AI SDLC — UAT Test Cases

Version: 1.0.0 **Date**: 2026-02-22 **Derived From**:
AISDLC_IMPLEMENTATION_REQUIREMENTS.md (v3.11.0),
FEATURE_VECTORS.md (v1.8.0) **Method**: Asset Graph Model §9.2 — BDD (Given/When/Then), business language only

---

Purpose & Approach

This document specifies **exhaustive functional use cases** for the AI SDLC Asset Graph Model. Every scenario is:

1. **Written in business language** (Given/When/Then) — no platform-specific syntax
2. **Traced to REQ keys** — every scenario carries `Validates: REQ-*` tags
3. **Event-verifiable** — expected outcomes verifiable via `events.jsonl`
4. **Fixture-based** — scenarios reference one of 5 base workspace fixtures
5. **Gap-annotated** — marked NEW (no existing coverage) or EXISTING (covered by which test)

This is a **spec-level artifact** — WHAT to test, not HOW. Platform-specific executable tests belong in `imp_<name>/tests/uat/`.

Audience

Implementers building AI SDLC tooling for any platform (Claude, Gemini, Codex, or future implementations).

Coverage Matrix

All 63 implementation requirements mapped to use case clusters and scenario counts.

REQ Key	Description	UC Cluster	Scenarios
REQ-INTENT-001	Intent Capture	UC-04	3
REQ-INTENT-002	Intent as Spec	UC-04	2

REQ Key	Description	UC Cluster	Scenarios
REQ-INTENT-003	Eco-Intent Generation	UC-06	2
REQ-INTENT-004	Spec Reproducibility	UC-03	3
REQ-GRAPH-001	Asset Type Registry	UC-01	3
REQ-GRAPH-002	Admissible Transitions	UC-01	4
REQ-GRAPH-003	Asset as Markov Object	UC-01	3
REQ-ITER-001	Universal Iteration Function	UC-01	4
REQ-ITER-002	Convergence and Promotion	UC-01	5
REQ-ITER-003	Functor Encoding Tracking	UC-01	5
REQ-EVAL-001	Three Evaluator Types	UC-02	4
REQ-EVAL-002	Evaluator Composition Per Edge	UC-02	4
REQ-EVAL-003	Human Accountability	UC-02	7
REQ-CTX-001	Context as Constraint Surface	UC-03	4
REQ-CTX-002	Context Hierarchy	UC-03	5
REQ-FEAT-001	Feature Vector Trajectories	UC-04	4
REQ-FEAT-002	Feature Dependencies	UC-04	4
REQ-FEAT-003	Task Planning as Trajectory Optimisation	UC-04	5
REQ-EDGE-001	TDD at Code/Tests Edges	UC-05	5
REQ-EDGE-002	BDD at Design/Test Edges	UC-05	4
REQ-EDGE-003	ADRs at Requirements/Design Edge	UC-05	4
REQ-EDGE-004	Code Tagging	UC-05	3
REQ-LIFE-001	CI/CD as Graph Edge	UC-06	3
REQ-LIFE-002	Telemetry and Homeostasis	UC-06	3
REQ-LIFE-003	Feedback Loop Closure	UC-06	3

REQ Key	Description	UC Cluster	Scenarios
REQ-LIFE-004	Feature Lineage in Telemetry	UC-06	2
REQ-LIFE-005	Intent Events as First-Class Objects	UC-06	3
REQ-LIFE-006	Signal Source Classification	UC-06	3
REQ-LIFE-007	Spec Change Events	UC-06	3
REQ-LIFE-008	Protocol Enforcement Hooks	UC-06	3
REQ-LIFE-009	Spec Review as Gradient Check	UC-06	2
REQ-LIFE-010	Development Observer Agent	UC-06	2
REQ-LIFE-011	CI/CD Observer Agent	UC-06	2
REQ-LIFE-012	Ops Observer Agent	UC-06	2
REQ-SENSE-001	Interoceptive Monitoring	UC-07	4
REQ-SENSE-002	Exteroceptive Monitoring	UC-07	3
REQ-SENSE-003	Affect Triage Pipeline	UC-07	4
REQ-SENSE-004	Sensory System Configuration	UC-07	2
REQ-SENSE-005	Review Boundary	UC-07	3
REQ-TOOL-001	Plugin Architecture	UC-08	2
REQ-TOOL-002	Developer Workspace	UC-08	3
REQ-TOOL-003	Workflow Commands	UC-08	3
REQ-TOOL-004	Release Management	UC-08	3
REQ-TOOL-005	Test Gap Analysis	UC-08	3
REQ-TOOL-006	Methodology Hooks	UC-08	2
REQ-TOOL-007	Project Scaffolding	UC-08	2
REQ-TOOL-008	Context Snapshot	UC-08	2
REQ-TOOL-009	Feature Views	UC-08	2
REQ-TOOL-010	Spec/Design Boundary Enforcement	UC-08	2
REQ-UX-001	State-Driven Routing	UC-09	8
REQ-UX-002	Progressive Disclosure	UC-09	4

REQ Key	Description	UC Cluster	Scenarios
REQ-UX-003	Project-Wide Observability	UC-09	4
REQ-UX-004	Automatic Feature/Edge Selection	UC-09	4
REQ-UX-005	Recovery and Self-Healing	UC-09	4
REQ-UX-006	Human Gate Awareness	UC-09	4
REQ-UX-007	Edge Zoom Management	UC-09	4
REQ-COORD-001	Agent Identity	UC-10	3
REQ-COORD-002	Feature Assignment via Events	UC-10	4
REQ-COORD-003	Work Isolation	UC-10	3
REQ-COORD-004	Markov-Aligned Parallelism	UC-10	3
REQ-COORD-005	Role-Based Evaluator Authority	UC-10	3
REQ-SUPV-001	IntentEngine Interface	UC-11	7
REQ-SUPV-002	Constraint Tolerances	UC-11	7

Total: 64 REQ keys, 220 scenarios.

Test Fixture Strategy

All scenarios reference one of 5 base workspace fixtures. Each fixture represents a reproducible workspace state.

Fixture 1: CLEAN

An empty directory with no `.ai-workspace/`. Represents a greenfield project.

Contents: Project source files only (e.g., `pyproject.toml`, `src/`). No methodology artifacts.

Fixture 2: INITIALIZED

A freshly initialized workspace. `project_initialized` event emitted. No features created yet.

Contents: - `.ai-workspace/` with graph topology, profiles, templates, project constraints - `specification/INTENT.md` with a concrete intent - `events.jsonl` with one `project_initialized` event - No feature vectors in `features/active/`

Fixture 3: IN_PROGRESS

A workspace with 3 active features at various stages of progress.

Contents: - Everything in INITIALIZED - Feature A (`REQ-F-ALPHA-001`): converged through `intent→requirements`, iterating on `requirements→design` - Feature B (`REQ-F-BETA-001`): converged through `design→code`, iterating on `code↔unit_tests` (iteration 3, delta=2) - Feature C (`REQ-F-GAMMA-001`): just started, at `intent→requirements` edge (iteration 1) - `events.jsonl` with ~15 events (edge_started, iteration_completed, edge_converged) - Standard profile active

Fixture 4: CONVERGED

A workspace where all features have completed all profile edges.

Contents: - Everything in INITIALIZED - 2 features, both fully converged (all edges in standard profile) - `events.jsonl` with complete trajectory (edge_started → iteration_completed → edge_converged for each edge) - All assets tagged with REQ keys

Fixture 5: STUCK

A workspace with features exhibiting stuck and blocked conditions.

