AI SDLC — Project Genesis: Implementation Requirements

Version: 3.13.0 **Date**: 2026-02-23 **Derived From**: AI_SDLC_ASSET_GRAPH_MODEL.md (v2.8.0) **Parent Theory**: Constraint-Emergence Ontology

---

Purpose

Platform-agnostic implementation requirements for tooling that delivers the AI SDLC Asset Graph Model. Defines WHAT, not HOW. Concept numbers (#N) reference the ontology.

Audience: Implementers building AI SDLC tooling for any platform.

Requirement Format

`REQ-{CATEGORY}-{SEQ}[.MAJOR.MINOR.PATCH]`

Categories: `INTENT`, `GRAPH`, `ITER`, `EVAL`, `CTX`, `FEAT`, `LIFE`, `EDGE`, `TOOL`, `UX`, `COORD`, `SUPV`

Version semantics: PATCH (clarification), MINOR (criteria change), MAJOR (breaking). No version suffix = current.

Document Structure

1. Intent & Spec — Entry point: what to build
2. Asset Graph — Topology: admissible transitions
3. Iteration Engine — The only operation
4. Evaluators — Convergence criteria
5. Context — Constraint surface
6. Feature Vectors & Traceability — Trajectories through the graph
7. Full Lifecycle — CI/CD, telemetry, homeostasis
8. Sensory Systems — Interoception, exteroception, affect triage
9. Edge Parameterisations — TDD, BDD, ADRs per edge type
10. Tooling — Plugins, workspace, commands
11. User Experience — State-driven routing, progressive disclosure, observability
12. Multi-Agent Coordination — Agent identity, event-sourced assignment, work isolation

13. Supervision (IntentEngine) — Universal observer/evaluator composition law

---

1. Intent & Spec

REQ-INTENT-001: Intent Capture

Priority: Critical | **Phase**: 1

The system shall capture intents (desires for change) in a structured format that enters the asset graph.

Acceptance Criteria: - Intents from: human input, runtime feedback, ecosystem changes - Unique identifier (INT-*) - Fields: description, source, timestamp, priority - Persisted, version-controlled

Traces To: Asset Graph Model §6.2 (Intent Lineage) | Ontology #36 (delta/intent)

REQ-INTENT-002: Intent as Spec

Priority: High | **Phase**: 1

Intents shall compose with Context[] to form the Spec — the encoded representation (#40) that drives construction.

Acceptance Criteria: - Intent + Context[] = Spec for a given graph traversal - Spec is the fitness landscape against which evaluators measure convergence - Spec evolves as intent lineage accumulates through traversal

Traces To: Asset Graph Model §3.1 (Iteration Function signature) | Ontology #40 (encoded representation)

REQ-INTENT-003: Eco-Intent Generation

Priority: Medium | **Phase**: 2

The system shall generate intents automatically when ecosystem changes are detected.

Acceptance Criteria: - Monitor for: security vulnerabilities, deprecations, API changes, compliance updates - Generate INT-ECO-* intents with ecosystem context - Feed into graph as new feature vectors

Traces To: Asset Graph Model §7.2 (Gradient at Production Scale) | Ontology #44 (deviation signal)

REQ-INTENT-004: Spec Reproducibility

Priority: High | **Phase**: 1

The Spec (Intent + Context[]) shall be canonically serialisable, versioned, and hashable for audit reproducibility.

Acceptance Criteria: - Spec has a deterministic canonical serialisation (same inputs → same byte sequence) - Each Spec instance is content-addressable (hash of canonical form) - Spec hash recorded at each iteration and at convergence - Independent tools given the same Intent + Context[] shall compute the same Spec hash - Spec versions are immutable — evolution produces new versions, not mutations

Traces To: Asset Graph Model §3.1 (Iteration Function — Spec as input), §5.1 (Context[] contents) | Ontology #40 (encoded representation), #7 (Markov object — stable boundary)

---

2. Asset Graph

REQ-GRAPH-001: Asset Type Registry

Priority: Critical | **Phase**: 1

The system shall maintain a registry of asset types with typed interfaces.

Acceptance Criteria: - Default types: Intent, Requirements, Design, Code, Unit Tests, Test Cases, UAT Tests, CI/CD, Running System, Telemetry - Each type has: schema/interface definition, Markov criteria (#7) - Registry extensible — new types addable without engine changes

Traces To: Asset Graph Model §2.1 (Asset Types) | Ontology #7 (Markov object), #8 (Markov blanket)

REQ-GRAPH-002: Admissible Transitions

Priority: Critical | **Phase**: 1

The system shall define and enforce admissible transitions between asset types.

Acceptance Criteria: - Transitions are directed edges with source type and target type - Graph is cyclic (feedback edges are first-class) - Only admissible transitions can be traversed - Transition registry extensible — new edges addable - Transitions logged for audit

Traces To: Asset Graph Model §2.2 (Graph Properties) | Ontology #5 (admissible transformation), #9 (constraint manifold)

REQ-GRAPH-003: Asset as Markov Object

Priority: High | **Phase**: 1

An asset shall achieve Markov object status only when it satisfies all evaluators for its type.

Acceptance Criteria: - Boundary: typed interface/schema present and valid - Conditional independence: usable without construction history - Stability: all evaluators report convergence - Non-converged assets are candidates, not Markov objects

Traces To: Asset Graph Model §2.3 (Asset as Markov Object) | Ontology #7 (Markov object)

---

3. Iteration Engine

REQ-ITER-001: Universal Iteration Function

Priority: Critical | **Phase**: 1

The system shall implement a single iteration function for all graph edge traversals.

Acceptance Criteria: - Signature: `iterate(Asset<Tn>, Context[], Evaluators(edge_type)) → Asset<Tn.k+1>` - The asset carries intent, lineage, and full history — these are not separate parameters - Same function for all edges; behaviour parameterised by evaluators and context - Constructor (#41) implementations are edge-specific (LLM agent, human, compiler, test runner) - Iteration repeats until `stable()` reports convergence

Traces To: Asset Graph Model §3.1 (Signature) | Ontology #15 (local preorder traversal), #41 (constructor)

REQ-ITER-002: Convergence and Promotion

Priority: Critical | **Phase**: 1

The system shall promote candidates to the next asset type only upon evaluator convergence.

Acceptance Criteria: - `stable(candidate, edge_type) = ∀ evaluator ∈ evaluators(edge_type): evaluator.delta(candidate, Spec) < ε` - Convergence threshold (ε) configurable per evaluator per edge - Promotion: `ATn.m` becomes `ATn+1.0` - Non-convergence after max iterations raises to human evaluator

Traces To: Asset Graph Model §3.3 (Convergence) | Ontology #7 (stability condition)

REQ-ITER-003: Functor Encoding Tracking

Priority: Medium | **Phase**: 2

The system shall track the functor encoding for each feature vector — mapping 8 functional units (Evaluate, Construct, Classify, Route, Propose, Sense, Emit, Decide) to rendering categories (F_D, F_P, F_H) — and record encoding escalations (natural transformation η) when a functional unit changes category during iteration.

Acceptance Criteria: - Profile encoding section defines strategy, mode, valence, and per-unit category mappings - Feature vectors carry a `functor` section with encoding_source, mode, valence, and overrides - Each trajectory entry supports an `escalations` array recording category changes - Category-fixed units enforced: Emit is always F_D, Decide is always F_H - `encoding_escalated` events emitted when η fires

Traces To: Asset Graph Model §2.9 (Functors), ADR-017 (Functor-Based Execution Model)

---

4. Evaluators

REQ-EVAL-001: Three Evaluator Types

Priority: Critical | **Phase**: 1

The system shall support three evaluator types, composable per graph edge.

