the same mandatory waypoints emerging from practice. This is the abiogenesis prediction: constraint topologies have natural cleavage points that independent discovery processes find. Falsifying this would mean graph topology is path-dependent (sensitive to the team's starting conditions), not convergent.

Prediction 5: Affect triage predicts conscious processing load. Teams that implement the affect phase (signal classification and escalation thresholds) should have lower conscious processing overhead (fewer human reviews per edge) than teams that route all signals directly to conscious review. The mechanism: affect filters noise, preventing consciousness from being overwhelmed.

Prediction 6: Living system properties correlate with system longevity. Systems that achieve all six living properties (continuous sensing, concurrent vectors, metabolism, perception, self-modification, pruning) should demonstrate longer operational life, faster response to incidents, and lower maintenance cost than systems missing one or more properties. Measurable from production telemetry and maintenance event logs.

Testing methodology: All six predictions can be tested against event logs from methodology instantiations. The event sourcing execution model (Principle 16) makes this possible — every methodology action produces an immutable event, and all observables are derived projections. A research programme would:

1. Collect event logs from multiple instantiations across different teams, domains, and projection profiles
2. Compute per-edge metrics: iteration count, evaluator failure rate, convergence time, context density (hash diversity)
3. Correlate metrics with outcomes: build success rate, time to production, incident frequency, maintenance cost
4. Test each prediction as a statistical hypothesis across the dataset

The formal system generates these predictions; the event sourcing model makes them testable; the projection profiles provide natural experimental conditions (different teams using different profiles for similar work). This is a research programme, not a single experiment.

The dogfooding chain: The methodology has been instantiated once (the AI SDLC toolchain itself). This provides the first data point for each prediction. The formal system predicts that subsequent instantiations — by different teams, in different domains, using different technologies — will observe the same patterns: constraint density predicting convergence cost, missing dimensions predicting failures, affect triage predicting cognitive load. Each independent replication strengthens or weakens the predictions.

Open question: meta-stability: The formal system claims that the four primitives are stable — no fifth is needed. But is this claim itself stable? Could a domain instantiation reveal a pattern that requires a fifth primitive? The framework would need to absorb the

discovery (modify the engine layer), which would violate the three-layer architecture's claim that the engine never changes. This is the deepest falsification target: not any specific prediction, but the sufficiency of the four primitives themselves.

VIII.5 What Would Falsify This Formal System

Falsifying observation	What it would mean
A valid methodology that violates one of the four invariants	The invariants are too strong — the formal system over-constrains
A domain that cannot be instantiated with the four primitives	The engine is not universal — the primitives are SDLC-specific
An IntentEngine invocation that produces an output not in {reflex.log, specEventLog, escalate}	The three-way partition is not exhaustive — ambiguity has more than three regimes
Constraint density failing to predict convergence cost across domains	The Hamiltonian formalism is decorative, not structural
Independent teams producing radically different graph topologies for the same domain	Graph abiogenesis is path-dependent, not convergent — the "natural waypoints" claim is false
A living system (all six properties active) that fails to self-maintain	The teleodynamic analogy breaks — the gradient at all scales is not sufficient for self-maintenance
An edge that converges without any evaluator	The evaluator invariant is too strong — some edges self-converge
A useful projection that cannot be expressed as Functor(Spec, Encoding)	The functor formalism is too restrictive — valid methodologies exist outside the composition law

The strongest claims — and therefore the most important to test — are: (a) the four primitives are necessary and sufficient, (b) the IntentEngine three-way partition is exhaustive, and (c) constraint density predicts convergence cost. These are structural claims about the formal system itself, not claims about any particular instantiation.

VIII.6 Connections to Existing Research

The formal system connects to several established research programmes:

Formal methods and software verification: The invariant preservation conditions (Part II, §II.3) are in the spirit of formal methods — specifying what must be true of any valid instance. The difference: traditional formal methods verify code against specifications; this formal system verifies *methodology instances* against *methodology invariants*. The four invariants are process-level, not code-level.

