AI SDLC: The Asset Graph Model

Version: 2.2.0 **Date**: 2026-02-20 **Foundation**: Constraint-Emergence Ontology (§V, §VIII-B, §VIII-C)

---

1. Overview

The AI SDLC is an instance of the information-driven construction pattern (ontology concept #38): **encoded representation → constructor → constructed structure**. This document formalises the methodology as an **asset graph** with a **universal iteration function**.

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 — is parameterisation of these four primitives for specific graph edges.

The graph is not universal. The SDLC graph (Intent → Requirements → Design → Code → Tests → …) is one domain-specific instantiation. A legal document, a physics paper, an organisational policy each have different graphs. The graph topology is itself constructed — it is Context[], not a law of nature. The four primitives are universal; the graph is parameterised.

---

2. The Asset Graph

2.1 Asset Types and Transitions

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 edge is the vector — the iterative process of assurance, disambiguation, discovery, and solutioning that evolves one asset into the next.

A **delivered feature** is the composite of all assets produced along its edges. Software feature delivery is a composite vector:

```
Feature F = |req⟩ + |design⟩ + |code⟩ + |tests⟩ + |uat⟩ + |telemetry⟩
```

The SDLC asset graph:

```
                          ┌──→ UAT Tests ──────────────────────────┐
                          │                                         │
Intent → Requirements → Design ──→ Code → Unit Tests                │
                 ↑            │          │                          │
                 │            └──→ Test Cases                       ↓
                 │                     │                  CI/CD →
Running System → Telemetry
                 │                     │
|
                 └─────────────────────┴──────── Observer/Evaluator
←───────────────┘
                                                        │
                                                        ▼
                                                    New Intent
```

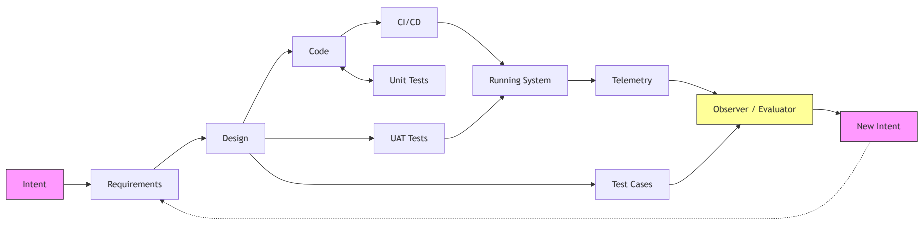


Diagram 0

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).

2.2 Graph Properties

- **Directed**: Edges have a source type and target type
- **Cyclic**: Feedback edges (Telemetry → New Intent) create cycles
- **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. A legal document, a physics paper, an organisational policy each have different graphs
- **Zoomable**: Graph granularity is a choice (see §2.5)

2.3 Asset as Markov Object

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. (You don't need the big bang to interact with the insect — just its boundary.)
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).

2.4 Graph Construction (Abiogenesis)

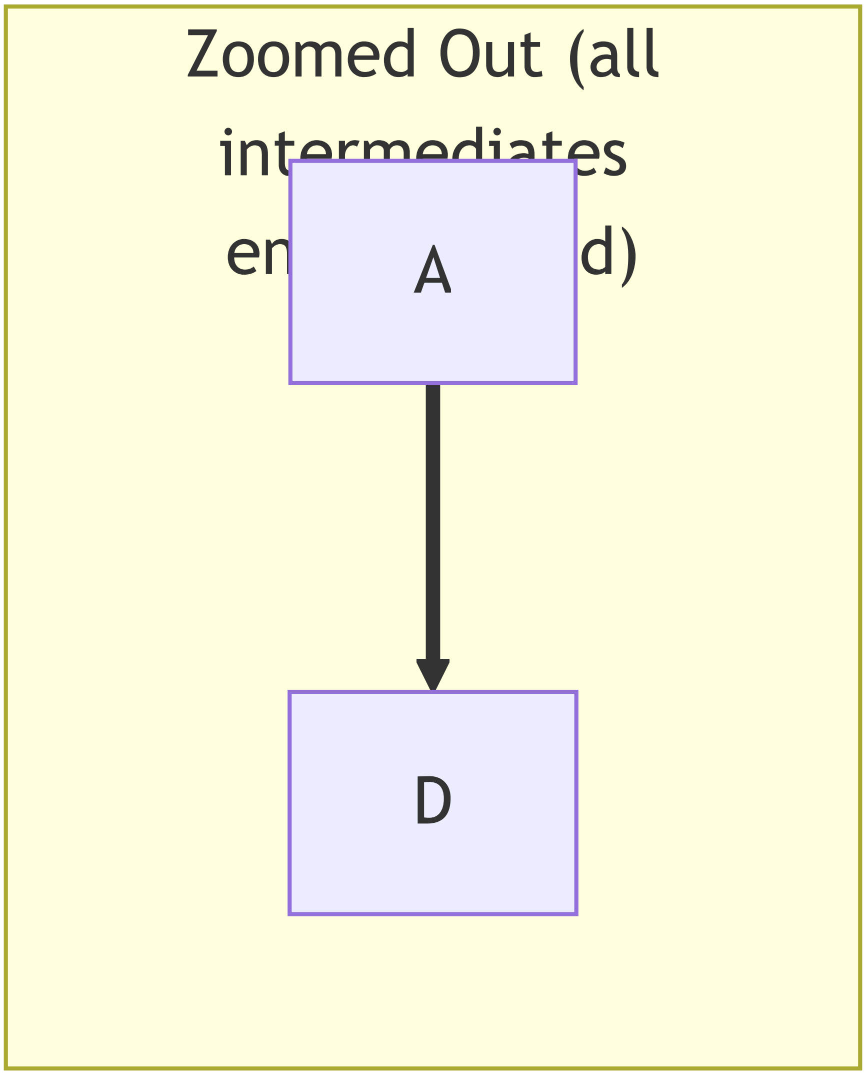
The graph topology follows the abiogenesis pattern (#39):