Acceptance Criteria: - **Human**: judgment, approval/rejection, domain evaluation - **Agent(intent, context)**: LLM-based gap analysis, coherence checking, refinement — probabilistic compute (#45) - **Deterministic Tests**: pass/fail — type checks, schema validation, test suites, SLA monitors — deterministic compute (#45) - All three compute delta (#36) between current state and target state - Each evaluator type declares `processing_phase: reflex|affect|conscious` — Deterministic is reflex (autonomic), Human and Agent deliberative are conscious (deliberative). Affect is not an evaluator type — it is a valence vector (urgency, severity, priority) emitted by ANY evaluator on its gap finding: F_D emits affect via threshold breach, F_P via classified severity, F_H via human urgency judgment

Traces To: Asset Graph Model §4.1 (Three Evaluator Types), §4.3 (Three Processing Phases) | Ontology #35 (evaluator-as-prompter), #45 (two compute regimes), #49 (teleodynamic)

REQ-EVAL-002: Evaluator Composition Per Edge

Priority: High | **Phase**: 1

Each graph edge shall declare its evaluator composition.

Acceptance Criteria: - Edge type → set of evaluators constituting `stable()` - Composition configurable (not hardcoded) - Default compositions provided (per Asset Graph Model §4.2 table) - Override at project, team, or organisation level

Traces To: Asset Graph Model §4.2 (Evaluator Composition Per Edge)

REQ-EVAL-003: Human Accountability

Priority: Critical | **Phase**: 1

Humans shall remain accountable for all decisions. AI evaluators assist; humans decide.

Acceptance Criteria: - AI (Agent evaluator) suggestions require human review before acceptance at edges where Human evaluator is configured - Decisions attributed to humans, not AI - Override capability always available

Traces To: Asset Graph Model §4.1 (Human evaluator type)

---

5. Context

REQ-CTX-001: Context as Constraint Surface

Priority: High | **Phase**: 1

The system shall manage Context[] as the standing constraint surface bounding construction.

Acceptance Criteria: - Context types include (open-ended): User Disambiguations, ADRs, Data Models, Templates, Prior Implementations, Policy - Each context element narrows the space of admissible constructions (constraint density #16) - Context shared across graph edges; subset relevance per edge configurable - Context version-controlled

Traces To: Asset Graph Model §5.1 (What Context Contains), §5.2 (Context as Constraint Density) | Ontology #16 (constraint density), #9 (constraint manifold)

REQ-CTX-002: Context Hierarchy

Priority: Medium | **Phase**: 2

The system shall support hierarchical context composition.

Acceptance Criteria: - Levels: global → organisation → team → project - Later contexts override earlier contexts - Deep merge for objects - Customisation without forking

Traces To: Asset Graph Model §5.4 (Context Stability) | Ontology #23 (scale-dependent time)

---

6. Feature Vectors & Traceability

REQ-FEAT-001: Feature Vector Trajectories

Priority: Critical | **Phase**: 1

Features shall be tracked as trajectories through the asset graph, identified by REQ keys.

Acceptance Criteria: - REQ key format: `REQ-{TYPE}-{DOMAIN}-{SEQ}` (the trajectory identifier — stable and immutable) - Requirement version: `REQ-{TYPE}-{DOMAIN}-{SEQ}.MAJOR.MINOR.PATCH` (the requirement statement — versioned per §Requirement Format) - The key (without version suffix) is the immutable identifier; the versioned form tracks how the requirement statement evolves - Types: F (functional), NFR (non-functional), DATA (data quality), BR (business rule) - Keys propagate through all assets on the trajectory - Bidirectional navigation: forward (intent → runtime) and backward (runtime → intent)

Traces To: Asset Graph Model §6.1 (Feature as Vector), §6.2 (Intent Lineage) | Ontology #2 (constraint propagation)

REQ-FEAT-002: Feature Dependencies

Priority: High | **Phase**: 1

The system shall track cross-vector dependencies between features.

Acceptance Criteria: - Feature B can depend on Feature A's asset at a given graph node - Dependencies are between trajectories, not between individual assets - Circular dependencies detected and flagged - Dependency graph visualisable

Traces To: Asset Graph Model §6.3 (Feature Dependencies)

REQ-FEAT-003: Task Planning as Trajectory Optimisation

Priority: High | **Phase**: 1

Tasks shall emerge from feature vector decomposition, not top-down planning.

Acceptance Criteria: - Decompose intent into feature vectors - Map each vector's path through the asset graph - Identify inter-vector dependencies at each graph node - Compress: batch parallel edges, sequence dependent ones - Result: task graph with dependency order, parallelisation, batching

Traces To: Asset Graph Model §6.4 (Task Planning as Trajectory Optimisation) | Ontology #3 (generative principle)

---

7. Full Lifecycle

REQ-LIFE-001: CI/CD as Graph Edge

Priority: High | **Phase**: 2

Deployment shall be a first-class iterative graph transition, not an external afterthought.

Acceptance Criteria: - Code → CI/CD → Running System are admissible graph edges - Evaluators: Deterministic Tests (build, package, deploy checks) - Iteration: failed deployment retries with evaluator feedback - Release manifests list feature vector IDs (REQ keys)

Traces To: Asset Graph Model §7.2 (Gradient at Production Scale) | Ontology #49 (teleodynamic)

REQ-LIFE-002: Telemetry and Homeostasis

Priority: High | **Phase**: 2

The running system shall be monitored as a Markov object maintaining its boundary conditions.

Acceptance Criteria: - Telemetry tagged with feature vector IDs (REQ keys) - Homeostasis check: is the running system within constraint bounds? - Evaluators at runtime: Deterministic Tests (alerting, SLA checks, health probes), Agent (anomaly detection), Human (incident response) - Drift detection generates deviation signals (#44)

Traces To: Asset Graph Model §7.2 (Gradient at Production Scale) | Ontology #49 (teleodynamic), #44 (deviation signal)

REQ-LIFE-003: Feedback Loop Closure

Priority: Critical | **Phase**: 2

Runtime deviations shall generate new intents that re-enter the asset graph as new feature vectors.

Acceptance Criteria: - Deviations generate INT-* intents (automatic or via human review) - Intents include: source (runtime), deviation details, impacted feature vectors - New intents enter graph as new feature vectors - The specification becomes a living encoding (#46) — updated from runtime observation

Traces To: Asset Graph Model §7.1 (The Gradient — spec as constraint surface), §7.3 (Spec Review) | Ontology #46 (living encoding), #49 (teleodynamic)

REQ-LIFE-004: Feature Lineage in Telemetry

Priority: High | **Phase**: 2

Runtime telemetry shall carry REQ keys as structured fields, enabling per-feature observability from spec through production.

Acceptance Criteria: - Telemetry (logs, metrics, traces) includes `req="REQ-*"` as a structured field - REQ keys in telemetry match those in code (`Implements: REQ-*`) - Production queries filterable by REQ key (e.g., latency per feature, error rate per feature) - Incidents traceable to originating requirement via telemetry REQ key

Traces To: Asset Graph Model §6.3 (Feature Lineage in Code and Telemetry) | Ontology #44 (deviation signal)

REQ-LIFE-005: Intent Events as First-Class Objects

Priority: Critical | **Phase**: 1

Every observer point that detects a non-trivial delta shall emit an `intent_raised` event with full causal chain. The gradient (§7.1) operates at every evaluator, not only the telemetry→intent production edge.

Acceptance Criteria: - `intent_raised` event emitted to `events.jsonl` whenever an observer detects a delta warranting action beyond the current iteration - Event includes: `intent_id`, `trigger`, `delta`, `signal_source`, `vector_type`, `affected_req_keys`, `prior_intents`, `edge_context`, `severity` - The `prior_intents` field traces the causal chain — if intent A led to spec change that caused intent B, the chain A→B is recorded - Signal sources classified as one of: `gap`, `test_failure`, `refactoring`, `source_finding`, `process_gap`, `runtime_feedback`, `ecosystem` - Human presented with generated intent for decision (create feature vector, acknowledge, or dismiss)

Traces To: Asset Graph Model §7.3.1 (Intent Events as First-Class Objects) | Ontology #49 (teleodynamic — self-maintaining), #46 (living encoding)

REQ-LIFE-006: Signal Source Classification

Priority: High | **Phase**: 1

All observations that feed back into the intent system shall be classified by signal source. Classification is an **affect phase** operation (§4.3) — it requires pattern recognition and severity assessment but not full deliberation. Classification enables telemetry analysis (which signal types generate the most work, which are noise).