Complex adaptive systems: The living system (Principle 17) exhibits properties studied in complexity science — concurrent agents (feature vectors), emergent behaviour (the gradient at all scales), self-organisation (graph abiogenesis), adaptation (spec evolution). The formal system provides a precise vocabulary for these properties in the software domain: the four primitives, the IntentEngine composition law, the Hamiltonian cost dynamics.

Organismic analogy in software: The biological correspondences in Principle 17 are not original — software systems have been compared to organisms before (Lehman's laws of software evolution, Conway's law). What the formal system adds is a *structural mechanism*: the gradient `delta(state, constraints) -> work` operating at every scale, the IntentEngine as the universal processing unit, and the Markov boundary as the concurrency enabler. These are not metaphors; they are structural correspondences that make specific predictions.

Constructor theory (Deutsch and Marletto): The three-layer architecture (Engine / Graph Package / Project Binding) parallels constructor theory's distinction between constructors (universal) and tasks (specific). The engine is a universal constructor; the graph package specifies which tasks are admissible; the project binding provides the input state. The formal system diverges from constructor theory in its emphasis on constraint (what bounds the constructor) rather than capability (what the constructor can do).

VIII.7 The Paper as Self-Reference

This paper is itself a Layer 2 artefact — a Graph Package encoding domain expertise. The SDLC domain has been traversed by practitioners (the author and collaborators), patterns have crystallised, and the graph topology and edge configurations have been encoded as this document. The paper follows the abiogenesis pattern it describes:

1. **Constraint**: The need to formalise software construction methodology
2. **Constructor**: Practitioners building software, observing patterns, experimenting with process
3. **Encoding emerges**: This paper — the patterns crystallised into formal structure
4. **Encoding drives constructor**: AI agents and human developers loading this document as Context[] and building according to its constraints
5. **Encoding evolves**: Runtime experience (dogfooding, multiple instantiations) reveals gaps, and the paper updates

The spec/design boundary applies: this paper is spec (WHAT the methodology is — tech-agnostic). Any particular implementation (Claude-based toolchain, Gemini-based toolchain, manual process) is design (HOW — tech-bound). The same paper can have many implementations, just as the same REQ can have many designs.

Applicability scope: This formal system is applicable wherever the following conditions hold: (a) work produces typed artefacts, (b) artefacts have quality criteria, (c) quality can be evaluated, and (d) artefacts are constructed iteratively. These conditions are not specific to software — they describe any domain with iterative construction and quality evaluation. The four primitives are the minimal vocabulary for describing such domains. The SDLC graph is one instantiation; this paper documents the formal system that generates all valid instantiations.

References

- **Constraint-Emergence Ontology** — github.com/foolishimp/constraint_emergence_ontology — parent theory. Concept numbers (#N) throughout this document refer to the canonical concept index in that repository.
- **Emergent Reasoning** — github.com/foolishimp/emergent_reasoning — formal companion: LLMs as constraint-manifold traversal systems.
- **AI SDLC Asset Graph Model v2.8** — github.com/foolishimp/ai_sdlc_method — full specification (source document for this paper).
- **Projections and Invariants v1.1** — github.com/foolishimp/ai_sdlc_method — projection profiles, vector types, spawning, fold-back (source document for this paper).