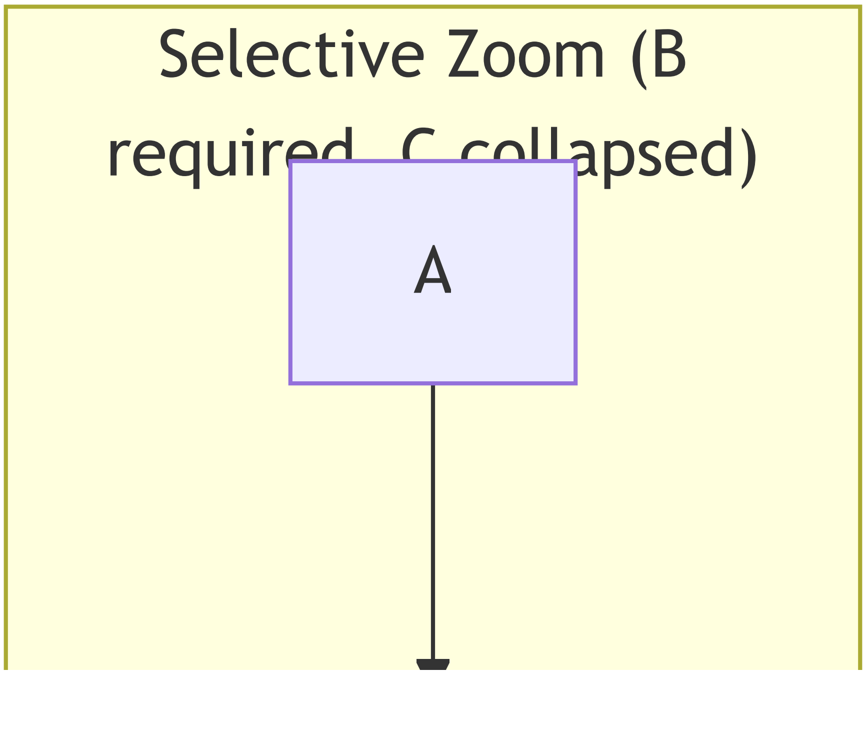
1. **Constraint**: Domain needs, technical limitations, regulatory requirements
2. **Constructor**: Practitioners working, experimenting (practice precedes structure)
3. **Encoding emerges**: Patterns crystallise into a graph topology (e.g., "we always need requirements before design")
4. **Encoding 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