Acceptance Criteria: - Seven signal source types recognised: `gap` (traceability validation), `test_failure` (stuck delta or test revealing upstream deficiency), `refactoring` (structural debt beyond current scope), `source_finding` (backward gap detection escalation), `process_gap` (inward gap detection — methodology deficiency), `runtime_feedback` (production telemetry deviation), `ecosystem` (external change) - Each signal source has an intent template in the feedback loop edge configuration - Signal source recorded in every `intent_raised` event - Development-time signals (gap, test_failure, refactoring, source_finding, process_gap) use the same mechanism as production signals (runtime_feedback, ecosystem) - Gap analysis operation emits `intent_raised` with `signal_source: "gap"` for each gap cluster - TDD iterate emits `intent_raised` with `signal_source: "test_failure"` when same check fails > 3 iterations - TDD iterate emits `intent_raised` with `signal_source: "refactoring"` when structural debt exceeds current scope

Traces To: Asset Graph Model §7.1 (The Gradient — signal sources table), §7.5 (Methodology Self-Observation) | Ontology #44 (deviation signal)

REQ-LIFE-007: Spec Change Events

Priority: High | **Phase**: 1

When the specification absorbs a signal and updates, a `spec_modified` event shall be emitted. This enables spec archaeology, feedback loop detection, and impact analysis.

Acceptance Criteria: - `spec_modified` event emitted to `events.jsonl` whenever a requirement is added, modified, or deprecated - Event includes: `trigger_intent`, `signal_source`, `what_changed` (list of REQ key changes), `affected_req_keys`, `spawned_vectors`, `prior_intents` - Spec changes traceable: given any REQ key, the full history of modifications is reconstructable from `spec_modified` events - Feedback loop detection: if intent A → spec change → intent B → spec change, and intent B traces back to A via `prior_intents`, the cycle is visible in the event log - Rate of evolution derivable: which spec sections are volatile vs stable

Traces To: Asset Graph Model §7.3.2 (Spec Change Events) | Ontology #46 (living encoding — spec updates from experience)

REQ-LIFE-008: Protocol Enforcement Hooks

Priority: High | **Phase**: 1

Every `iterate()` invocation shall enforce mandatory side effects. Skipping event emission, feature vector update, or gap reporting shall be detectable and blockable.

Acceptance Criteria: - After every iteration, the following side effects are mandatory: 1. Event emitted to `events.jsonl` 2. Feature vector state updated 3. Project state view regenerated (or marked stale) 4. `source_findings` array present in the event (may be empty) 5. `process_gaps` array present in the event (may be empty) - Detection mechanism: verify side effects occurred after each iteration - Circuit breaker: if enforcement itself fails, log the failure as a TELEM signal rather than entering infinite regression - Protocol violations reported as `process_gap` with type `PROTOCOL_VIOLATION` - Hooks are classified as reflex processing phase (§4.3) — they fire unconditionally and require no deliberative judgment

Traces To: Asset Graph Model §7.7 (Protocol Enforcement Hooks), §4.3 (Three Processing Phases) | Ontology #49 (teleodynamic — boundary maintenance)

REQ-LIFE-009: Spec Review as Gradient Check

Priority: High | **Phase**: 1

The system shall support stateless review of workspace state against the spec, computing deltas and generating intents where the gradient is non-zero. This is `delta(state, constraints) → work` applied at the workspace-spec scale.

Acceptance Criteria: - Invocable on demand and after any completion event (feature convergence, release, gap validation) - Reads workspace snapshot: feature vectors, event log, convergence status, telemetry signals - Computes delta against spec: what spec asserts vs what workspace contains - Classifies non-zero deltas by signal source (gap, discovery, ecosystem, optimisation, user, TELEM) - Applies affect triage (§4.3): severity assessment, escalation decision - Generates draft `intent_raised` events with full causal chain (trigger, delta, signal_source, vector_type, prior_intents) - Draft intents presented to human for approval (create vector, acknowledge, dismiss) - Stateless and idempotent: same state + same spec = same intents

Traces To: Asset Graph Model §7.1 (The Gradient), §7.3 (Spec Review) | Ontology #36 (delta/intent), #49 (teleodynamic)

REQ-LIFE-010: Development Observer Agent

Priority: High | **Phase**: 2

The system shall provide a development observer agent — an agent specification executed by the AI assistant — that watches the workspace event stream and computes `delta(workspace_state, spec) → intents` after iterate/converge events. This closes the right side of the abiogenesis loop: act → emit event → observe → judge → feed back.

Acceptance Criteria: - Specified as an agent behaviour model (same delivery as iterate agent — no executable code beyond the specification) - Triggered by hooks after `iteration_completed`, `edge_converged`, `release_created`, and `gaps_validated` events - Reads events.jsonl, feature vector state, convergence status, and telemetry signals - Computes delta against the spec: what spec asserts vs what workspace contains - Classifies non-zero deltas by signal source (gap, discovery, ecosystem, optimisation, user, TELEM) - Emits `observer_signal` events with: observer_id, signal_source, delta_description, severity, recommended_action - Markov object: reads inputs (event log), emits events, no shared mutable state — event log is the mailbox - Draft intents presented to human for approval (actor model — no autonomous spec modification) - Idempotent: same workspace state + same spec = same observations

Traces To: Asset Graph Model §7.1 (The Gradient), §7.3 (Spec Review), §7.6 (The Conscious but not Living System) | Ontology #49 (teleodynamic — self-maintaining), #36 (delta/intent)

REQ-LIFE-011: CI/CD Observer Agent

Priority: High | **Phase**: 2

The system shall provide a CI/CD observer agent — an agent specification executed by the AI assistant — that watches build and test results after push events and computes `delta(build_state, quality_spec) → intents`. This is the gradient at the build/deploy scale.

Acceptance Criteria: - Specified as an agent behaviour model (same delivery as iterate agent) - Triggered by hooks after CI/CD pipeline completion (build success/failure, test results available) - Reads build logs, test results, coverage reports, and deployment status - Computes delta against quality constraints: expected green vs actual red, coverage thresholds, deployment health - Maps build failures back to REQ keys via `Implements:` / `Validates:` tags in failing code/tests - Emits `observer_signal` events with: observer_id=`cicd_observer`, build_status, failing_req_keys, coverage_delta - Generates draft `intent_raised` events for test gaps, build regressions, and coverage drops - Markov object: reads build artifacts, emits events, no shared mutable state - Can trigger automatic rollback intents when deployment health checks fail (draft only — human approves)

Traces To: Asset Graph Model §7.1 (The Gradient), §7.2 (Gradient at Production Scale) | Ontology #44 (deviation signal), #49 (teleodynamic)

REQ-LIFE-012: Ops Observer Agent

Priority: High | **Phase**: 2

The system shall provide an ops observer agent — an agent specification executed by the AI assistant — that watches production telemetry and computes `delta(running_system, spec) → intents`. This is the gradient at the operational scale — runtime homeostasis.

Acceptance Criteria: - Specified as an agent behaviour model (same delivery as iterate agent) - Triggered on schedule (configurable interval) or on alert from monitoring infrastructure - Reads production telemetry: latency percentiles, error rates, resource utilisation, incident reports - Computes delta against spec constraints: SLA targets, performance envelopes, resource bounds - Correlates telemetry anomalies with REQ keys via `req=` structured logging tags - Emits `observer_signal` events with: observer_id=`ops_observer`, metric_deltas, affected_req_keys, severity - Generates draft `intent_raised` events for: SLA breaches, performance regressions, resource exhaustion trends - Markov object: reads telemetry streams, emits events, no shared mutable state - Integrates with interoceptive signals (REQ-SENSE-001) — ops observer is a higher-level consumer of sensory data - Stateless: same telemetry snapshot + same spec = same observations

Traces To: Asset Graph Model §7.1 (The Gradient), §7.2 (Gradient at Production Scale), §7.6 (The Conscious but not Living System) | Ontology #49 (teleodynamic), #44 (deviation signal), #7 (Markov object)

---

8. Sensory Systems

These requirements specify continuous observation capabilities that run independently of iterate(), feeding signals into the three processing phases (§4.3) via interoception (self-sensing, §4.5.1) and exteroception (environment-sensing, §4.5.2).