Contents: - Everything in INITIALIZED - Feature X: same delta (3 failing checks) for 4 consecutive iterations on `code↔unit_tests` - Feature Y: blocked — depends on spawn `REQ-F-SPIKE-001` which has not converged - Feature Z: blocked — human review pending on `requirements→design` - `events.jsonl` with iteration events showing unchanging delta for Feature X

UC-01: Asset Graph Engine

Feature Vector: REQ-F-ENGINE-001 **Satisfies**: REQ-GRAPH-001, REQ-GRAPH-002, REQ-GRAPH-003, REQ-ITER-001, REQ-ITER-002, REQ-ITER-003

UC-01-01: Asset type registry contains all default types

`Validates: REQ-GRAPH-001` | Fixture: INITIALIZED | **EXISTING** (TestGraphTopology)

```
Given an initialized workspace
When I inspect the graph topology configuration
Then I find exactly 10 asset types registered:
  Intent, Requirements, Design, Code, Unit Tests, Test Cases, UAT
Tests, CI/CD, Running System, Telemetry
```

```
And each asset type has a schema definition
And each asset type has Markov criteria for promotion
```

UC-01-02: Asset type registry is extensible

Validates: REQ-GRAPH-001 | Fixture: INITIALIZED | **NEW**

```
Given an initialized workspace with the default 10 asset types
When I add a custom asset type "security_audit" to the graph topology
  with schema fields [audit_id, findings, severity_counts]
  and Markov criteria [has_findings, human_approved]
Then the graph engine accepts the extended topology
And I can define transitions involving "security_audit"
And existing transitions are unaffected
```

UC-01-03: Asset type interfaces are typed

Validates: REQ-GRAPH-001 | Fixture: INITIALIZED | **NEW**

```
Given an initialized workspace
When I create an asset of type "requirements"
  missing the mandatory field "acceptance_criteria"
Then the asset fails Markov criteria validation
And the failure report identifies the missing field
```

UC-01-04: Only admissible transitions can be traversed

Validates: REQ-GRAPH-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a workspace with Feature A at the "requirements" stage
When I attempt to iterate on the edge "requirements→unit_tests"
  which is not in the admissible transitions
Then the system rejects the traversal
And reports "No admissible transition from requirements to unit_tests"
```

UC-01-05: Feedback loop edge makes the graph cyclic

Validates: REQ-GRAPH-002 | Fixture: INITIALIZED | **EXISTING**
(TestGraphTopology)

```
Given an initialized workspace
When I inspect the graph topology
Then I find a transition "Telemetry → Intent (Feedback Loop)"
And the graph is marked as cyclic: true
And the feedback loop edge has evaluators [agent, human]
```

UC-01-06: Transition registry is extensible

Validates: REQ-GRAPH-002 | Fixture: INITIALIZED | **NEW**

```
Given an initialized workspace with 10 default transitions
When I add a new transition "design→security_audit"
  with evaluators [agent, human]
  and constructor: agent
Then the graph topology contains 11 transitions
And the new transition is traversable by the iterate function
And all transitions are logged for audit
```

UC-01-07: Non-converged assets are candidates, not Markov objects

`Validates: REQ-GRAPH-003` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B iterating on "code↔unit_tests" with delta=2
When I inspect Feature B's code asset status
Then the asset is marked as "candidate" (not "stable")
And the asset cannot be promoted to the next graph node
And the 2 failing evaluator checks are listed with remediation
guidance
```

UC-01-08: Converged asset achieves Markov object status

`Validates: REQ-GRAPH-003` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A whose "requirements" asset has delta=0
  with all evaluators reporting convergence
When I inspect Feature A's requirements asset status
Then the asset is marked as "stable" (Markov object)
And the asset is usable without its construction history
And the asset's typed interface is valid
```

UC-01-09: Markov object is usable without construction history

`Validates: REQ-GRAPH-003` | Fixture: CONVERGED | **NEW**

```
Given a converged feature with a stable "design" asset
When a new feature references Feature A's design as context
Then the new feature can consume the design asset
  without access to Feature A's iteration history
And the design asset's typed interface is sufficient for downstream
use
```

UC-01-10: Iterate function has universal signature

`Validates: REQ-ITER-001` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at the "requirements→design" edge
When I invoke iterate(Asset<requirements>, Context[],
Evaluators(requirements_design))
Then the function returns Asset<design.k+1> (next candidate)
```

```
And the same function signature works for all 10 edge types
And the asset carries intent lineage and REQ keys
```

UC-01-11: Iterate behaviour is parameterised by edge config

`Validates: REQ-ITER-001` | Fixture: IN_PROGRESS | **NEW**

```
Given two features at different edges:
  Feature A at "requirements→design" (evaluators: agent, human)
  Feature B at "code↔unit_tests" (evaluators: agent, deterministic)
When I invoke iterate on each feature
Then Feature A's iteration uses ADR generation pattern
And Feature B's iteration uses TDD co-evolution pattern
And both use the same iterate function with different edge configs
```

UC-01-12: Iteration repeats until convergence

`Validates: REQ-ITER-001` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B at "code↔unit_tests" with delta=2
When I invoke iterate and the agent produces code that passes 1 more
check
Then delta decreases to 1
And the feature remains at the same edge
And iterate can be invoked again to reduce delta further
And convergence is only declared when delta=0
```

UC-01-13: Constructor is edge-specific

`Validates: REQ-ITER-001` | Fixture: INITIALIZED | **EXISTING** (TestEdgeConfigs)

```
Given the graph topology with 10 transitions
When I inspect the constructor field for each transition
Then "intent→requirements" uses constructor: agent
And "code→cicd" uses constructor: deterministic
And "code↔unit_tests" uses constructor: agent
And each constructor type is appropriate for its edge
```

UC-01-14: Convergence requires all evaluators to pass

`Validates: REQ-ITER-002` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B at "code↔unit_tests" with 3 evaluator checks
When 2 of 3 required checks pass but 1 fails
Then delta = 1
And the feature is NOT promoted to the next asset type
And the iteration report shows exactly which check failed
```

UC-01-15: Convergence threshold is configurable per evaluator

Validates: REQ-ITER-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a feature with a coverage threshold of 80% (project default)
And the feature vector overrides coverage threshold to 95%
When the code achieves 85% coverage
Then the project-level check would pass (85% > 80%)
But the feature-level check fails (85% < 95%)
And the effective threshold is 95% (feature override wins)
```

UC-01-16: Promotion creates next asset type version

Validates: REQ-ITER-002 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "requirements→design" with delta=0
When all evaluators report convergence (including human approval)
Then the requirements asset is promoted: ATn.m becomes ATn+1.0
And a new design asset (version 0) is created
And an "edge_converged" event is emitted
And the feature trajectory advances to the next edge
```

UC-01-17: Non-convergence after max iterations escalates to human

Validates: REQ-ITER-002 | Fixture: STUCK | **NEW**

```
Given Feature X with delta=3 unchanged for 4 iterations
And the iteration budget is 5 iterations before escalation
When the 5th iteration also produces delta=3
Then the system escalates to the human evaluator
And the escalation includes: which checks are stuck, what has been
tried
And the human can: force-converge, spawn discovery, or continue
iterating
```

UC-01-18: Extended convergence for discovery vectors (question answered)

Validates: REQ-ITER-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a discovery vector investigating "Can technology X handle load
Y?"
And the vector has delta=1 (one non-required check remains)
When the human evaluator judges "Yes, the question has been answered"
Then the vector converges with convergence_type: "question_answered"
And the findings are packaged as fold-back payload
And a "spawn_folded_back" event is emitted to the parent feature
```

UC-01-19: Profile encoding section defines functor categories

Validates: REQ-ITER-003 | Fixture: INITIALIZED | **EXISTING** (TestFunctorEncoding)

```
Given an initialized workspace with the standard profile
When I inspect the profile encoding section
Then I find strategy: "balanced", mode: "interactive", valence:
"medium"
And 8 functional units with category mappings:
  evaluate: F_D, construct: F_P, classify: F_D, route: F_H,
  propose: F_P, sense: F_D, emit: F_D, decide: F_H
```

UC-01-20: Feature vectors carry functor encoding

Validates: REQ-ITER-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B iterating on "code↔unit_tests"
When I inspect the iteration event for Feature B
Then the event contains an "encoding" section with:
  encoding_source (profile name), mode, valence, and active_units
And the active_units map all 8 functional units to categories
```

UC-01-21: Category-fixed units are enforced

Validates: REQ-ITER-003 | Fixture: INITIALIZED | **NEW**

```
Given a custom profile that attempts to set emit: F_P (overriding the
fixed category)
When the profile is validated
Then the system rejects the override for "emit" (must be F_D)
And the system rejects any override for "decide" (must be F_H)
And other functional units accept category overrides
```

UC-01-22: Encoding escalation emits event

Validates: REQ-ITER-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B on "code↔unit_tests" with evaluate: F_D
(deterministic)
When the deterministic evaluator cannot classify an ambiguous test
failure
And the system escalates evaluate from F_D to F_P (probabilistic
agent)
Then an "encoding_escalated" event is emitted with:
  functional_unit: "evaluate", from_category: "F_D", to_category:
"F_P"
And the trigger reason is recorded
```

UC-01-23: Escalation trajectory is recorded per feature

Validates: REQ-ITER-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B with 2 encoding escalations during its trajectory
When I inspect the feature vector's trajectory
Then each trajectory entry supports an "escalations" array
And the escalations record: which unit, from/to category, iteration
number
And the escalation history is available for trajectory analysis
```

UC-02: Evaluator Framework

Feature Vector: REQ-F-EVAL-001 **Satisfies**: REQ-EVAL-001, REQ-EVAL-002, REQ-EVAL-003