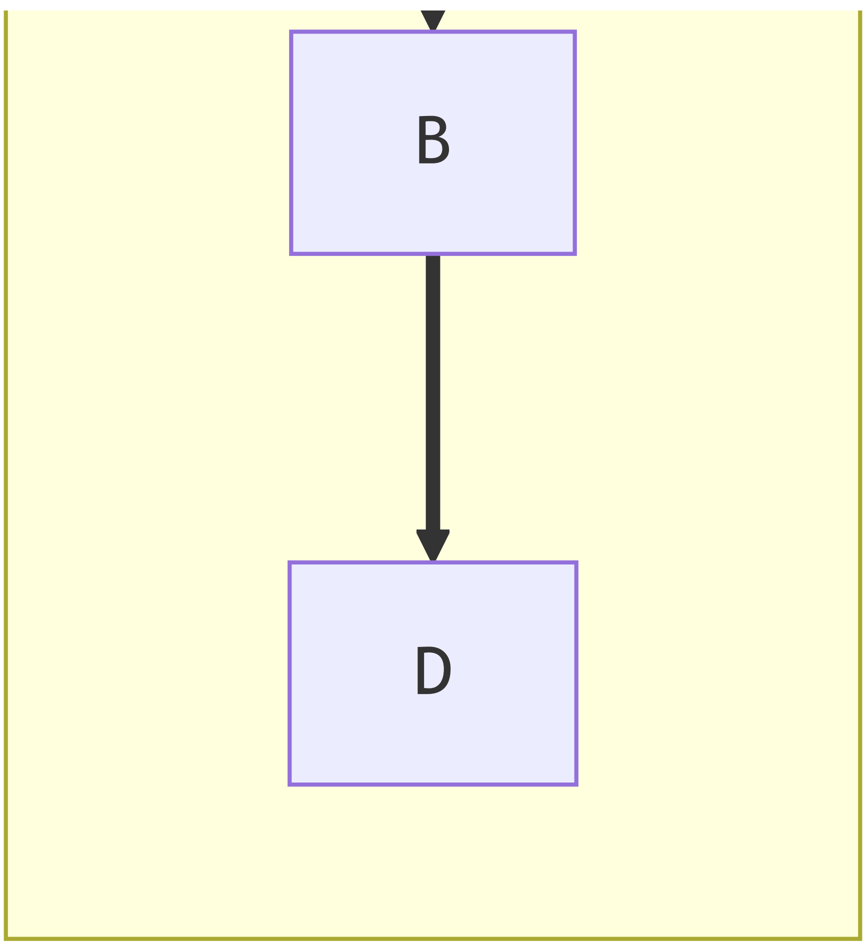
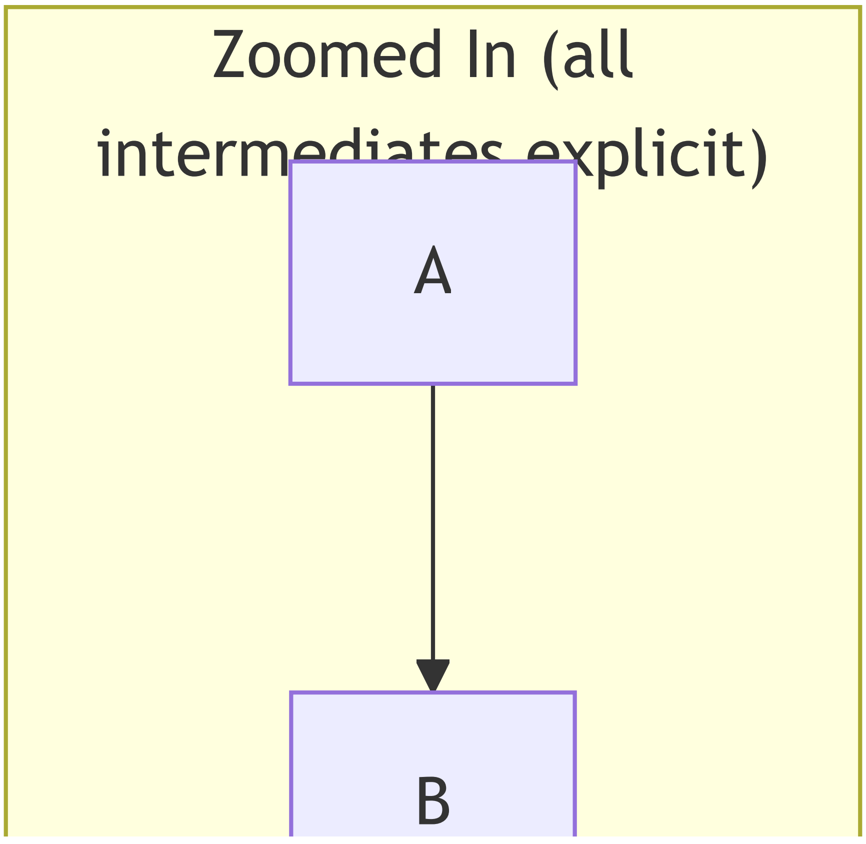
The SDLC graph is one such crystallisation. It is not privileged.