REQ-SENSE-001: Interoceptive Monitoring

Priority: High | **Phase**: 2

The system shall continuously observe its own health state, independent of iterate() calls, and generate signals when internal conditions deviate from expected bounds. Monitoring runs within a long-running sensory service, not as a per-request on-demand invocation.

Acceptance Criteria: - Monitors run within a long-running sensory service, on a configurable schedule (default: daily) or on workspace open, not only during iterate() - Minimum interoceptive monitors (INTRO-001 through INTRO-007): event freshness, feature vector stall, test health, state view freshness, build health, spec/code drift, event log integrity - Each monitor produces a typed signal with: monitor_id, observation_timestamp, metric_value, threshold, severity (info/warning/critical) - Signals feed into the affect triage pipeline (§4.3, §4.5.4) — not directly to conscious review - Interoceptive signals logged to events.jsonl as `interoceptive_signal` event type - Monitor thresholds configurable per project (project_constraints.yml) and per profile (§2.6.2) - Hotfix profile may suppress non-critical interoceptive signals; full profile runs all monitors - Monitors are observation-only — they do not modify workspace files

Traces To: Asset Graph Model §4.5.1 (Interoception), §4.5.4 (Sensory Service Architecture), §4.3 (Three Processing Phases) | Ontology #49 (teleodynamic — self-maintaining), #44 (deviation signal)

REQ-SENSE-002: Exteroceptive Monitoring

Priority: High | **Phase**: 2

The system shall continuously observe the external environment and generate signals when external conditions affect the project. Monitoring runs within the sensory service, not as a per-request on-demand invocation.

Acceptance Criteria: - Monitors run within the sensory service, on a configurable schedule (default: daily) or on workspace open - Minimum exteroceptive monitors (EXTRO-001 through EXTRO-004): dependency freshness, CVE scanning, runtime telemetry deviation, API contract changes - Each monitor produces a typed signal with: monitor_id, observation_timestamp, external_source, finding, severity - Signals feed into the affect triage pipeline (§4.3, §4.5.4) for classification and escalation - Exteroceptive signals logged to events.jsonl as `exteroceptive_signal` event type - Monitor configuration specifies external sources (package registries, CVE databases, telemetry endpoints) - Spike profile may disable exteroception entirely; full profile runs all configured monitors - Monitors are observation-only — they do not modify workspace files

Traces To: Asset Graph Model §4.5.2 (Exteroception), §4.5.4 (Sensory Service Architecture), §4.3 (Three Processing Phases) | Ontology #49 (teleodynamic — self-maintaining), #44 (deviation signal)

REQ-SENSE-003: Affect Triage Pipeline

Priority: High | **Phase**: 2

Signals from interoceptive and exteroceptive monitors shall pass through an affect triage pipeline that classifies, weights, and decides escalation before reaching conscious review. Triage operates within the sensory service using a tiered approach: rule-based classification first, agent-classified for ambiguous signals.

Acceptance Criteria: - Every sensory signal passes through affect triage before escalation to conscious phase - Triage is tiered: rule-based classification (fast, pattern-matching) first; agent-classified (probabilistic agent) only for signals that don't match any rule - Triage assigns: signal_source classification (gap / ecosystem / vulnerability / staleness / drift / runtime_deviation), severity (info / warning / critical), recommended_action (defer / log / escalate) - Escalation thresholds configurable per profile: hotfix profile has low threshold (escalate aggressively), spike profile has high threshold (suppress most) - Below-threshold signals logged but not escalated — available for batch review - Above-threshold signals generate draft proposals via probabilistic agent evaluation (draft only — no file modifications) - Draft proposals surfaced to human via the review boundary (tool interface, REQ-SENSE-005) - Triage decisions logged to events.jsonl as `affect_triage` event type (signal_id, classification, severity, escalation_decision)

Traces To: Asset Graph Model §4.3 (Three Processing Phases — affect), §4.5.3 (Sensory Systems and Processing Pipeline), §4.5.4 (Sensory Service Architecture) | Ontology #49 (teleodynamic — self-regulating)

REQ-SENSE-004: Sensory System Configuration

Priority: Medium | **Phase**: 2

Interoceptive and exteroceptive monitors shall be configurable per project and per projection profile. Configuration is read by the sensory service from design-specified YAML schemas.

Acceptance Criteria: - Monitor registry in sensory service configuration (`sensory_monitors.yml` — which monitors are active, their schedules, thresholds) - Affect triage rules in separate configuration (`affect_triage.yml` — classification patterns, severity mapping, escalation thresholds) - Profile-level overrides: profiles (§2.6.2) can enable/disable specific monitors and adjust thresholds - Custom monitors addable via plugin architecture (REQ-TOOL-001) - Monitor health: if a monitor itself fails to run, it generates an interoceptive meta-signal (the system senses that its own sensing has failed) - Sensory configuration included in context snapshot (REQ-TOOL-008) for reproducibility

Traces To: Asset Graph Model §4.5 (Two Sensory Systems), §4.5.4 (Sensory Service Architecture), §2.6.2 (Projection Profiles) | Ontology #9 (constraint manifold — configurable)

REQ-SENSE-005: Review Boundary

Priority: High | **Phase**: 2

The sensory service shall expose a tool interface (the review boundary) that separates autonomous sensing from human-approved changes. Draft proposals generated by the sensory service shall only become workspace modifications after human approval through the interactive session.

Acceptance Criteria: - Sensory service exposes tool interface for: viewing monitor status, listing pending proposals, approving proposals, dismissing proposals - Draft proposals include full context: triggering signal, triage classification, proposed action (intent, diff, or spec modification) - Approved proposals are applied by the interactive session (not the sensory service) — the service cannot modify workspace files - Dismissed proposals are logged with reason for future learning - The review boundary preserves REQ-EVAL-003 (Human Accountability) — all file modifications require human approval - Two distinct event categories enforced: sensor/evaluate events (autonomous, observation-only) vs change-approval events (conscious, human-approved)

Traces To: Asset Graph Model §4.5.4 (Sensory Service Architecture — review boundary), §4.3 (Three Processing Phases — conscious phase requires human) | Ontology #35 (evaluator-as-prompter), #49 (teleodynamic — self-regulating within boundaries)

REQ-SENSE-006: Artifact Write Observation

Priority: High | **Phase**: 1

The system shall observe all file writes to methodology-managed directories (specification, design, code, tests) and emit structured events, regardless of whether the write occurred through iterate() or direct tool invocation. This ensures observability survives optimization — as iterate() collapses to fewer steps or is bypassed entirely, the system retains visibility into what assets are being constructed, modified, and completed.

Design Principle: Observability lives on a sliding scale from strict process adherence (full iterate protocol with evaluators) to pure invariant observation (detect that the asset appeared). This requirement ensures the invariant-observation end of the scale always functions, providing a floor of visibility that holds regardless of process compliance.

Acceptance Criteria: - File writes (create or modify) to artifact directories detected via platform hook mechanism (PostToolUse or equivalent) - Each detected write emits an `artifact_modified` event to events.jsonl with: timestamp, file_path, asset_type (mapped from path), tool used - First write to a new asset type in a session emits `edge_started` with trigger `artifact_write_detected` - Writes to non-artifact paths (.ai-workspace/, .git/, infrastructure files) are excluded - The hook is observation-only — never blocks the write operation - The hook fails silently on error (observation failure must not block construction) - Multi-tenant aware: paths under imp_/ mapped by subdirectory after tenant prefix - Events are sufficient for: real-time progress tracking, post-hoc audit, and root cause analysis

Traces To: Asset Graph Model §7.7 (Protocol Enforcement — addresses dogfooding observation at line 1600), §4.3 (Reflex Processing Phase), §4.5.1 (Interoception — self-observation of file system mutations) | Ontology #49 (teleodynamic — boundary maintenance)

---

9. Edge Parameterisations

These requirements specify evaluator configuration for common graph edges. They are parameterisations of the universal iteration function, not separate engines.