UC-02-01: Human evaluator captures judgment

Validates: REQ-EVAL-001 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "requirements→design" edge which requires human
evaluation
When the iterate function presents the design candidate for review
Then the human can: approve, reject, or provide refinement guidance
And the decision is recorded in the iteration history
And a "review_completed" event is emitted with the human's feedback
```

UC-02-02: Agent evaluator computes delta via LLM

Validates: REQ-EVAL-001 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B at "code↔unit_tests" with agent evaluator configured
When the agent evaluator assesses the code candidate
Then it computes delta between current state and target state
And provides specific, actionable feedback for the next iteration
And does NOT declare convergence if known gaps exist
```

UC-02-03: Deterministic evaluator produces binary pass/fail

Validates: REQ-EVAL-001 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B at "code↔unit_tests" with deterministic evaluator
"tests_pass"
When the test runner executes and 3 of 5 tests fail
Then the deterministic evaluator reports: FAIL
And identifies the 3 failing tests with their error messages
And provides remediation guidance for each failure
```

UC-02-04: Evaluators declare processing phase

Validates: REQ-EVAL-001 | Fixture: INITIALIZED | **EXISTING** (TestProcessingPhases)

```
Given the evaluator configuration for all edge types
When I inspect the processing_phase field for each evaluator type
Then Deterministic Tests are classified as "reflex" (autonomic)
And Human and Agent deliberative are classified as "conscious"
(deliberative)
And Affect is a valence vector emitted by ANY evaluator on its gap
finding (not an evaluator type assignment)
```

UC-02-05: Evaluator composition is configurable per edge

Validates: REQ-EVAL-002 | Fixture: INITIALIZED | **EXISTING** (TestEdgeConfigs)

```
Given the graph topology with 10 transitions
When I inspect the evaluator composition for each edge
Then "intent→requirements" uses [agent, human]
And "code↔unit_tests" uses [agent, deterministic]
And "code→cicd" uses [deterministic] only
And each composition is read from config, not hard-coded
```

UC-02-06: Evaluator composition overridable at project level

Validates: REQ-EVAL-002 | Fixture: INITIALIZED | **NEW**

```
Given the default "design→code" edge with evaluators [agent,
deterministic]
When the project overrides this edge to [agent, deterministic, human]
Then the iterate function uses 3 evaluator types for this edge
And convergence requires all 3 to pass
And the override is stored in project configuration, not the graph
topology
```

UC-02-07: Profile overrides evaluator composition

Validates: REQ-EVAL-002 | Fixture: INITIALIZED | **EXISTING** (TestProfiles)

```
Given the standard profile with evaluator overrides:
  "intent→requirements": [agent, human]
  "requirements→design": [agent, human]
When a feature using the standard profile reaches
"intent→requirements"
Then the iterate function uses [agent, human] evaluators (not the
graph default)
And the override source is the profile, not the edge default
```

UC-02-08: Default compositions match spec table

Validates: REQ-EVAL-002 | Fixture: INITIALIZED | **EXISTING** (TestEvaluatorDefaults)

```
Given the graph topology configuration
When I compare evaluator compositions against the Asset Graph Model §4.2 table
Then every edge matches the specified default composition
And the composition is complete (no edges missing evaluator assignments)
```

UC-02-09: AI suggestions require human review at human-configured edges

Validates: REQ-EVAL-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "requirements→design" with evaluators [agent, human]
When the agent evaluator reports delta=0 (all agent checks pass)
Then the feature does NOT auto-converge
And the system presents the candidate for human review
And convergence is blocked until the human explicitly approves
```

UC-02-10: Human can override any agent decision

Validates: REQ-EVAL-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "requirements→design" where the agent recommends convergence
When the human reviews and disagrees with the agent's assessment
And the human provides refinement feedback
Then the human's rejection overrides the agent's approval
And iteration continues with the human's feedback as guidance
And the rejection is attributed to the human, not the AI
```

UC-02-11: Decisions attributed to humans, not AI

Validates: REQ-EVAL-003 | Fixture: IN_PROGRESS | **NEW**

```
Given a "review_completed" event emitted after human evaluation
When I inspect the event
Then the decision is attributed to the human reviewer
And the event records: who decided, what they decided, and their feedback
And the AI's prior recommendation is logged but not authoritative
```

UC-02-12: Human evaluator override always available

Validates: REQ-EVAL-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B at "code↔unit_tests" with evaluators [agent,
deterministic]
  where human evaluator is NOT in the default composition
When the user requests a human review on this edge
Then the system allows the human to review and provide feedback
And the human's feedback is recorded in the iteration history
And the human override capability is always available regardless of
edge config
```

UC-02-13: Spec mutations always require human approval

`Validates: REQ-EVAL-003` | Fixture: IN_PROGRESS | **NEW**

```
Given an agent evaluator that detects a spec gap during iteration
When the agent proposes a specification modification
Then the modification is presented as a draft to the human
And the specification is NOT modified until the human approves
And the approval is recorded as a "spec_modified" event with human
attribution
```

UC-02-14: Human approval required at human-required edges before convergence

`Validates: REQ-EVAL-003` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "requirements→design" where human_required: true
When all agent and deterministic evaluators report delta=0
Then convergence status shows "PENDING_HUMAN_REVIEW"
And the edge does NOT converge until human explicitly approves
And auto-mode pauses at this gate
```

UC-02-15: Human review records feedback in iteration history

`Validates: REQ-EVAL-003` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "requirements→design" pending human review
When the human provides refinement guidance: "Add error handling for
edge case X"
Then the feedback is stored in the feature's iteration history
And the next iteration uses this feedback as construction guidance
And the feedback is traceable in the event log
```

UC-03: Context Management

Feature Vector: REQ-F-CTX-001 **Satisfies**: REQ-CTX-001, REQ-CTX-002, REQ-INTENT-004

UC-03-01: Context types include required categories

Validates: REQ-CTX-001 | Fixture: INITIALIZED | **EXISTING** (TestContextSources)

```
Given an initialized workspace
When I inspect the context directory structure
Then I find directories for: ADRs, data models, templates, policy,
standards
And the context system accepts open-ended context types
And context elements are version-controlled in the workspace
```

UC-03-02: Context narrows admissible constructions

Validates: REQ-CTX-001 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "requirements→design" edge
And Context[] contains an ADR mandating "PostgreSQL for persistence"
When the iterate function generates a design candidate
Then the design references PostgreSQL (not an arbitrary database)
And the ADR constraint narrowed the space of admissible designs
And the context element is recorded in the iteration's context hash
```

UC-03-03: Context subset configurable per edge

Validates: REQ-CTX-001 | Fixture: INITIALIZED | **NEW**

```
Given a workspace with 5 context elements:
  ADR-001, DataModel-User, Template-API, Policy-Security, Standard-
Naming
When iterating on "design→code" edge
Then the iterate function loads only the context elements relevant to
this edge
And the relevance is configurable (not all context applies to all
edges)
And the loaded context is recorded in the context hash
```

UC-03-04: Context version control

Validates: REQ-CTX-001 | Fixture: INITIALIZED | **NEW**

```
Given a workspace with ADR-001 v1 in Context[]
When ADR-001 is updated to v2 (changed decision)
Then the context system preserves v1 (immutable)
And subsequent iterations use v2
And the context hash changes to reflect the new version
And the version history is traceable
```

UC-03-05: Context hierarchy — global to project override

Validates: REQ-CTX-002 | Fixture: INITIALIZED | **NEW**

```
Given a context hierarchy:
  Global: {coverage_threshold: 80%, style: "PEP8"}
  Organisation: {coverage_threshold: 90%}
  Project: {style: "Google Python Style"}
When the iterate function resolves the effective context
Then coverage_threshold = 90% (organisation overrides global)
And style = "Google Python Style" (project overrides global)
And unoverridden values inherit from parent levels
```

UC-03-06: Context deep merge for objects

Validates: REQ-CTX-002 | Fixture: INITIALIZED | **NEW**

```
Given a global context with deployment: {cloud: "AWS", region: "us-
east-1"}
And a project context with deployment: {region: "eu-west-1"}
When the iterate function resolves the effective context
Then deployment = {cloud: "AWS", region: "eu-west-1"} (deep merge)
And the cloud field is inherited, the region field is overridden
```

UC-03-07: Later context overrides earlier

Validates: REQ-CTX-002 | Fixture: INITIALIZED | **NEW**

```
Given context loaded in order: global → organisation → team → project
When the same key appears at multiple levels
Then the project value wins (last loaded)
And the override chain is traceable for audit
```

UC-03-08: Customisation without forking

Validates: REQ-CTX-002 | Fixture: INITIALIZED | **NEW**