2.5 Graph Scaling (Zoom)

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


Zoomed Out (all intermediates en[illegible]d)
A
D


Selective Zoom (B required, C collapsed)
A


B
D

Zoomed In (all intermediates explicit)
A
B

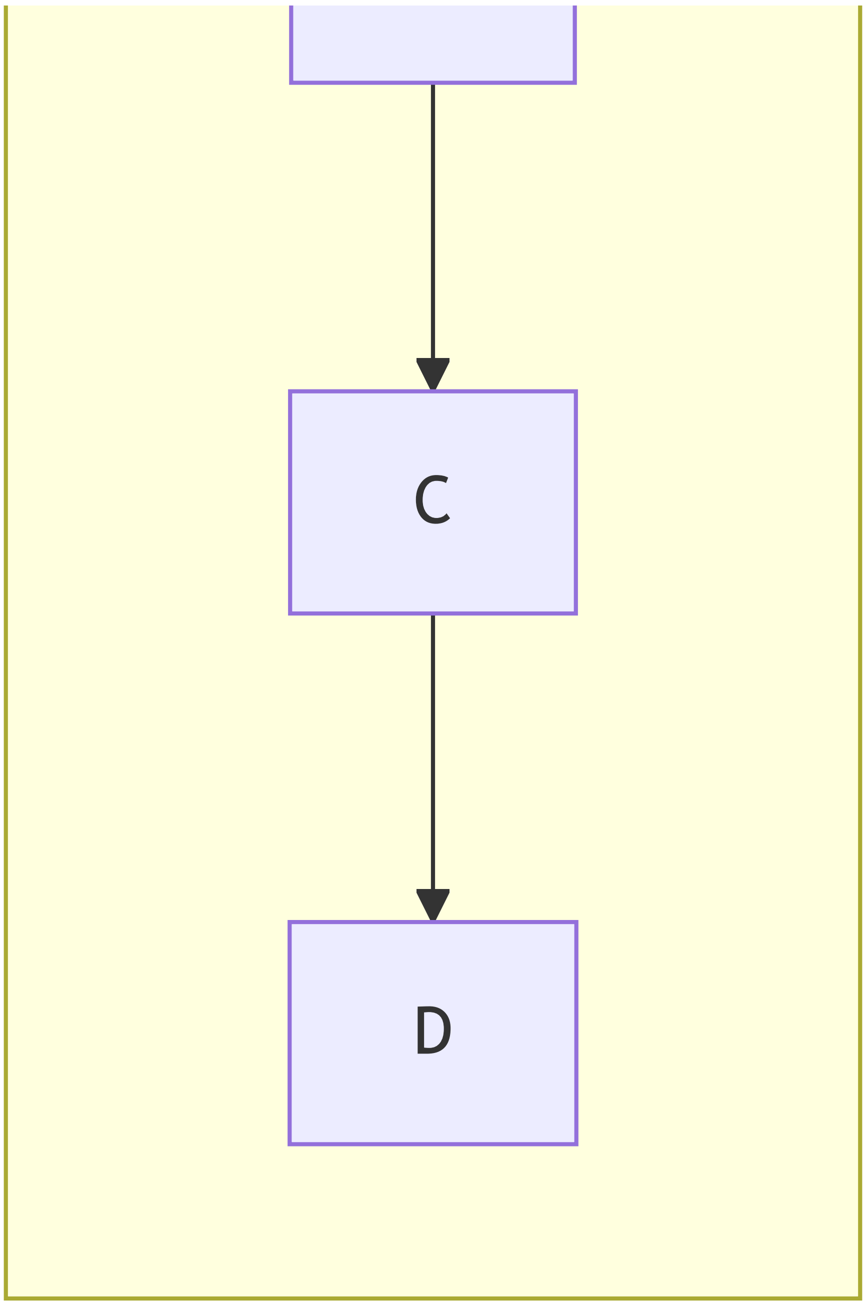


Diagram 1

Zooming in forces intermediate results to be created as explicit, evaluable assets. **Zooming out** treats the whole sub-graph as a single iterative edge. **Selective zoom** — the common case — collapses some intermediates while retaining others as mandatory waypoints.

Example: a PoC skips formal requirements and UAT but still requires a design document:

```
Full:       Intent → Requirements → Design → Code ↔ Tests → UAT
PoC:        Intent ──────────────→ Design → Code ↔ Tests
                                    ↑ mandatory intermediate
```

You can also zoom in further and require multiple layers within a single asset type. Design might expand into:

```
Design (zoomed out):   |design⟩
Design (zoomed in):    |high_level_design⟩ → |component_design⟩ →
|api_spec⟩ → |data_model⟩
Design (selective):    |high_level_design⟩ → |api_spec⟩
```

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.”

This is an operational choice, driven by Context[]:

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, balanced cost
Zoomed out	Rapid prototyping, well-understood problems, trusted constructors	Fewer assets, faster delivery, less overhead

The choice of which intermediates are mandatory is itself Context[] — it can be set at the project level (project_constraints), the feature level (feature vector overrides), or the profile level (projection profiles).

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.

The four primitives (Graph, Iterate, Evaluators, Spec+Context) are the same at every scale. The graph is fractal.

2.6 The Spec/Design Boundary

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

- **Spec** (Requirements) = **WHAT** the system does. Tech-agnostic. One spec can have many designs.
- **Design** = **HOW** architecturally. Tech-bound — ADRs, ecosystem binding, component architecture.

This boundary is the Requirements → Design edge. Everything upstream of this edge (Intent, Requirements) describes the problem and constraints without committing to technology. Everything downstream (Design, Code, Tests) is bound to a specific technology stack and architecture.

```
                 Spec/Design boundary
                        │
  Intent → Requirements │ Design → Code ↔ Tests → ...
                        │
  WHAT (tech-agnostic)  │  HOW (tech-bound)
  One spec              │  Many possible designs
```

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

2.7 Multiple Implementations Per Spec

A single requirement key can have multiple design variants:

```
REQ-F-AUTH-001 (spec)
  ├── design.claude  → code.python  → tests.pytest
  ├── design.codex   → code.python  → tests.pytest
  └── design.rust    → code.rust    → tests.cargo
```

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 (§11) in action — multiple vectors in superposition until measurement (production telemetry) collapses to the best-performing variant.

---

3. The Iteration Function

3.1 Signature

```
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
```

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.

This is the **only operation**. Every edge in the graph is this function called repeatedly until evaluators report convergence, at which point the candidate is promoted:

```
while not stable(candidate, edge_type):
    candidate = iterate(candidate, context, evaluators)
return promote(candidate)   // ATn.m becomes ATn+1.0
```