REQ-EDGE-001: TDD at Code ↔ Tests Edges

Priority: High | **Phase**: 1

The Code ↔ Unit Tests edge shall use TDD workflow as its iteration pattern. TDD is a **co-evolution** — test and code assets iterate together, not a strict directional transition from one to the other.

Acceptance Criteria: - RED: write failing test first (Deterministic Test evaluator reports delta — test exists, code does not yet satisfy it) - GREEN: write minimal code to pass (Deterministic Test evaluator reports convergence) - REFACTOR: improve code quality (all evaluators re-confirm convergence) - COMMIT: save with REQ key in message - The iteration oscillates between test and code assets until both converge — this is a single edge with bidirectional construction - Minimum coverage threshold configurable (default: 80%)

Traces To: Asset Graph Model §9.2 (What Is Preserved — TDD), §2.5 (Graph Scaling — co-evolution as zoomed-in sub-graph) | Context[]: Key Principles

---

REQ-EDGE-002: BDD at Design→Test Edges

Priority: High | **Phase**: 1

Design→Test Cases and Design→UAT Tests edges shall use BDD (Given/When/Then) as their iteration pattern.

Acceptance Criteria: - Scenarios in Gherkin format, tagged with REQ keys - System Test BDD: technical integration scenarios - UAT BDD: pure business language (no technical jargon) - Scenarios are executable, not just documentation - Every REQ key has ≥1 BDD scenario

Traces To: Asset Graph Model §9.2 (What Is Preserved — BDD)

REQ-EDGE-003: ADRs at Requirements→Design Edge

Priority: High | **Phase**: 1

The Requirements→Design edge shall produce Architecture Decision Records as Context[] artefacts.

Acceptance Criteria: - ADRs document: decision, context, consequences, alternatives considered - ADRs acknowledge ecosystem constraints (Context[]) - ADRs reference requirement keys - ADRs are versioned and become part of Context[] for downstream edges

Traces To: Asset Graph Model §5.1 (Context[] contents — ADRs)

REQ-EDGE-004: Code Tagging

Priority: High | **Phase**: 1

Code assets shall include feature vector trajectory tags.

Acceptance Criteria: - Code comments include structured tags: `Implements: REQ-*` (comment syntax is language-specific; the tag format is the contract) - Test comments include structured tags: `Validates: REQ-*` - Commit messages include REQ keys - Tagging validated (not just documentation) — tooling parses the tag format, not the comment syntax

Traces To: Asset Graph Model §6.1 (REQ key as trajectory identifier)

---

10. Tooling

REQ-TOOL-001: Plugin Architecture

Priority: High | **Phase**: 1

Methodology delivered as installable plugins.

Acceptance Criteria: - Plugins include: agent configurations, skills, commands, templates - Discoverable, installable, versioned (semver)

Traces To: Asset Graph Model §5.1 (Context[] — open-ended, not a fixed list) | Ontology #9 (constraint manifold — composable)

REQ-TOOL-002: Developer Workspace

Priority: High | **Phase**: 1

Workspace structure for task and context management.

Acceptance Criteria: - Task tracking: active, completed, archived - Context preservation across sessions - Version-controlled (git)

Traces To: Asset Graph Model §5.4 (Context Stability) | Ontology #40 (encoded representation)

REQ-TOOL-003: Workflow Commands

Priority: Medium | **Phase**: 1

Commands for common workflow operations.

Acceptance Criteria: - Task management: create, update, complete - Context: checkpoint, restore - Status: progress, coverage, gaps

Traces To: Asset Graph Model §3.1 (Iteration Function — operationalising the engine) | Ontology #41 (constructor)

REQ-TOOL-004: Release Management

Priority: High | **Phase**: 1

Versioning and distribution capabilities.

Acceptance Criteria: - Semantic versioning - Changelog generation - Release tagging - Feature vector coverage summary in release notes

Traces To: Asset Graph Model §7.2 (Gradient at Production Scale — CI/CD as graph asset) | Ontology #7 (Markov object — stable boundary for release)

REQ-TOOL-005: Test Gap Analysis

Priority: High | **Phase**: 1

Identify test coverage gaps against feature vectors.

Acceptance Criteria: - Analyse REQ keys vs existing tests - Identify uncovered trajectories - Suggest test cases for gaps

Traces To: Asset Graph Model §6.1 (Feature as Composite Vector — completeness check) | Ontology #35 (evaluator-as-prompter — gap = delta)

REQ-TOOL-006: Methodology Hooks

Priority: Medium | **Phase**: 1

Lifecycle hooks automating methodology compliance.

Acceptance Criteria: - Hooks trigger on: commit, edge transition, session start, artifact write - Validate: REQ-* tags present, evaluator convergence recorded - Configurable per project

Traces To: Asset Graph Model §4.1 (Deterministic Tests evaluator type) | Ontology #45 (deterministic compute regime)

REQ-TOOL-007: Project Scaffolding

Priority: High | **Phase**: 1

Installer creates scaffolded project structure including all artifacts required by the observability layer.

Acceptance Criteria: - Creates asset graph configuration (`.ai-workspace/graph/graph_topology.yml`), context directories, workspace - Graph topology file includes asset types, transitions, and constraint dimensions — this is the integration contract between the methodology and any monitor/dashboard - Templates with placeholder guidance - Design-implementation binding via manifest - Scaffolded workspace passes monitor integration verification (all parsers return non-null)

Traces To: Asset Graph Model §2.4 (Graph Construction — abiogenesis) | Ontology #39 (abiogenesis — encoding emerges from practice)

Validated By: test06 GAP-042 (installer did not scaffold `graph_topology.yml`, breaking Genesis Monitor integration)

REQ-TOOL-008: Context Snapshot

Priority: Medium | **Phase**: 1

Context snapshot for session recovery and continuity.

Acceptance Criteria: - Captures: active tasks, current work context, timestamp - Immutable once created - Integrates with task checkpoint mechanism

Traces To: Asset Graph Model §2.3 (Asset as Markov Object — conditional independence) | Ontology #7 (Markov object — usable without construction history)

REQ-TOOL-009: Feature Views

Priority: High | **Phase**: 1

The system shall generate per-feature cross-artifact status reports by searching for REQ keys across all project artifacts.

Acceptance Criteria: - Given a REQ key, produce a feature view showing which artifacts reference it and at which stage (spec, design, code, tests, telemetry) - Feature views generated by searching tag format (`Implements: REQ-*`, `Validates: REQ-*`, `req="REQ-*"`) across all artifacts - Coverage summary: N of M stages tagged for each feature - Missing stages flagged (e.g., "REQ-F-AUTH-001: no telemetry tagging") - Feature views invocable on demand (e.g., trace command for a given REQ key)

Traces To: Asset Graph Model §6.4 (Feature Views) | Ontology #15 (trajectory observation)

REQ-TOOL-010: Spec/Design Boundary Enforcement

Priority: Medium | **Phase**: 1

The system shall enforce the Spec/Design boundary — requirements (WHAT) must be technology-agnostic; design (HOW) is technology-bound.

Acceptance Criteria: - Requirements documents must not reference specific technologies, frameworks, or libraries - Design documents must explicitly list technology choices as ADRs - Agent evaluator at the Requirements → Design edge checks for technology leakage in requirements - Multiple design variants per spec are supported (same REQ key, different design trajectories)

Traces To: Asset Graph Model §2.6 (Spec/Design Boundary), §2.7 (Multiple Implementations) | Ontology #40 (encoded representation), #41 (constructor)

REQ-TOOL-011: Installability

Priority: High | **Phase**: 1

The methodology tooling shall be installable into any project directory via a single platform-appropriate command. A methodology that cannot be installed cannot be used — installability is constitutive, not optional.