```
Given a team-level context with standard evaluator thresholds
When a project needs stricter coverage (95% instead of 80%)
Then the project overrides only the coverage threshold
And all other team-level context is inherited unchanged
And the team context is not modified (no fork)
```

UC-03-09: Context hierarchy supports four levels

Validates: REQ-CTX-002 | Fixture: INITIALIZED | **NEW**

```
Given context defined at all four levels: global, organisation, team,
project
When I request the effective context for a feature
Then all four levels are composed in order
And conflicts resolve by the "later overrides earlier" rule
And the composition is deterministic (same inputs = same output)
```

UC-03-10: Spec canonical serialisation is deterministic

`Validates: REQ-INTENT-004` | Fixture: IN_PROGRESS | **NEW**

```
Given the same Intent + Context[] provided twice
When the spec is canonically serialised each time
Then both serialisations produce identical byte sequences
And the content-addressable hash is the same
And independent tools would compute the same hash
```

UC-03-11: Spec hash recorded at each iteration

`Validates: REQ-INTENT-004` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B on "code↔unit_tests" at iteration 3
When I inspect the iteration_completed event
Then the event contains a "context_hash" field with "sha256:..."
format
And the hash changes if and only if the effective context changes
```

UC-03-12: Spec versions are immutable

`Validates: REQ-INTENT-004` | Fixture: IN_PROGRESS | **NEW**

```
Given a spec version V1 with hash H1
When a requirement is modified (creating spec version V2 with hash H2)
Then V1 remains accessible with its original hash H1
And V2 is a new version, not a mutation of V1
And the evolution from V1 to V2 is traceable via "spec_modified"
events
```

UC-04: Feature Vector Traceability

Feature Vector: REQ-F-TRACE-001 **Satisfies**: REQ-INTENT-001, REQ-INTENT-002, REQ-FEAT-001, REQ-FEAT-002, REQ-FEAT-003

UC-04-01: Intent captured in structured format

`Validates: REQ-INTENT-001` | Fixture: CLEAN | **NEW**

```
Given a clean project directory
When the user provides an intent: "Build a user authentication system"
Then an intent is created with:
  unique identifier (INT-001)
  description: "Build a user authentication system"
  source: "human"
  timestamp: current time
  priority: assigned by user
```

```
And the intent is persisted in specification/INTENT.md
And the intent is version-controlled
```

UC-04-02: Intent from runtime feedback

Validates: REQ-INTENT-001 | Fixture: CONVERGED | **NEW**

```
Given a converged project with running telemetry
When the ops observer detects a latency deviation
Then an intent is generated with:
  identifier: INT-002
  source: "runtime_feedback"
  description including the deviation details
And the intent enters the asset graph as a new feature vector
candidate
```

UC-04-03: Intent from ecosystem changes

Validates: REQ-INTENT-001 | Fixture: CONVERGED | **NEW**

```
Given a converged project using library X v2.0
When the exteroceptive monitor detects library X v2.0 has a critical
CVE
Then an intent is generated with:
  identifier: INT-ECO-001
  source: "ecosystem"
  description: "Critical CVE in library X v2.0"
And the intent enters the graph for human triage
```

UC-04-04: Intent composes with Context to form Spec

Validates: REQ-INTENT-002 | Fixture: IN_PROGRESS | **NEW**

```
Given INT-001 ("Build a user authentication system")
And Context[] containing: ADR-001 (OAuth2), DataModel-User, Policy-
Security
When the intent composes with context to form the spec
Then the spec is the fitness landscape against which evaluators
measure convergence
And the spec includes both the intent and all context constraints
And the spec evolves as intent lineage accumulates through traversal
```

UC-04-05: Spec is the fitness landscape for evaluators

Validates: REQ-INTENT-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a spec formed from INT-001 + Context[]
When evaluators assess a candidate design
Then the evaluators measure convergence against the spec
And delta is the distance between the candidate and the spec's
```

```
constraints
And the spec bounds what counts as "correct"
```

UC-04-06: REQ key format is enforced

`Validates: REQ-FEAT-001` | Fixture: INITIALIZED | **EXISTING** (TestReqKeyCoverage)

```
Given the requirement key format: REQ-{TYPE}-{DOMAIN}-{SEQ}
When I create requirements with valid keys (REQ-F-AUTH-001, REQ-NFR-
PERF-001)
Then the keys are accepted
And when I create a requirement with invalid key format ("AUTH-001")
Then the key is rejected with a format error
And REQ key types include: F, NFR, DATA, BR
```

UC-04-07: REQ keys propagate through all assets on trajectory

`Validates: REQ-FEAT-001` | Fixture: CONVERGED | **EXISTING** (TestEndToEndTraceability)

```
Given a converged feature REQ-F-AUTH-001
When I trace the REQ key through all artifacts
Then the spec references REQ-F-AUTH-001
And the design references REQ-F-AUTH-001
And the code contains "Implements: REQ-F-AUTH-001"
And the tests contain "Validates: REQ-F-AUTH-001"
And bidirectional navigation works: forward (intent→runtime) and
backward (runtime→intent)
```

UC-04-08: REQ key is immutable identifier

`Validates: REQ-FEAT-001` | Fixture: IN_PROGRESS | **NEW**

```
Given REQ-F-AUTH-001 assigned to a requirement
When the requirement text is modified (creating version REQ-F-AUTH-
001.1.0.0)
Then the key REQ-F-AUTH-001 remains the same (immutable trajectory
identifier)
And the version suffix tracks how the requirement statement evolves
And all artifacts continue referencing REQ-F-AUTH-001 (not the
versioned form)
```

UC-04-09: Bidirectional navigation works

`Validates: REQ-FEAT-001` | Fixture: CONVERGED | **NEW**

```
Given a converged feature REQ-F-AUTH-001 with artifacts at all stages
When I navigate forward from intent
Then I can trace: Intent → Requirements → Design → Code → Tests
```

```
And when I navigate backward from a test
Then I can trace: Test → Code → Design → Requirements → Intent
And the REQ key is the thread connecting all stages
```

UC-04-10: Cross-feature dependencies tracked

Validates: REQ-FEAT-002 | Fixture: IN_PROGRESS | **EXISTING** (TestFeatureVectorConsistency)

```
Given Feature A (REQ-F-AUTH-001) depends on Feature B (REQ-F-DB-001)
at the design stage
When I inspect Feature A's dependency graph
Then the dependency is recorded: A.design depends on B.design
And the dependency is between trajectories, not individual assets
And the dependency graph shows the relationship
```

UC-04-11: Circular dependencies detected

Validates: REQ-FEAT-002 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A depends on Feature B at the code stage
And Feature B depends on Feature A at the code stage
When the dependency graph is validated
Then the system detects the circular dependency
And flags it with both feature IDs and the stage where the cycle
occurs
And the cycle blocks concurrent iteration on both features
```

UC-04-12: Dependency graph is visualisable

Validates: REQ-FEAT-002 | Fixture: IN_PROGRESS | **NEW**

```
Given 3 features with dependencies: A→B, A→C (no dependency between B
and C)
When I request the dependency graph
Then the system produces a visual representation showing:
  A depends on B and C
  B and C are independent (parallelisable)
And the visualisation identifies the critical path
```

UC-04-13: Feature views show per-REQ cross-artifact status

Validates: REQ-FEAT-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a feature REQ-F-AUTH-001 with artifacts at spec, design, and
code stages
When I request the feature view for REQ-F-AUTH-001
Then I see which stages have the REQ key tagged:
  Spec: tagged, Design: tagged, Code: tagged, Tests: not yet,
Telemetry: not yet
```

```
And missing stages are flagged
And the coverage summary shows "3 of 5 stages tagged"
```

UC-04-14: Tasks emerge from feature vector decomposition

Validates: REQ-FEAT-003 | Fixture: INITIALIZED | **NEW**

```
Given an intent "Build a user authentication system"
When the system decomposes the intent into feature vectors
Then each feature vector maps to a trajectory through the asset graph
And the features are: REQ-F-AUTH-001, REQ-F-SESSION-001, REQ-F-RBAC-
001
And each feature has a defined path through graph edges
```

UC-04-15: Task graph identifies parallelisation opportunities

Validates: REQ-FEAT-003 | Fixture: IN_PROGRESS | **NEW**

```
Given 3 features with dependency graph: A→B, A→C, B and C independent
When the task graph is generated
Then B and C are marked as parallelisable (can execute concurrently)
And A must complete its dependency edge before B and C can proceed
And the compressed task graph shows the optimal execution order
```

UC-04-16: Task graph batches parallel edges

Validates: REQ-FEAT-003 | Fixture: IN_PROGRESS | **NEW**