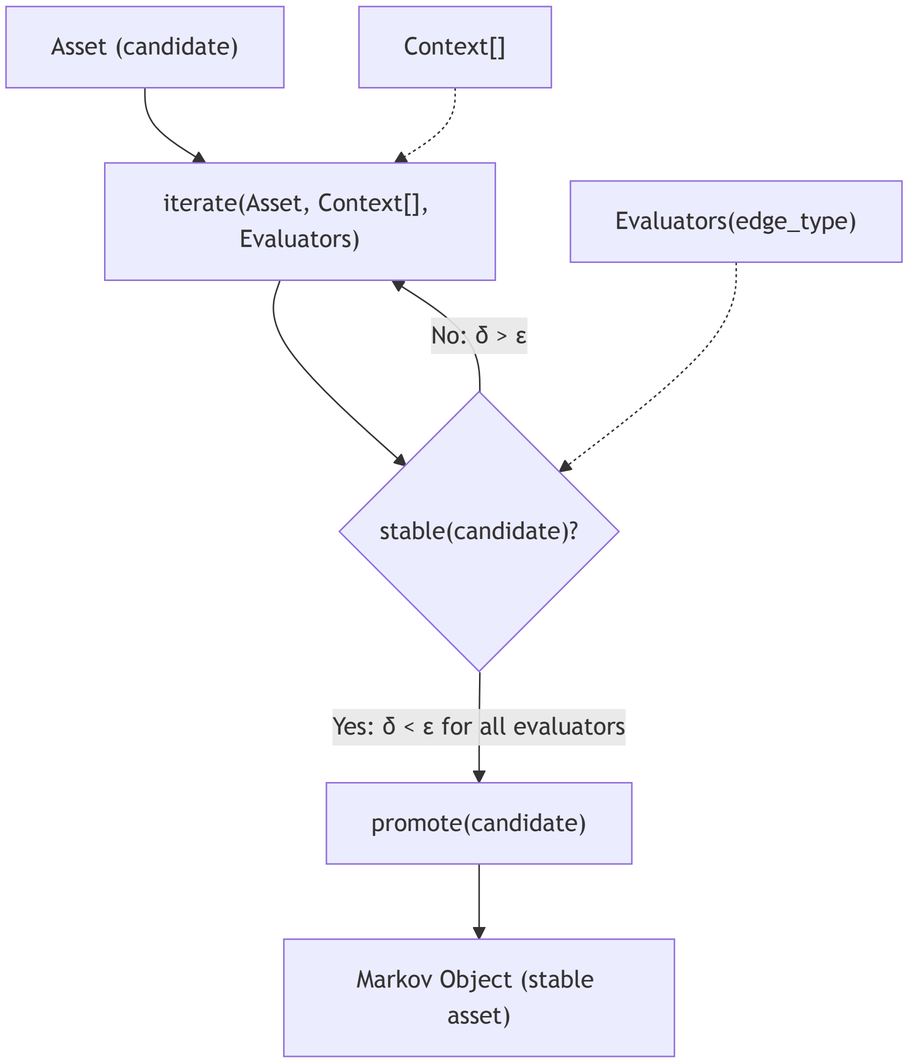

Asset (candidate)
Context[]
iterate(Asset, Context[], Evaluators)
Evaluators(edge_type)
No: δ > ε
stable(candidate)?
Yes: δ < ε for all evaluators
promote(candidate)
Markov Object (stable asset)

Diagram 2

3.2 Ontology Mapping

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.

3.3 Convergence

```
stable(candidate, edge_type) =
    ∀ evaluator ∈ evaluators(edge_type):
        evaluator.delta(candidate, Spec) < ε
```

Convergence is evaluator-dependent and edge-dependent. The iteration engine is universal. The stopping condition is parameterised.

---

4. Evaluators

4.1 The 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 suggestions
Deterministic Tests	Deterministic (#45)	Pass/fail. Type checks, schema validation, test suites, contract verification, SLA monitors

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.

4.2 Evaluator Composition Per Edge

Different graph edges use different evaluator combinations:

Transition	Typical Evaluators
	intent⟩ →
	req⟩ →
	design⟩ →
	code⟩ →
	design⟩ →
	code⟩ →
	running⟩ →
	telemetry⟩ →

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

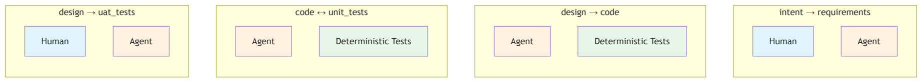


Diagram 3

4.3 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.

---

5. Context: The Constraint Surface

5.1 What Context Contains

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.

5.2 Context as Constraint Density

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
```

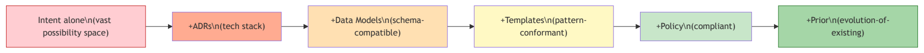

Intent alone\n(vast possibility space)
+ADRs\n(tech stack)
+Data Models\n(schema-compatible)
+Templates\n(pattern-conformant)
+Policy\n(compliant)
+Prior\n(evolution-of-existing)

Diagram 4

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.**

5.3 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.

---

6. Feature Vectors: Trajectories Through the Graph

6.1 Feature as Composite Vector

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

```
Feature F = |req⟩ + |design⟩ + |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.

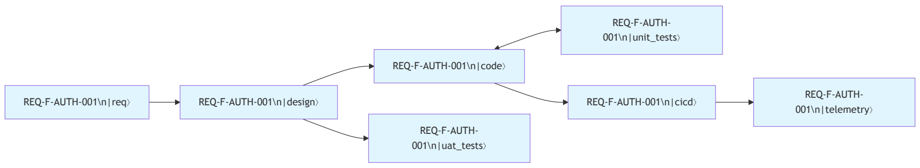


Diagram 5

6.2 Intent Lineage

Intent lineage is the feature's **accumulated state** across all edges: the original intent, every intermediate candidate produced, every decision made, every constraint satisfied. It is the full traceability chain, carried forward in each asset (§2.3).