Acceptance Criteria: - Single-command installation from a remote source (e.g., `curl ... | python3 -`, `pip install`, `npm install`, or platform equivalent) - Installation creates the workspace structure (`.ai-workspace/`, configs, event log) and registers methodology commands/agents - Installation is idempotent — re-running preserves existing work (events, feature vectors, user artifacts) - Installation emits a `project_initialized` event on first run - Offline/air-gapped installation supported via local source path - Installation verifiable — a version/health command confirms successful setup - Uninstallation leaves user artifacts intact (events, specs, code) — only removes tooling scaffolding

Traces To: Asset Graph Model §0 (Genesis is a product — products must be deployable) | Ontology #39 (abiogenesis — the methodology must bootstrap itself into existence within a project)

REQ-TOOL-012: Multi-Tenant Folder Structure

Priority: High | **Phase**: 1

The system shall enforce a multi-tenant folder structure where specification (WHAT) and implementation (HOW) occupy distinct, well-defined directory subtrees. This enables independent parallel design variants that are each self-contained and independently buildable.

Acceptance Criteria: - Shared specification lives in `specification/` at project root — technology-agnostic, one per project - Each design variant lives in `imp_<design_name>/` — self-contained with its own `design/`, source code, build config, and tests - No generated source code, build files, or design documents may exist at the project root outside `specification/` and `imp_*/` - Each `imp_<name>/` directory is independently buildable (its own `build.sbt`, `setup.py`, `dbt_project.yml`, etc.) - Adding a second design variant (e.g., `imp_pyspark/` alongside `imp_scala_spark/`) requires no restructuring of the first - The `design→code` edge evaluator checklist includes a deterministic check: "All generated source files are under `imp_{variant}/`" - `project_constraints.yml` includes a `structure.design_tenants` section specifying the pattern

Traces To: Asset Graph Model §2.7 (Multiple Implementations — one spec, many designs) | Ontology #40 (encoded representation), #41 (constructor — design binds technology)

Validated By: test06 GAP-043 (code generated at root), GAP-044 (no constraint enforcing the pattern). Also the methodology repository itself (`imp_claude/`, `imp_gemini/`, `imp_codex/`) as the reference implementation of this pattern.

REQ-TOOL-013: Output Directory Binding

Priority: High | **Phase**: 1

The `design→code` edge shall bind a concrete output directory derived from the design variant name before code generation begins. The iterate agent must resolve this binding from `project_constraints.yml` or the feature vector before invoking any constructor.

Acceptance Criteria: - Output directory defaults to `imp_{design_variant}/` where `design_variant` is the active design name - Overridable via `project_constraints.yml` field `structure.output_directory` - The iterate agent passes the resolved output directory to the constructor as part of Context[] - Post-generation evaluator verifies no files were written outside the bound directory (excluding `specification/` and `.ai-workspace/`) - Design document (`DESIGN.md`, ADRs) placed in `imp_{variant}/design/`, not `specification/`

Traces To: Asset Graph Model §5 (Context as Constraint Surface — output directory is a project-level constraint) | REQ-CTX-001

Validated By: test06 GAP-043 (agent placed `build.sbt` and 8 module directories at project root; `DESIGN.md` placed in `specification/` conflating spec with design)

REQ-TOOL-014: Observability Integration Contract

Priority: Medium | **Phase**: 1

The installer shall scaffold all files required by the methodology's observability layer (monitors, dashboards, analysis tools). The workspace structure is the integration contract — if the monitor expects a file, the installer must create it.

Acceptance Criteria: - Installer creates `.ai-workspace/graph/graph_topology.yml` (asset types, transitions, constraint dimensions) - Installer creates or templates `.ai-workspace/STATUS.md` (project status summary) - Post-install verification checks that all expected paths exist and are parseable - Graph topology file is sourced from the methodology plugin's canonical `config/graph_topology.yml` - Workspace structure documented as the formal contract between methodology commands and observability tools - Changes to workspace structure are versioned and backward-compatible

Traces To: Asset Graph Model §7.4 (Event Sourcing — observable state) | REQ-UX-003 (Project-Wide Observability)

Validated By: test06 GAP-042 (Genesis Monitor showed blank dashboard because `graph_topology.yml` was absent)

---

11. User Experience

REQ-UX-001: State-Driven Routing

Priority: High | **Phase**: 1

The system shall detect project state and route to the appropriate action without requiring the user to know command names, edge syntax, or feature vector conventions.

Acceptance Criteria: - State detection: UNINITIALISED, NEEDS_CONSTRAINTS, NEEDS_INTENT, NO_FEATURES, IN_PROGRESS, ALL_CONVERGED, ALL_BLOCKED, STUCK - Single entry point routes to correct action (init, constraint capture, intent authoring, iterate, review, spawn, gaps, release) - State derived from workspace filesystem + event log — never from stored "current state" variable

Traces To: Asset Graph Model §7.4 (Event Sourcing — state derived, not stored) | Ontology #2 (constraint propagation — state computed from constraints)

REQ-UX-002: Progressive Disclosure

Priority: High | **Phase**: 1

The system shall request only the information needed for the current edge, deferring configuration until the edge where it is consumed.

Acceptance Criteria: - Project initialisation requires ≤5 user inputs (name, project kind, language, test runner, intent description) - Constraint dimensions (ecosystem, deployment, security, build) deferred until `requirements→design` edge - Advisory dimensions deferred until explicitly needed or user opts in - Sensible defaults inferred from project detection (language, framework, test runner from config files)

Traces To: Asset Graph Model §2.8 (Instantiation Layers — Layer 3 binding is incremental) | Ontology #39 (abiogenesis — encoding emerges from practice, not upfront specification)

REQ-UX-003: Project-Wide Observability

Priority: High | **Phase**: 1

The system shall provide a project-level status view that aggregates across all features, shows blocked/in-progress/pending state, surfaces unactioned signals, and previews the next recommended action.

Acceptance Criteria: - "You are here" indicator per feature: compact graph path with convergence markers - Cross-feature rollup: aggregate edge convergence counts - Blocked features shown with reason (spawn dependency, human review pending, stuck) - Unactioned `intent_raised` events surfaced as signals - Preview of what state-driven routing would do next - Workspace health check: event log integrity, feature vector consistency, orphaned spawns, stuck detection

Traces To: Asset Graph Model §7.5 (Self-Observation) | Ontology #15 (trajectory observation — observing the system's own trajectories)

REQ-UX-004: Automatic Feature and Edge Selection

Priority: Medium | **Phase**: 1

When multiple features are active, the system shall select the highest-priority actionable feature and determine the next unconverged edge from the graph topology, filtered by the active profile.

Acceptance Criteria: - Feature selection priority: time-boxed spawns (urgency) → closest-to-complete (reduce WIP) → feature priority → most recently touched - Edge determination: topological walk of graph, skip converged, skip edges not in profile - User can override selection explicitly - Selection reasoning displayed to user

Traces To: Asset Graph Model §6.3 (Task Planning as Trajectory Optimisation) | Ontology #45 (deterministic compute regime — selection is computable)

REQ-UX-005: Recovery and Self-Healing

Priority: Medium | **Phase**: 1

The system shall detect and guide recovery from inconsistent workspace states without destructive reset.

Acceptance Criteria: - Detect: corrupted event log, missing feature vectors, orphaned spawns, stuck features (δ unchanged 3+ iterations), unresolved mandatory constraints - Guided recovery: suggest specific fix actions for each detected issue - Non-destructive: never silently delete user data; always ask before overwrite - Workspace rebuildable from event log (event sourcing guarantee)

Traces To: Asset Graph Model §7.4 (Event Sourcing — views reconstructible from events) | Ontology #7 (Markov object — stable boundary enables recovery)

REQ-UX-006: Human Gate Awareness

Priority: High | **Phase**: 1

The system shall ensure that users are made aware of events requiring their approval or judgment through multiple, configurable notification channels. When an IntentEngine escalates (ambiguity persistent/unbounded), the escalation must reach a human — it must never be silently deferred or lost.