```
Given 3 independent features all at the "design→code" edge
When the task graph is generated
Then the 3 design→code iterations are batched as parallel work
And dependencies at later edges are sequenced correctly
And the batch execution reduces total development time
```

UC-04-17: Inter-vector dependencies identified at each node

Validates: REQ-FEAT-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A's code depends on Feature B's code (shared module)
When the task graph maps trajectories through the asset graph
Then the dependency is identified at the "code" node specifically
And Feature A's code edge is sequenced after Feature B's code edge
And their design edges may still execute in parallel (no dependency at
design)
```

UC-04-18: Compression produces minimal task graph

Validates: REQ-FEAT-003 | Fixture: IN_PROGRESS | **NEW**

```
Given 5 features with various inter-dependencies
When the task graph is compressed
Then parallel-eligible edges are batched
And sequential dependencies are ordered
And the result is the minimum spanning task graph with dependency
order
And no unnecessary sequencing is introduced
```

UC-05: Edge Parameterisations

Feature Vector: REQ-F-EDGE-001 **Satisfies**: REQ-EDGE-001, REQ-EDGE-002, REQ-EDGE-003, REQ-EDGE-004

UC-05-01: TDD RED phase — write failing test first

`Validates: REQ-EDGE-001` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B at "code↔unit_tests" beginning a new TDD cycle
When the iterate function starts the TDD co-evolution
Then the first action is to write a failing test
And the test references "Validates: REQ-F-BETA-001"
And the deterministic evaluator confirms: test exists, test fails (RED
state)
```

UC-05-02: TDD GREEN phase — minimal code to pass

`Validates: REQ-EDGE-001` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B in RED state (failing test exists)
When the iterate function writes code to pass the test
Then the code is minimal (just enough to make the test pass)
And the code references "Implements: REQ-F-BETA-001"
And the deterministic evaluator confirms: test passes (GREEN state)
```

UC-05-03: TDD REFACTOR phase — quality improvement

`Validates: REQ-EDGE-001` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B in GREEN state (all tests pass)
When the iterate function refactors for quality
Then the refactored code still passes all tests
And all evaluators re-confirm convergence after refactoring
And the refactoring does not change behaviour
```

UC-05-04: TDD co-evolution oscillates between test and code

`Validates: REQ-EDGE-001` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B at "code↔unit_tests" (bidirectional edge)
When I inspect the iteration history
Then iterations alternate between writing tests and writing code
And this is a single edge with bidirectional construction
And convergence requires both test and code assets to be stable
```

UC-05-05: TDD coverage threshold configurable

Validates: REQ-EDGE-001 | Fixture: IN_PROGRESS | **NEW**

```
Given the project default coverage threshold of 80%
When Feature B's vector overrides the threshold to 95%
Then the TDD edge uses 95% as the convergence criterion
And coverage below 95% reports delta > 0 even if code compiles and
tests pass
And the threshold source (feature override) is recorded in the
checklist
```

UC-05-06: BDD scenarios in Gherkin format

Validates: REQ-EDGE-002 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at the "design→uat_tests" edge
When the iterate function generates UAT test scenarios
Then each scenario is in Given/When/Then format
And each scenario is tagged with "Validates: REQ-F-ALPHA-001"
And scenarios use business language only (no technical jargon)
And scenarios are executable, not just documentation
```

UC-05-07: Every REQ key has at least 1 BDD scenario

Validates: REQ-EDGE-002 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A with 3 REQ keys: REQ-F-ALPHA-001, REQ-F-ALPHA-002,
REQ-F-ALPHA-003
When the BDD edge evaluator checks scenario coverage
Then each REQ key has at least 1 BDD scenario
And missing scenarios are reported as delta > 0
```

UC-05-08: System test BDD vs UAT BDD differentiation

Validates: REQ-EDGE-002 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "design→test_cases" (system test BDD)
And Feature A at "design→uat_tests" (UAT BDD)
When I compare the generated scenarios
Then system test BDD includes technical integration details
And UAT BDD uses pure business language (no technical jargon)
And both formats trace to the same REQ keys
```

UC-05-09: BDD human approval for business scenarios

Validates: REQ-EDGE-002 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "design→uat_tests" with evaluators [agent, human]
When the agent generates BDD scenarios
Then the scenarios are presented to the human (business stakeholder)
And the human validates the business language accuracy
And convergence requires human approval (business language review)
```

UC-05-10: ADR generation at requirements to design edge

Validates: REQ-EDGE-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A at "requirements→design" edge
And mandatory constraint dimensions: ecosystem, deployment, security,
build
When the iterate function produces a design candidate
Then ADRs are generated for each mandatory constraint dimension
And each ADR documents: decision, context, consequences, alternatives
considered
And each ADR references the requirement keys it addresses
And ADRs become part of Context[] for downstream edges
```

UC-05-11: ADRs acknowledge ecosystem constraints

Validates: REQ-EDGE-003 | Fixture: IN_PROGRESS | **NEW**

```
Given the ecosystem constraint: "Python 3.12 + Django 5.0"
When ADRs are generated at the requirements→design edge
Then at least one ADR acknowledges and resolves the ecosystem
constraint
And the ADR documents why Django 5.0 was chosen over alternatives
And the ADR references the requirement keys it satisfies
```

UC-05-12: ADRs are versioned context artifacts

Validates: REQ-EDGE-003 | Fixture: IN_PROGRESS | **NEW**

```
Given ADR-001 generated during Feature A's requirements→design
iteration
When Feature B reaches the same edge
Then Feature B can reference ADR-001 as part of its Context[]
And ADR-001 is version-controlled and immutable once accepted
```

UC-05-13: ADR referencing requirement keys

Validates: REQ-EDGE-003 | Fixture: IN_PROGRESS | **NEW**

```
Given ADR-001 documenting the database choice
When I inspect ADR-001
Then it references the requirement keys that drove the decision
And each referenced REQ key is traceable back to the original intent
```

UC-05-14: Code tagged with Implements

`Validates: REQ-EDGE-004` | Fixture: IN_PROGRESS | **EXISTING** (TestEndToEndTraceability)

```
Given Feature B at "design→code" producing code files
When the iterate function generates code
Then each code file contains "Implements: REQ-F-BETA-001" in language-appropriate comments
And the tag format is the contract (comment syntax varies by language)
```

UC-05-15: Tests tagged with Validates

`Validates: REQ-EDGE-004` | Fixture: IN_PROGRESS | **EXISTING** (TestEndToEndTraceability)

```
Given Feature B at "code↔unit_tests" producing test files
When the iterate function generates tests
Then each test file contains "Validates: REQ-F-BETA-001"
And the tag format matches the code tagging convention
```

UC-05-16: Commit messages include REQ keys

`Validates: REQ-EDGE-004` | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B completing a TDD cycle (RED→GREEN→REFACTOR→COMMIT)
When the iterate function produces a commit
Then the commit message includes the REQ key: "Implements: REQ-F-BETA-001"
And the REQ key in the commit message is validatable by tooling
```

UC-06: Full Lifecycle Closure

Feature Vector: REQ-F-LIFE-001 **Satisfies**: REQ-LIFE-001 through REQ-LIFE-012, REQ-INTENT-003

UC-06-01: CI/CD is a first-class graph edge

`Validates: REQ-LIFE-001` | Fixture: INITIALIZED | **EXISTING** (TestGraphTopology)

```
Given the graph topology
When I inspect transitions involving CI/CD
```

```
Then "Code → CI/CD" exists with evaluators: [deterministic],
constructor: deterministic
And "CI/CD → Running System" exists with evaluators: [deterministic],
constructor: deterministic
And both are iterative edges (failed deployment retries with evaluator
feedback)
```

UC-06-02: CI/CD iteration retries on failure

`Validates: REQ-LIFE-001` | Fixture: IN_PROGRESS | **NEW**

```
Given a feature at the "code→cicd" edge
When the build fails (deterministic evaluator reports FAIL)
Then delta > 0 and the iteration can be retried
And the failure feedback guides the next iteration
And the CI/CD edge behaves like any other iterate() call
```

UC-06-03: Release manifest lists feature vector IDs

`Validates: REQ-LIFE-001` | Fixture: CONVERGED | **NEW**

```
Given all features converged and a release is created
When I inspect the release manifest
Then the manifest lists all feature vector IDs (REQ keys) included in
the release
And a "release_created" event is emitted with features_included count
and coverage_pct
```

UC-06-04: Telemetry tagged with REQ keys

`Validates: REQ-LIFE-002` | Fixture: CONVERGED | **NEW**

```
Given a running system with telemetry configured
When the system emits telemetry (logs, metrics, traces)
Then telemetry records contain req="REQ-F-AUTH-001" as a structured
field
And the REQ keys in telemetry match those in code ("Implements: REQ-
*")
```