Appendix: Constraint Tolerances

The gradient (Principle 14) is `delta(state, constraints) -> work`. But delta is undefined unless constraints have **measurable thresholds** — tolerances. A constraint without a tolerance is a wish; a constraint with a tolerance is a sensor.

```
Constraint: "the system must be fast"        -> unmeasurable, delta
undefined
Constraint: "P99 latency < 200ms"            -> measurable, delta =
|observed - 200ms|
Constraint: "all tests pass"                 -> measurable, delta =
failing_count
Constraint: "design uses protocol X"         -> measurable, delta =
drift from X's properties
```

Tolerances are what make the sensory system (Principle 9) operational. Without them, monitors have nothing to measure against. With them, every monitor becomes an IntentEngine invocation:

```
monitor observes metric ->
  evaluator compares metric to tolerance ->
    within bounds:  reflex.log (system healthy at this point)
    drifting:       specEventLog (optimisation intent deferred)
    breached:       escalate (corrective intent raised)
```

Tolerances exist at every scale:

Scale	Constraint	Tolerance	Breach signal
Single iteration	Edge evaluator	Pass/fail threshold, max iterations	Non-converging iteration count
Edge convergence	All evaluators	Convergence within N iterations	Stuck threshold exceeded

Scale	Constraint	Tolerance	Breach signal
Feature traversal	Graph topology	Feature completes within profile bounds	Stalled feature vector
Production	SLAs, health checks	Latency, error rate, uptime	Operational degradation
Spec review	Spec adequacy	Constraint coverage, consistency	Gaps between spec and reality
Design bindings	Technology choices	Performance, cost, fitness tolerances	Ecosystem drift

Bidirectional pressure on graph topology: Tolerances create a feedback signal that balances graph discovery:

- **Escalation pressure**: Persistent ambiguity -> the graph needs more edges (Principle 10 — graph discovery)
- **Tolerance pressure**: Complexity breaches -> the graph needs fewer or simpler edges
- **Equilibrium**: The optimal graph topology for the current constraints

This is the homeostatic principle applied to the graph itself: the graph is not prescribed, not discovered once, but continuously maintained at equilibrium between expressive power and operational cost.

Design-level tolerances are critical: When an implementation chooses a protocol, a runtime, or a service model, that choice implies performance, cost, and fitness bounds. When those bounds are breached — because the project scaled, the ecosystem shifted, or better options emerged — the breach is a delta that produces optimisation intent. The design evaluates its own fitness.

Without tolerances, there is no homeostasis. The gradient requires measurable delta. The IntentEngine requires classifiable observations. The sensory system requires thresholds. Tolerances are not optional metadata on constraints — they are the mechanism by which constraints become operational. A methodology without tolerances is a methodology without intent generation — it can execute prescribed work but cannot sense drift, propose corrections, or evolve.

Appendix B: Summary — The Formal System in One Page

The AI SDLC is:

1. An **asset graph** — a directed cyclic graph of typed assets with admissible transitions (zoomable)
2. With a **universal iteration function** — the only operation, converging each edge until evaluators pass

3. Traced by **feature vectors** — composite vectors (trajectories through the graph), identified by REQ keys
4. Bounded by **Spec + Context[]** — the constraint surface that prevents degeneracy
5. Evaluated by **{Human, Agent, Tests}** — composable convergence criteria per edge
6. Operating in **three processing phases** — reflex (sensing), affect (triage), conscious (judgment)
7. Split at a **Spec/Design boundary** — WHAT vs HOW, one spec many designs
8. Observable via **feature views** — REQ keys grepped across all artefacts
9. Completing the **full lifecycle** — through CI/CD, Telemetry, Homeostasis, back to Intent
10. Packaged in **three layers** — Engine / Graph Package / Project Binding
11. Executed via **event sourcing** — immutable events, all state as derived projections
12. **Self-observing** — TELEM signals feed back into graph package evolution
13. **Protocol-enforced** — hooks verify iterate() side effects before allowing the agent to stop
14. **Gradient at every scale** — `delta(state, constraints) -> work` from single iteration through spec review
15. **Continuously sensing** — interoceptive + exteroceptive monitors run independently of iterate()
16. **Fractally composed** — the IntentEngine on every edge, at every scale, with consciousness-as-relative
17. **Alive** — when operational, the gradient at all scales simultaneously produces a self-maintaining ecology of Markov objects

Four primitives. One operation. Three layers. The rest is parameterisation — including the graph itself.