Acceptance Criteria: - Escalation events (`escalate` output from any IntentEngine level) are surfaced to the user through at least one active notification channel - In interactive mode (CLI session): escalations displayed inline with clear visual distinction (not buried in output) - In async/service mode (sensory service, CI/CD, multi-agent): escalations written to a **pending review queue** in `.ai-workspace/reviews/pending/` as structured YAML files - `/gen-status` always shows count of pending escalations and their summary (zero-query awareness) - `/gen-start` checks pending escalations before feature selection — escalations take priority over iteration - Notification channels are configurable per project (in `project_constraints.yml` under `notification_channels:`): - `inline` (default): display in current session - `review_queue`: write to pending review queue (always active as fallback) - Future extensible: webhook, email, Slack (platform-specific implementation) - Each pending escalation includes: source IntentEngine level, triggering signal, affected feature/edge/REQ keys, timestamp, recommended action - Escalations that remain unactioned for a configurable threshold (default: 3 iterations of the affected feature) are re-surfaced with elevated urgency - No escalation is silently dropped — the review queue is the guarantee of delivery

Traces To: Asset Graph Model §4.6.3 (`escalate` output type), §4.3 (conscious processing phase requires human), §4.5.4 (review boundary) | Ontology #23 (scale-dependent observation — human is the observer at conscious scale)

REQ-UX-007: Edge Zoom Management

Priority: High | **Phase**: 1

The system shall provide a user experience for expanding any graph edge into a sub-graph (zoom in), collapsing a sub-graph back into a single edge (zoom out), and managing the lifecycle of intermediate nodes created by zoom. The zoom UX enables the user to decompose complex edges without losing traceability or graph integrity.

Acceptance Criteria: - User can request zoom on any edge: `/gen-zoom --edge "design→code" --level in|out|selective` - **Zoom in**: creates intermediate asset types as new nodes between source and target. The system: - Proposes intermediate node types based on edge type and project context (e.g., `design→code` proposes `module_decomp → basis_projections → code_per_module`) - Each intermediate node gets its own asset type entry in the workspace graph topology (`.ai-workspace/graph/graph_topology.yml`) - Each new sub-edge inherits parent edge's evaluator types but can be overridden - Intent for the zoom (why this decomposition) is captured and traced to the parent edge - Sub-edge parameterisation files created in `.ai-workspace/graph/edges/` - **Zoom out**: collapses sub-graph back to a single edge. The system: - Verifies all intermediate nodes are converged (or explicitly abandoned) - Archives intermediate assets (not deleted — available for audit) - Restores original edge with convergence state reflecting sub-graph completion - **Selective zoom**: retains specific intermediate nodes as mandatory waypoints while collapsing others - User selects which intermediates to retain - Retained intermediates become explicit convergence checkpoints - Collapsed intermediates treated as internal to the encapsulating edge - Zoom operations emit events (`graph_zoom_in`, `graph_zoom_out`, `graph_zoom_selective`) with full before/after topology - Feature vectors traversing a zoomed edge see the sub-graph topology — their trajectory expands to include intermediate nodes - Zoom is reversible and auditable — the event log preserves the full zoom history - Zoom does not violate graph invariants: the graph remains directed, typed, and all edges have evaluators

Traces To: Asset Graph Model §2.5 (Graph Scaling — zoom levels, selective zoom, build decomposition) | Ontology #23 (scale-dependent observation — zoom is choosing the observation scale)

---

12. Multi-Agent Coordination

REQ-COORD-001: Agent Identity

Priority: High | **Phase**: 2

Every event emitted by an agent SHALL include an `agent_id` field (unique per agent instance) and an `agent_role` field (drawn from a project-defined role registry). Agent roles describe capabilities and evaluator authority (e.g., architect, tdd_engineer, documentation).

Acceptance Criteria: - All events in `events.jsonl` include `agent_id` and `agent_role` when operating in multi-agent mode - Single-agent mode remains backward-compatible (fields optional, default to `agent_id: "primary"`) - Agent role registry is a simple YAML list; no role hierarchy or inheritance required - Agent identity is self-declared, not centrally assigned — the event log is the source of truth for which agents have participated

Traces To: Asset Graph Model §7.4 (Event Sourcing — every event carries full provenance) | Ontology #7 (Markov object — agent boundary is its emitted events)

REQ-COORD-002: Feature Assignment via Events

Priority: High | **Phase**: 2

Feature/edge assignment SHALL be managed entirely through the event log. No lock files, mutex, or external coordination state. An agent claims work by emitting an `edge_claim` event; a single-writer serialiser resolves the claim into an `edge_started` (granted) or `claim_rejected` (conflict) event in the canonical log. The agent releases by emitting `edge_converged` or `edge_released`. Current assignments are a derived projection from the event log.

Acceptance Criteria: - An agent proposes work by emitting an `edge_claim` event with `agent_id`, `agent_role`, `feature`, and `edge` - The system serialises claims into the canonical event log: first claim wins → `edge_started`; conflicting claim → `claim_rejected` with reason and holding agent - In single-agent mode, the agent is the sole writer and may emit `edge_started` directly (no claim step needed) - Assignment state is never stored outside the event log — it is always derivable by replaying events - Stale claims (no follow-up event from the claiming agent within a configurable timeout) are detectable via status/health and result in a `claim_expired` telemetry signal - No lock files, lease files, or filesystem-level locking mechanisms - Inbox directories (if used for staging) are non-authoritative write buffers — their contents are transient and may be discarded without data loss once processed into the event log

Traces To: Asset Graph Model §7.4 (Event Sourcing — single source of truth), §7.6.3 (Markov boundary — agents interact through converged assets, not internal state) | Ontology #8 (Markov blanket — conditional independence through boundaries)

REQ-COORD-003: Work Isolation

Priority: High | **Phase**: 2

Agent working state (drafts, reasoning traces, session context) SHALL be isolated from shared project state. Only converged assets and events enter the shared workspace. Promotion from agent-private space to shared state passes through an evaluator gate.

Acceptance Criteria: - Each agent has a private working directory for in-progress artifacts - Shared project state (features, events, spec, design) is never written directly by agents during iteration — only via event emission and convergence promotion - Promotion of agent-produced assets to shared paths requires passing the configured evaluators for that edge (including human review where configured) - Agent-private state is ephemeral: if an agent crashes, only its emitted events persist; its working state may be discarded - Spec mutations (changes to requirements) always require human evaluator approval regardless of agent role

Traces To: Asset Graph Model §7.6.3 (Markov boundary — converged boundary is the interaction surface), §2.6 (Spec/Design boundary — spec is the constraint surface, not a working document) | Ontology #7 (Markov object stability), #40 (encoded representation)

REQ-COORD-004: Markov-Aligned Parallelism

Priority: Medium | **Phase**: 2

The system SHALL use the inner product of feature vectors (shared module count) to determine safe parallelism. Features with zero inner product MAY be assigned to different agents with no coordination. Features with non-zero inner product MUST build shared modules before diverging.

Acceptance Criteria: - Feature assignment (via `/start` or equivalent) considers inner product when routing agents to features - Orthogonal features (zero shared modules) are freely assignable to parallel agents - Non-orthogonal features trigger a warning or sequencing constraint when assigned concurrently - The inner product computation uses the module dependency DAG from build decomposition, not a heuristic

Traces To: Asset Graph Model §6.7 (Basis Projections — orthogonal projections execute in parallel), §11.1 (Inner product — shared modules determine coordination need) | Ontology #45 (superposition — concurrent vectors), #8 (Markov blanket — independence through boundaries)

REQ-COORD-005: Role-Based Evaluator Authority

Priority: Medium | **Phase**: 2

Agent roles SHALL define which edge types an agent may converge autonomously. Convergence of edges outside an agent's declared role authority requires escalation to a human evaluator or an agent with the appropriate role.