This is the ontology's constraint propagation (#2) through the asset graph. REQ-F-AUTH-001 propagating across edges is a constraint signal maintaining coherence across the entire composite vector.

6.3 Feature Lineage in Code and Telemetry

REQ keys are not just documentation — they are **runtime-observable identifiers** that thread from spec to production:

```
Spec:         REQ-F-AUTH-001 defined
Design:       Implements: REQ-F-AUTH-001
Code:         # Implements: REQ-F-AUTH-001
Tests:        # Validates: REQ-F-AUTH-001
Telemetry:    logger.info("login", req="REQ-F-AUTH-001", latency_ms=42)
```

The tag format (`Implements: REQ-*`, `Validates: REQ-*`) is the contract. The comment syntax is language-specific (`#` for Python, `//` for Go/JS, `--` for SQL). In telemetry, the REQ key appears as a structured field in logs, metrics, and traces.

This makes features **observable at runtime**: you can query production telemetry by REQ key to see how a specific requirement behaves in production, measure latency per feature, detect regressions per feature, and route incidents back to the originating requirement.

6.4 Feature Views

A **feature view** is a generated cross-artifact status report for a single REQ key, produced by grepping the tag format across all artifacts:

```
Feature View: REQ-F-AUTH-001
────────────────────────────
Spec:       docs/specification/requirements.md     ✓ defined
Design:     docs/design/auth/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
```

Feature views are generated at every stage — they are the mechanism for answering "where is this feature in the graph?" at any point in time. The view is computed by searching for the REQ key across all artifacts, not maintained as a separate document.

This is the practical realisation of the composite vector (§6.1): the feature view shows the vector's current projection onto each basis component.

6.5 Feature Dependencies

Features can depend on other features. Feature B's code asset might require Feature A's code asset to be stable first:

```
Feature A: |req⟩ → |design⟩ → |code⟩●
                                    │ dependency
Feature B: |req⟩ → |design⟩ → |code⟩ → |tests⟩
```

Dependencies are between features (or their component assets), not between pipeline stages. This is a cross-vector constraint.

6.6 Task Planning as Trajectory Optimisation

Tasks **emerge** 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⟩)
```

---

7. The Full Lifecycle

7.1 Beyond Testing: CI/CD, Telemetry, Homeostasis

The full lifecycle includes deployment, runtime, and feedback assets in the graph:

Phase	Assets	What converges
Build		req⟩ →
Deploy		cicd⟩
Observe		telemetry⟩
Maintain	Homeostasis check	Is
Discover		new_intent⟩

7.2 The Running System as Markov Object

The deployed, running system is itself a Markov object (#7):

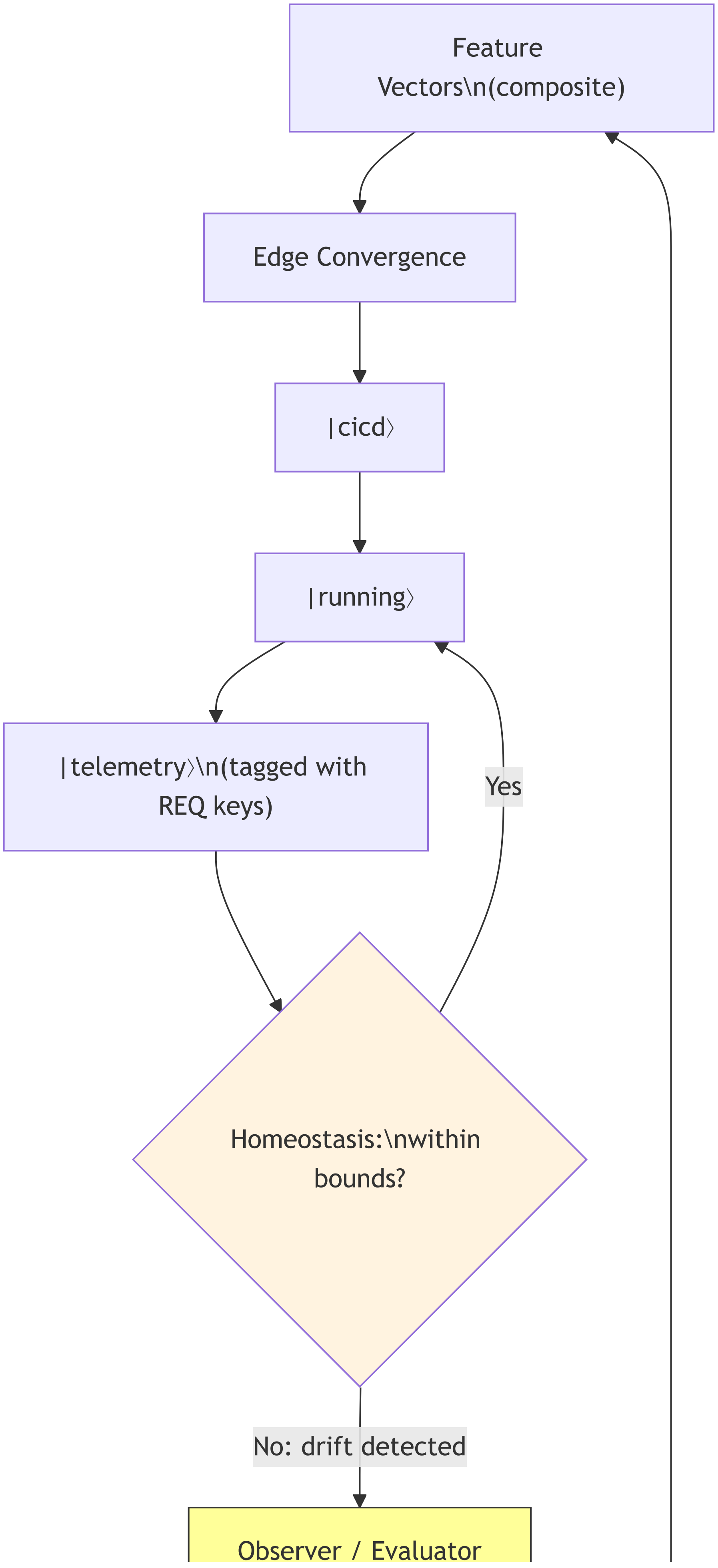
- **Boundary**: Its interfaces, SLAs, contracts, health endpoints
- **Internal dynamics**: Its runtime behaviour
- **Conditional independence**: It runs without knowing how it was built

Homeostasis is this Markov object maintaining its own boundary conditions — the ontology's teleodynamic transition (#49): a self-maintaining system that acts on its own behalf.

7.3 The Complete Cycle