UC-06-05: Homeostasis check at runtime

`Validates: REQ-LIFE-002` | Fixture: CONVERGED | **NEW**

```
Given a running system with SLA bounds configured
When the system's latency exceeds the p99 SLA threshold
Then the deterministic evaluator (health probes) reports deviation
And the deviation is tagged with the affected REQ keys
And a drift detection signal is generated
```

UC-06-06: Runtime evaluators span all three types

Validates: REQ-LIFE-002 | Fixture: CONVERGED | **NEW**

```
Given a running system with telemetry monitoring
When evaluators are invoked at the telemetry stage
Then Deterministic Tests check: alerting rules, SLA thresholds, health
probes
And Agent evaluators check: anomaly detection, trend analysis
And Human evaluators respond to: incident triage, capacity planning
```

UC-06-07: Deviation generates new intent

Validates: REQ-LIFE-003 | Fixture: CONVERGED | **NEW**

```
Given a running system that detects a latency deviation on REQ-F-AUTH-
001
When the deviation is confirmed by the ops observer
Then a new intent INT-003 is generated with:
  source: "runtime_feedback"
  deviation details: what metric, expected vs actual
  impacted feature vectors: REQ-F-AUTH-001
And the intent enters the graph as a new feature vector
And the specification becomes a living encoding updated from runtime
observation
```

UC-06-08: New intent enters graph as new feature vector

Validates: REQ-LIFE-003 | Fixture: CONVERGED | **NEW**

```
Given INT-003 generated from runtime deviation
When the human approves the intent for action
Then a new feature vector is spawned from INT-003
And the feature vector enters the graph at the intent node
And the trajectory starts: intent → requirements → design → ...
```

UC-06-09: Feedback loop closure from production to spec

Validates: REQ-LIFE-003 | Fixture: CONVERGED | **NEW**

```
Given a deviation detected at telemetry stage
When the deviation generates an intent that leads to a spec update
Then the loop is closed: telemetry → intent → spec update → new
feature → code → deployment → telemetry
And the full loop is traceable via events.jsonl
And the specification is a living encoding updated from runtime
observation
```

UC-06-10: Feature lineage queryable in production

Validates: REQ-LIFE-004 | Fixture: CONVERGED | **NEW**

```
Given a running system with REQ-tagged telemetry
When I query production metrics filtered by REQ-F-AUTH-001
Then I get: latency per feature, error rate per feature, incident
count per feature
And incidents are traceable to originating requirement via telemetry
REQ key
```

UC-06-11: REQ keys in telemetry match code tags

Validates: REQ-LIFE-004 | Fixture: CONVERGED | **NEW**

```
Given code with "Implements: REQ-F-AUTH-001"
And telemetry with req="REQ-F-AUTH-001"
When I cross-reference code tags with telemetry tags
Then every req tag in telemetry matches an Implements tag in code
And there are no orphaned telemetry REQ keys
```

UC-06-12: Intent raised event has full causal chain

Validates: REQ-LIFE-005 | Fixture: IN_PROGRESS | **EXISTING** (TestConsciousnessLoopIntentRaised)

```
Given an observer that detects a non-trivial delta during iteration
When an "intent_raised" event is emitted
Then the event includes:
  intent_id, trigger, delta, signal_source, vector_type,
  affected_req_keys, prior_intents (causal chain), edge_context,
severity
And the prior_intents field traces: if intent A led to intent B, chain
A→B is visible
```

UC-06-13: Intent raised from development-time signals

Validates: REQ-LIFE-005 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature B at "code↔unit_tests" with the same test failing for 4
iterations
When the iterate agent detects the stuck condition
Then an "intent_raised" event is emitted with signal_source:
"test_failure"
And the delta describes: expected (test passes) vs observed (test
keeps failing)
And the human is presented with the intent for decision (create
vector, acknowledge, dismiss)
```

UC-06-14: Human decision on raised intent

Validates: REQ-LIFE-005 | Fixture: IN_PROGRESS | **NEW**

```
Given an "intent_raised" event presented to the human
When the human decides to "create feature vector"
Then a new feature vector is spawned from the intent
When the human decides to "dismiss"
Then the intent is logged as dismissed with reason
And no new feature vector is created
```

UC-06-15: Seven signal source types recognised

Validates: REQ-LIFE-006 | Fixture: IN_PROGRESS | **EXISTING**
(TestSignalSourceClassification)

```
Given all observer points in the system
When I enumerate the signal source types
Then exactly 7 types exist: gap, test_failure, refactoring,
  source_finding, process_gap, runtime_feedback, ecosystem
And each has an intent template in the feedback loop edge
configuration
And the signal source is recorded in every intent_raised event
```

UC-06-16: Gap analysis emits intent with signal_source: gap

Validates: REQ-LIFE-006 | Fixture: IN_PROGRESS | **NEW**

```
Given a gap analysis run that finds REQ-F-AUTH-001 has no test
coverage
When the gap is significant enough to warrant action
Then an "intent_raised" event is emitted with signal_source: "gap"
And affected_req_keys includes REQ-F-AUTH-001
And the delta describes: expected (test coverage) vs observed (none)
```

UC-06-17: TDD stuck emits test_failure signal

Validates: REQ-LIFE-006 | Fixture: STUCK | **NEW**

```
Given Feature X with the same test failing for 4+ iterations
When the iterate agent detects the stuck condition
Then an "intent_raised" event is emitted with signal_source:
"test_failure"
And the delta includes: which test, how many iterations stuck, what
was tried
```

UC-06-18: Spec modified event emitted on spec change

Validates: REQ-LIFE-007 | Fixture: IN_PROGRESS | **EXISTING**
(TestSpecChangeEvents)

```
Given a signal that leads to a specification update
When a requirement is added, modified, or deprecated
Then a "spec_modified" event is emitted with:
  trigger_intent, signal_source, what_changed (list of REQ key
changes),
  affected_req_keys, spawned_vectors, prior_intents
```

UC-06-19: Spec change history reconstructable from events

Validates: REQ-LIFE-007 | Fixture: IN_PROGRESS | **NEW**

```
Given multiple spec modifications over the project lifecycle
When I replay the "spec_modified" events from events.jsonl
Then the full history of every REQ key modification is reconstructable
And I can determine which spec sections are volatile vs stable
```

UC-06-20: Feedback loop detection in spec changes

Validates: REQ-LIFE-007 | Fixture: IN_PROGRESS | **NEW**

```
Given intent A that caused spec change S1
And spec change S1 that generated intent B
And intent B that caused spec change S2
When I trace prior_intents for intent B
Then the chain A→B is visible
And the feedback loop (A → S1 → B → S2) is detectable from the event
log
```

UC-06-21: Every iteration emits mandatory event

Validates: REQ-LIFE-008 | Fixture: IN_PROGRESS | **EXISTING** (TestProtocolEnforcementHooks)

```
Given any iteration on any edge
When the iteration completes (whether converged or not)
Then an "iteration_completed" event is emitted to events.jsonl
And the event includes: feature, edge, iteration number, delta,
evaluator results
And event emission is a reflex-phase operation (fires unconditionally)
```

UC-06-22: Protocol enforcement detects missing side effects

Validates: REQ-LIFE-008 | Fixture: IN_PROGRESS | **NEW**

```
Given a completed iteration that should have emitted an event
When the protocol enforcement hook verifies side effects
Then it checks: event emitted, feature vector updated, project state
view regenerated,
  source_findings array present, process_gaps array present
```

```
And any missing side effect is flagged as a PROTOCOL_VIOLATION process
gap
```

UC-06-23: Circuit breaker prevents infinite regression

Validates: REQ-LIFE-008 | Fixture: IN_PROGRESS | **NEW**

```
Given an iteration where event emission itself fails (filesystem
error)
When the protocol enforcement hook detects the failure
Then the failure is logged as a TELEM signal in the iteration report
And the iteration does NOT block on the observability failure
And no infinite regression occurs (enforcement failure doesn't trigger
enforcement)
```

UC-06-24: Spec review computes delta against workspace

Validates: REQ-LIFE-009 | Fixture: IN_PROGRESS | **NEW**

```
Given a workspace with 3 features and 15 events
When spec review is invoked
Then it reads: feature vectors, event log, convergence status,
telemetry signals
And computes delta against spec: what spec asserts vs what workspace
contains
And classifies non-zero deltas by signal source
And generates draft intent_raised events with full causal chain
And presents draft intents to human for approval
```

UC-06-25: Spec review is stateless and idempotent

Validates: REQ-LIFE-009 | Fixture: IN_PROGRESS | **NEW**