Acceptance Criteria: - The role registry maps each role to a set of edge types it may converge (e.g., `tdd_engineer: [code_unit_tests]`, `architect: [requirements_design, design_code]`) - An agent attempting to converge an edge outside its authority triggers an automatic escalation event (`convergence_escalated`) - Human evaluator authority is universal — a human may converge any edge - Role definitions are project-configurable, not hard-coded

Traces To: Asset Graph Model §4 (Evaluators — convergence as composition of evaluator types), §7.6.3 (Markov boundary — authority scoping is a boundary property) | Ontology #7 (Markov object — boundary defines what passes through)

13. Supervision (IntentEngine)

REQ-SUPV-001: IntentEngine Interface

Priority: High | **Phase**: 1

Every actor implementation (agents, monitors, hooks, observers) shall expose the IntentEngine interface: `observer → evaluator → typed_output(reflex.log | specEventLog | escalate)`, parameterised by intent+affect. This is not a fifth primitive — it is the composition law by which the four primitives (Graph, Iterate, Evaluators, Spec+Context) compose into a universal processing unit at every edge and every scale.

Acceptance Criteria: - Every edge traversal produces a classified observation (observer output) and a typed routing decision (evaluator output) - The evaluator classifies ambiguity into exactly one of three regimes: zero (reflex), bounded nonzero (probabilistic disambiguation), persistent (escalate to higher consciousness) - The typed output is exactly one of: `reflex.log` (fire-and-forget event), `specEventLog` (deferred intent for further processing), `escalate` (push to higher consciousness level) - The three output types map to existing event types in `events.jsonl` — no new event types required - Affect (urgency, valence) is carried as input alongside intent and propagates through chained IntentEngine invocations - The IntentEngine pattern applies at every scale: single iteration, edge convergence, feature traversal, sensory monitoring, production homeostasis, spec review - Level N's `escalate` output becomes Level N+1's reflex input (consciousness-as-relative) - Level N's `reflex.log` is invisible to Level N+1 (handled at origin)

Traces To: Asset Graph Model §4.6 (The IntentEngine), §4.6.2 (Ambiguity Classification), §4.6.3 (Three Output Types), §4.6.6 (Consciousness as Relative) | Ontology #49 (teleodynamic — fractal self-repair), #23 (scale-dependent observation), #9 (constraint manifold — ambiguity as constraint density)

REQ-SUPV-002: Constraint Tolerances

Priority: High | **Phase**: 1

Every constraint in the system — spec requirements, design bindings, edge evaluator thresholds, operational SLAs — shall have an associated measurable tolerance. Tolerances are the mechanism that produces `delta > 0` in the gradient (`delta(state, constraints) → work`). Without tolerances, the sensory system has nothing to measure, the IntentEngine has nothing to classify, and the gradient is undefined. A constraint without a tolerance is inert.

Acceptance Criteria: - Every constraint surface (spec, design, edge config, operational) expresses constraints as measurable thresholds, not qualitative assertions - Sensory monitors (§4.5) use tolerances as their evaluation criteria — the monitor observes a metric, the tolerance defines the threshold, the IntentEngine classifies the delta - Tolerance breach produces a classified signal: within bounds (reflex.log), drifting (specEventLog — optimization intent deferred), breached (escalate — corrective intent raised) - Design-level technology bindings (protocol choices, runtime selections, service models) declare tolerances for performance, cost, and fitness — breaches trigger optimization intents - Tolerances exist at every scale of the gradient: iteration, edge, feature, production, spec review, design binding - Tolerance pressure (complexity/cost breach → simplify) balances escalation pressure (ambiguity → add structure) to produce graph topology equilibrium - A constraint without a declared tolerance is flagged by gap analysis as incomplete

Traces To: Asset Graph Model §5.3 (Constraint Tolerances), §7.1 (The Gradient), §4.5 (Sensory Systems) | Ontology #49 (teleodynamic self-maintenance — tolerances enable self-repair), #9 (constraint manifold — tolerances make the manifold measurable)

REQ-SUPV-003: Failure Observability

Priority: High | **Phase**: 1

The system shall capture its own failures as structured events in `events.jsonl`. Without failure events, the delta function returns zero for broken subsystems — the LLM evaluator cannot design improvements for failures it cannot observe. Unobserved failure is invisible to homeostasis. This is the interoceptive prerequisite: the system must sense its own dysfunction to self-correct.

Acceptance Criteria: - **Evaluator failure detail**: When an evaluator check fails during `iterate()`, an `evaluator_detail` event is emitted with the check name, check type (F_D/F_P/F_H), the expected vs observed values, and the consecutive failure count for that check on that edge. This enables pattern detection: “check X has failed 4 consecutive iterations” is a signal that the requirement or evaluator needs revision, not just another iteration. - **Command error capture**: When any methodology command (`/gen-start`, `/gen-iterate`, `/gen-review`, etc.) encounters an error (missing config, invalid YAML, broken workspace state, unresolvable reference), a `command_error` event is emitted with the command name, error category, error detail, and workspace state at time of failure. This enables the system to detect tooling dysfunction patterns. - **Health check event**: When `/gen-status --health` runs, a `health_checked` event is emitted with the full check results (passed count, failed count, failed check names, recommendations). This enables trending: “bootloader missing for 3 consecutive health checks” or “same orphaned spawn persists across 5 checks” become visible patterns. - **Session abandonment detection**: When a methodology session ends without completing an in-progress iteration (no `edge_converged` or `iteration_completed` after `edge_started`), the next session detects the gap and emits an `iteration_abandoned` event with the feature, edge, last iteration number, and time since last event. This enables dropout pattern analysis. - All failure events use the same `events.jsonl` append-only log and follow the common event schema (event_type, timestamp, project) - Failure events are inputs to the same `observer → evaluator → typed_output` pipeline as success events — they can generate `intent_raised` events when patterns indicate systemic issues

Traces To: Asset Graph Model §4.6 (IntentEngine — observer requires observable failures), §7.6 (Self-Observation), Genesis Bootloader §VI (Homeostasis — intent is computed from delta, delta requires observation of failures) | Ontology #49 (teleodynamic self-maintenance requires sensing dysfunction, not just health)

Requirement Summary

Category	Count	Critical	High	Medium
Intent & Spec	4	1	2	1
Asset Graph	3	2	1	0
Iteration Engine	3	2	0	1
Evaluators	3	2	1	0
Context	2	0	1	1
Feature Vectors	3	1	2	0
Full Lifecycle	12	2	10	0
Sensory Systems	6	0	5	1
Edge Parameterisations	4	0	4	0
Tooling	14	0	10	4
User Experience	7	0	5	2
Multi-Agent Coordination	5	0	3	2
Supervision (IntentEngine)	3	0	3	0
Total	**69**	**10**	**47**	**12**

Phase 1 (Core Graph Engine): 45 requirements

Intent capture + spec, graph topology, iteration engine, evaluators, context, feature vectors, edge parameterisations, tooling (including installability), gradient mechanics (intent events, signal classification, spec change events, protocol enforcement, spec review as gradient check), sensory (artifact write observation), user experience (state-driven routing, progressive disclosure, observability, feature/edge selection, recovery, human gate awareness, edge zoom management), supervision (IntentEngine interface, constraint tolerances).

Phase 2 (Full Lifecycle + Coordination): 20 requirements

Eco-intent, context hierarchy, CI/CD edges, telemetry/homeostasis, feedback loop closure, feature lineage in telemetry, dev observer agent, CI/CD observer agent, ops observer agent, interoceptive monitoring, exteroceptive monitoring, affect triage pipeline, sensory configuration, review boundary, agent identity, event-sourced assignment, work isolation, Markov-aligned parallelism, role-based evaluator authority, functor encoding tracking.

Traces To: AI_SDLC_ASSET_GRAPH_MODEL.md (v2.8.0) — every requirement anchored to a specific section. **Parent Theory**: Constraint-Emergence Ontology — concept numbers (#N) throughout.