```
Feature vectors (composite) → edge convergence → |cicd⟩ → |running⟩
                                                                 │
         ┌───────────────────────────────────────────────────────┘
         ▼
   |telemetry⟩ (tagged with REQ keys)
         │
         ▼
   Homeostasis: within constraint bounds?
         │                    │
        yes                   no → drift detected
         │                    │
         ▼                    ▼
      continue          Observer/Evaluator
```

```
       │
       ▼
|new_intent⟩ (new feature vectors)
       │
       ▼
back into the graph
```


Feature Vectors\n(composite)
Edge Convergence
|cicd〉
|running〉
|telemetry〉\n(tagged with REQ keys)
Yes
Homeostasis:\nwithin bounds?
No: drift detected
Observer / Evaluator

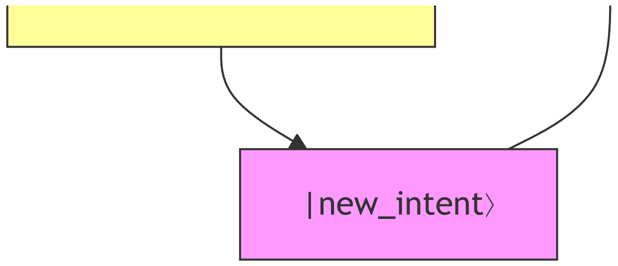


Diagram 6

The Observer/Evaluator at runtime uses the same three evaluator types:

Evaluator	At Runtime
Human	User feedback, business review, incident response
Agent(intent, context)	Anomaly detection, pattern analysis, intent discovery
Deterministic Tests	Alerting thresholds, SLA checks, health probes, contract monitors

7.4 Self-Maintaining Specification

When the cycle closes, the system becomes **teleodynamic** (#49): it maintains and updates its own specification from production experience. The specification is a **living encoding** (#46):

- Defines target structure (functional, quality, data constraints)
- Is continuously compared against runtime behaviour
- Evolves based on deviations and discoveries
- Drives corrective construction automatically

This is the abiogenesis insight (#39) completing itself: the methodology starts as a pipeline (constructor without feedback), then the feedback loop closes (encoding updates from runtime observation), and the system becomes self-maintaining. The graph topology itself can evolve through this process.

---

8. Ontology Traceability

8.1 Concept Mapping

Model Element	Ontology Concept	#
Stable asset	Markov object	7
Asset boundary (interface/schema)	Markov blanket	8

Model Element	**Ontology Concept**	**#**
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[] (incl. graph topology)	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 (σ)
Candidate (not yet stable)	Pre-Markov iteration state	15
Feature vector (composite)	Trajectory through constraint manifold	15 + 9
REQ key	Vector identifier / lineage tag	44
Task graph	Emergent structure from feature dependencies	3 (generative principle)
Probabilistic compute	Stochastic expansion	45
Deterministic compute	Verification contraction	45
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 topology as Context	Abiogenesis — encoded structure from practice	39
Spec/Design boundary	Boundary between encoded representation and constructor	40, 41

Model Element	Ontology Concept	#
Feature view (cross-artifact)	Observable projection of trajectory state	15 + 9
Multiple design variants	Superposition of alternative constructions	45

8.2 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)
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

8.3 The Abiogenesis Sequence

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

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

---

9. Relationship to Prior Model

9.1 What Changes

Prior Model (v1.x)	Asset Graph Model (v2.x)
7 sequential stages	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	Universal iteration function, parameterised per edge
Linear pipeline	Cyclic graph with feedback edges
Fixed graph topology	Graph is domain-constructed, zoomable, lives in Context[]
Stops at 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 tagging	Traceability as composite vector coherence (REQ key = vector ID)
UAT as a gate	UAT as scale-dependent assurance — explicit asset or inherent evaluator depending on zoom

9.2 What Is Preserved

- **REQ key system** — now understood as composite vector identifiers (trajectory tags)
- **TDD workflow** — iteration pattern at the |code⟩ → |unit_tests⟩ edge
- **BDD scenarios** — iteration pattern at the |design⟩ → |uat_tests⟩ edge
- **Key Principles** — constraints within Context[] that bound construction
- **Requirement traceability** — now formalised as intent lineage carried in each asset
- **Two compute regimes** — probabilistic (LLM generation) + deterministic (test verification)
- **Feedback loops** — first-class graph edges

9.3 What Is Added

- **Composite vectors** — features are trajectories through the graph, not pipeline travellers

- **Zoomable graph** — granularity is a choice; any edge expandable into sub-graph, any sub-graph collapsible
- **Scale-dependent assurance** — UAT:product :: unit tests:module — same pattern at different scales
- **Domain-constructed graphs** — topology is Context[], not universal