```
Given the same workspace state and the same spec
When spec review is invoked twice with no intervening changes
Then both invocations produce the same intents
And the review does not modify workspace state
```

UC-06-26: Dev observer watches event stream

Validates: REQ-LIFE-010 | Fixture: IN_PROGRESS | **NEW**

```
Given the dev observer agent specification
When an "iteration_completed" event is emitted
Then the dev observer is triggered (via hooks)
And it reads: events.jsonl, feature vector state, convergence status
And computes delta(workspace_state, spec) → intents
And emits "observer_signal" events with observer_id, signal_source,
severity
```

UC-06-27: Dev observer is a Markov object

Validates: REQ-LIFE-010 | Fixture: IN_PROGRESS | **NEW**

```
Given the dev observer agent
When it executes
Then it reads inputs (event log) and emits events (observer_signal)
And it has no shared mutable state (event log is the mailbox)
And draft intents require human approval (actor model — no autonomous
spec modification)
And it is idempotent: same workspace state + same spec = same
observations
```

UC-06-28: CI/CD observer maps failures to REQ keys

Validates: REQ-LIFE-011 | Fixture: IN_PROGRESS | **NEW**

```
Given the CI/CD observer agent specification
When a CI/CD pipeline fails after a push
Then the observer reads build logs and test results
And maps failures back to REQ keys via Implements/Validates tags in
failing code
And emits "observer_signal" with observer_id: "cicd_observer"
And generates draft intent_raised events for test gaps and build
regressions
```

UC-06-29: CI/CD observer drafts rollback intent

Validates: REQ-LIFE-011 | Fixture: IN_PROGRESS | **NEW**

```
Given a deployment where health checks fail
When the CI/CD observer detects the failure
Then it generates a draft rollback intent (not automatic — human
approves)
And the draft includes: which deployment, what failed, proposed
rollback steps
```

UC-06-30: Ops observer correlates telemetry with REQ keys

Validates: REQ-LIFE-012 | Fixture: CONVERGED | **NEW**

```
Given the ops observer agent specification
And production telemetry with req="REQ-F-AUTH-001" tags
When the ops observer detects a latency anomaly
Then it correlates the anomaly with REQ-F-AUTH-001 via the req= tag
And emits "observer_signal" with observer_id: "ops_observer",
affected_req_keys
And generates draft intent_raised events for SLA breaches
```

UC-06-31: Ops observer runs on schedule

Validates: REQ-LIFE-012 | Fixture: CONVERGED | **NEW**

```
Given the ops observer configured with a daily schedule
When the scheduled interval elapses
Then the observer reads production telemetry
And computes delta against spec constraints (SLA targets, resource
bounds)
And emits observer_signal events
And is stateless: same telemetry + same spec = same observations
```

UC-06-32: Eco-intent generated from ecosystem changes

Validates: REQ-INTENT-003 | Fixture: CONVERGED | **NEW**

```
Given the exteroceptive monitor detects a deprecated API in a
dependency
When the ecosystem signal is classified as requiring action
Then an INT-ECO-* intent is generated with:
  source: "ecosystem"
  ecosystem context (which dependency, what changed)
And the intent feeds into the graph as a new feature vector
```

UC-06-33: Eco-intent covers security, deprecation, API, compliance

Validates: REQ-INTENT-003 | Fixture: CONVERGED | **NEW**

```
Given the monitoring system watches for ecosystem changes
When any of these changes are detected:
  security vulnerability, API deprecation, upstream API change,
compliance update
Then the corresponding INT-ECO-* intent is generated
And each intent type carries the appropriate ecosystem context
```

UC-07: Sensory Systems

Feature Vector: REQ-F-SENSE-001 **Satisfies**: REQ-SENSE-001, REQ-SENSE-002, REQ-SENSE-003, REQ-SENSE-004, REQ-SENSE-005

UC-07-01: Interoceptive monitors run on schedule

Validates: REQ-SENSE-001 | Fixture: IN_PROGRESS | **NEW**

```
Given a sensory service with interoceptive monitors configured
When the configured schedule triggers (default: daily or on workspace
open)
Then all 7 minimum interoceptive monitors execute:
```

```
  INTRO-001 event freshness, INTRO-002 feature vector stall,
  INTRO-003 test health, INTRO-004 state view freshness,
  INTRO-005 build health, INTRO-006 spec/code drift, INTRO-007 event
log integrity
And each monitor produces a typed signal with:
  monitor_id, observation_timestamp, metric_value, threshold, severity
And monitors are observation-only (do not modify workspace)
```

UC-07-02: Interoceptive signal logged to events

Validates: REQ-SENSE-001 | Fixture: IN_PROGRESS | **NEW**

```
Given the INTRO-002 monitor detects a feature stalled for 5 days
When the monitor produces its signal
Then an "interoceptive_signal" event is logged to events.jsonl
And the signal feeds into the affect triage pipeline
And the signal does NOT go directly to conscious review
```

UC-07-03: Monitor thresholds configurable per project and profile

Validates: REQ-SENSE-001 | Fixture: INITIALIZED | **NEW**

```
Given a project using the hotfix profile
And the hotfix profile suppresses non-critical interoceptive signals
When the INTRO-002 (feature stall) monitor fires with severity: info
Then the signal is suppressed (not escalated)
When the INTRO-003 (test health) monitor fires with severity: critical
Then the signal is escalated (critical signals are never suppressed)
```

UC-07-04: Full profile runs all interoceptive monitors

Validates: REQ-SENSE-001 | Fixture: INITIALIZED | **NEW**

```
Given a project using the full profile
When the sensory service runs its schedule
Then all 7 interoceptive monitors execute without suppression
And all signals (info, warning, critical) are processed by the triage
pipeline
```

UC-07-05: Exteroceptive monitors run on schedule

Validates: REQ-SENSE-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a sensory service with exteroceptive monitors configured
When the configured schedule triggers
Then all 4 minimum exteroceptive monitors execute:
  EXTRO-001 dependency freshness, EXTRO-002 CVE scanning,
  EXTRO-003 runtime telemetry deviation, EXTRO-004 API contract
changes
And each monitor produces a typed signal with:
```

```
  monitor_id, observation_timestamp, external_source, finding,
severity
And monitors are observation-only (do not modify workspace)
```

UC-07-06: Exteroceptive signal logged to events

Validates: REQ-SENSE-002 | Fixture: IN_PROGRESS | **NEW**

```
Given the EXTRO-002 CVE scanner detects a critical vulnerability
When the monitor produces its signal
Then an "exteroceptive_signal" event is logged to events.jsonl
And the signal feeds into the affect triage pipeline
```

UC-07-07: Spike profile disables exteroception

Validates: REQ-SENSE-002 | Fixture: INITIALIZED | **NEW**

```
Given a project using the spike profile
When the sensory service runs its schedule
Then exteroceptive monitors are disabled entirely
And no exteroceptive_signal events are emitted
And interoceptive monitors may still run (if configured for spike)
```

UC-07-08: Affect triage — rule-based classification first

Validates: REQ-SENSE-003 | Fixture: IN_PROGRESS | **NEW**

```
Given a sensory signal from INTRO-003 (test health): "3 tests failing"
When the affect triage pipeline processes the signal
Then rule-based classification fires first (fast, pattern-matching)
And the signal is classified: signal_source=test_failure,
severity=warning
And if the rule matches, no agent classification is invoked
And an "affect_triage" event is logged with the classification
decision
```

UC-07-09: Affect triage — agent classification for ambiguous signals

Validates: REQ-SENSE-003 | Fixture: IN_PROGRESS | **NEW**

```
Given a sensory signal that doesn't match any classification rule
When the affect triage pipeline processes the signal
Then the probabilistic agent classifier is invoked
And the agent assigns: signal_source, severity, recommended_action
And an "affect_triage" event is logged with classification method:
"agent"
```

UC-07-10: Escalation thresholds vary by profile

Validates: REQ-SENSE-003 | Fixture: INITIALIZED | **NEW**

```
Given the hotfix profile with low escalation threshold (escalate
aggressively)
And the spike profile with high escalation threshold (suppress most)
When the same warning-severity signal is processed
Then the hotfix profile escalates the signal (below threshold =
escalate)
And the spike profile suppresses the signal (above threshold = defer)
And below-threshold signals are logged but not escalated
```

UC-07-11: Above-threshold signals generate draft proposals

Validates: REQ-SENSE-003 | Fixture: IN_PROGRESS | **NEW**

```
Given a critical-severity signal above the escalation threshold
When the affect triage pipeline processes the signal
Then a draft proposal is generated via probabilistic agent evaluation
And the proposal is surfaced to human via the review boundary
And the proposal does NOT modify workspace files (draft only)
```