- **Graph abiogenesis** — graph topology itself follows the construction pattern
- **CI/CD, Telemetry, Homeostasis** — as graph assets, not afterthoughts
- **Formal ontology grounding** — every element traceable to constraint-emergence concepts
- **Spec/Design boundary** — WHAT (tech-agnostic) vs HOW (tech-bound), one spec many designs
- **Feature lineage in telemetry** — REQ keys in logs/metrics/traces, observable at runtime
- **Feature views** — cross-artifact status per REQ key, generated by grepping tag format
- **Multiple implementations** — same spec, different designs, telemetry-driven variant selection

---

10. Summary

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 (including the graph topology itself) that prevents degeneracy
5. Evaluated by **{Human, Agent, Tests}** — composable convergence criteria per edge
6. Split at a **Spec/Design boundary** — Spec = WHAT (tech-agnostic), Design = HOW (tech-bound). One spec, many designs.
7. Observable via **feature views** — REQ keys grepped across all artifacts produce per-feature status at any time
8. Completing the **full lifecycle** — through CI/CD, Telemetry (tagged with REQ keys), Homeostasis, and back to Intent

The graph is not universal — it is domain-constructed via abiogenesis (#39). The SDLC graph is one crystallisation. A legal document, a physics paper, an organisational policy each produce different graphs from the same four primitives. The graph is zoomable: any edge expandable into a sub-graph, any sub-graph collapsible into a single edge.

The methodology is an instantiation of the constraint-emergence ontology's information-driven construction pattern (#38), with the iteration function as local preorder traversal (#15) on the constraint manifold (#9), producing stable Markov objects (#7) that become constraints for the next construction, closing the feedback loop (#44) through runtime observation, and achieving self-maintaining teleodynamic status (#49) when the specification becomes a living encoding (#46) updated from production experience.

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

---

11. Feature Vectors as Hilbert Space

> **Note**: This section describes structure discovered in the SDLC instantiation that should propagate back to the parent ontology as a new concept: **cost dynamics on constraint manifolds**.

11.1 The Vector Space

Feature vectors form a Hilbert space whose basis is the set of asset types:

Hilbert space concept	Feature vector interpretation
Basis	Asset types — the node types defined by the graph
Vectors	Composite features: F = Σ
Inner product	Feature overlap — shared assets at a given node type
Orthogonality	Independent features — zero shared components, fully parallelisable
Basis decomposition	Atomic features — irreducible single-edge trajectories
Norm	Vector cost — sum of iteration effort across all components
Projection	Task batching — grouping features that share edges/assets
Superposition	Multiple features in-flight, components not yet converged
Measurement/collapse	Evaluator convergence — asset collapses to stable Markov object

The two compute regimes (#45) map directly:

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

11.2 The Hamiltonian: Cost and Effort

The **Hamiltonian** H is the operator governing evolution (effort/cost) on the feature vector space:

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

Hamiltonian concept	Asset graph interpretation
H (total energy)	Total effort/cost of the system
Kinetic energy T	Iteration cost — compute, human time per cycle
Potential energy V	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
Eigenvalues	Stable cost points — converged components have definite effort

- **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)

Feature vector total cost:

```
Cost(feature) = Σ H(edge) for each component projection
```

11.3 Task Planning as Action Minimisation

Task planning (§6.4) becomes: **minimise the action** over the feature vector space.

- Decompose into basis vectors (atomic single-component features)
- Identify orthogonal subsets (parallel work — zero inner product)
- Project overlapping vectors onto shared components (batching)
- Sequence by dependency constraints
- Optimise: converge the composite vector at minimum total H

Constraint density (#16) is the **metric** on the space — it determines how “far apart” states are and therefore how much effort transitions require. This connects hallucination prevention (§5.2) to cost dynamics: sparse constraints = high-dimensional degeneracy = expensive to converge = high V.

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.
- **Prior AI SDLC (v1.x)** — tagged `v1.x-final` in this repository. The 7-stage pipeline model this document supersedes.