UC-07-12: Sensory monitor registry in configuration

Validates: REQ-SENSE-004 | Fixture: INITIALIZED | **EXISTING**
(TestSensorySystemsAcceptanceCriteria)

```
Given an initialized workspace
When I inspect the sensory configuration
Then sensory_monitors.yml defines which monitors are active, their
schedules, and thresholds
And affect_triage.yml defines classification patterns, severity
mapping, and escalation thresholds
And profile-level overrides can enable/disable specific monitors
```

UC-07-13: Monitor health — meta-monitoring

Validates: REQ-SENSE-004 | Fixture: IN_PROGRESS | **NEW**

```
Given the INTRO-003 test health monitor
When the monitor itself fails to run (e.g., test runner not installed)
Then an interoceptive meta-signal is generated:
  "The system senses that its own sensing has failed"
And the meta-signal has severity: critical (sensing failure is always
critical)
And the meta-signal feeds into the triage pipeline
```

UC-07-14: Review boundary separates sensing from changes

`Validates: REQ-SENSE-005` | Fixture: IN_PROGRESS | **NEW**

```
Given the sensory service has generated a draft proposal
When the proposal reaches the review boundary
Then the human can: view the proposal, approve it, or dismiss it
And approved proposals are applied by the interactive session (NOT the
sensory service)
And the sensory service cannot modify workspace files
And dismissed proposals are logged with reason
```

UC-07-15: Two event categories enforced at review boundary

`Validates: REQ-SENSE-005` | Fixture: IN_PROGRESS | **NEW**

```
Given the review boundary between sensory service and interactive
session
When events are classified
Then sensor/evaluate events are autonomous and observation-only
And change-approval events are conscious and human-approved
And no sensor event can trigger a file modification
And human approval is required for all workspace changes
```

UC-07-16: Review boundary preserves human accountability

`Validates: REQ-SENSE-005` | Fixture: IN_PROGRESS | **NEW**

```
Given a draft proposal from the sensory service
When the human reviews and approves the proposal
Then the file modification is attributed to the human decision
And REQ-EVAL-003 (Human Accountability) is preserved
And the approval event records: who approved, what was proposed, what
was applied
```

UC-08: Developer Tooling

Feature Vector: REQ-F-TOOL-001 **Satisfies**: REQ-TOOL-001 through REQ-TOOL-010

UC-08-01: Plugin is installable and discoverable

`Validates: REQ-TOOL-001` | Fixture: CLEAN | **EXISTING** (TestPluginJson)

```
Given a clean project directory
When I install the AI SDLC methodology plugin
Then the plugin is discovered by the platform
And the plugin includes: agent configurations, commands, templates,
```

```
edge configs
And the plugin is versioned (semver)
```

UC-08-02: Plugin is versioned with semver

Validates: REQ-TOOL-001 | Fixture: INITIALIZED | **EXISTING**
(TestVersionConsistency)

```
Given an installed plugin
When I inspect the plugin version
Then the version follows semantic versioning (MAJOR.MINOR.PATCH)
And the plugin version is consistent across all plugin artifacts
```

UC-08-03: Workspace supports task tracking

Validates: REQ-TOOL-002 | Fixture: INITIALIZED | **EXISTING**
(TestTaskUpdateOnConvergence)

```
Given an initialized workspace
When I inspect the workspace structure
Then I find task tracking directories: active, completed, archived
And context preservation across sessions via the checkpoint mechanism
And the workspace is version-controlled (git)
```

UC-08-04: Context preserved across sessions

Validates: REQ-TOOL-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a feature in progress with 5 iterations completed
When I close the session and start a new one
Then the feature's iteration history is preserved
And the event log contains all previous iterations
And the workspace state is reconstructable from the event log
```

UC-08-05: Workspace is git-integrated

Validates: REQ-TOOL-002 | Fixture: INITIALIZED | **NEW**

```
Given an initialized workspace
When I check the version control status
Then the .ai-workspace/ directory is within the git repository
And workspace artifacts are tracked by git
And context elements are version-controlled
```

UC-08-06: Workflow commands for task management

Validates: REQ-TOOL-003 | Fixture: IN_PROGRESS | **NEW**

```
Given an initialized workspace with active features
When I invoke task management commands
```

```
Then I can: create tasks, update task status, complete tasks
And task status changes are reflected in ACTIVE_TASKS.md
```

UC-08-07: Workflow commands for context operations

Validates: REQ-TOOL-003 | Fixture: IN_PROGRESS | **NEW**

```
Given an in-progress workspace
When I invoke checkpoint
Then a context snapshot is saved with current timestamp and git ref
When I invoke restore from a previous checkpoint
Then the workspace context is restored to that point
And the restore is non-destructive (no data loss)
```

UC-08-08: Status command shows progress and coverage

Validates: REQ-TOOL-003 | Fixture: IN_PROGRESS | **NEW**

```
Given a workspace with 3 features at various stages
When I invoke the status command
Then I see: progress per feature, coverage gaps, pending actions
And the status is derived from the event log (not stored state)
```

UC-08-09: Release management with semver

Validates: REQ-TOOL-004 | Fixture: CONVERGED | **NEW**

```
Given all features converged
When I create a release with version "1.0.0"
Then the release is tagged with semantic version
And a changelog is generated from the feature vectors and events
And the release manifest includes feature vector coverage summary
And a "release_created" event is emitted
```

UC-08-10: Release includes feature coverage summary

Validates: REQ-TOOL-004 | Fixture: CONVERGED | **NEW**

```
Given a release manifest for version 1.0.0
When I inspect the manifest
Then it lists all included feature vectors with their REQ keys
And shows coverage percentage (how many REQ keys have full trajectory)
And flags known gaps (REQ keys without test or telemetry coverage)
```

UC-08-11: Release tagging creates git tag

Validates: REQ-TOOL-004 | Fixture: CONVERGED | **NEW**

```
Given a release created for version 1.0.0
When I inspect the git repository
```

```
Then a tag "v1.0.0" exists
And the tag points to the commit with the release manifest
```

UC-08-12: Test gap analysis identifies uncovered REQ keys

Validates: REQ-TOOL-005 | Fixture: IN_PROGRESS | **EXISTING** (TestTraceabilityPipeline)

```
Given 5 features with 15 REQ keys total
When I run the gap analysis command
Then the analysis identifies which REQ keys have "Validates:" tags in
tests
And which REQ keys are missing test coverage
And uncovered trajectories are reported with suggested test cases
```

UC-08-13: Gap analysis suggests test cases for gaps

Validates: REQ-TOOL-005 | Fixture: IN_PROGRESS | **NEW**

```
Given a gap analysis that finds REQ-F-AUTH-002 has no test coverage
When the analysis report is generated
Then it suggests test cases that would cover REQ-F-AUTH-002
And the suggestions reference the acceptance criteria from the
requirement
```

UC-08-14: Gap analysis reports as events

Validates: REQ-TOOL-005 | Fixture: IN_PROGRESS | **NEW**

```
Given a gap analysis run
When the analysis completes
Then a "gaps_validated" event is emitted with:
  layers_run, total_req_keys, full_coverage count, test_gaps count,
telemetry_gaps count
And the event is queryable from the event log
```

UC-08-15: Methodology hooks trigger on commit

Validates: REQ-TOOL-006 | Fixture: IN_PROGRESS | **NEW**

```
Given methodology hooks configured for commit events
When a commit is made
Then the hook validates: REQ-* tags present in changed files
And the hook validates: evaluator convergence recorded for changed
features
And the hook is configurable per project
```

UC-08-16: Methodology hooks trigger on edge transition

Validates: REQ-TOOL-006 | Fixture: IN_PROGRESS | **NEW**

```
Given methodology hooks configured for edge transitions
When an edge converges (edge_converged event)
Then the hook validates: all required side effects completed
And the hook fires unconditionally (reflex phase)
```

UC-08-17: Project scaffolding creates structure

`Validates: REQ-TOOL-007` | Fixture: CLEAN | **EXISTING** (TestInitWorkflow)

```
Given a clean project directory
When I initialize the AI SDLC workspace
Then the scaffolding creates:
  asset graph configuration, context directories, workspace structure
And templates with placeholder guidance are populated
And a design-implementation binding manifest is created
```

UC-08-18: Scaffolding generates templates

`Validates: REQ-TOOL-007` | Fixture: CLEAN | **NEW**

```
Given a clean project directory
When I run the project scaffolding
Then template files are generated:
  feature_vector_template.yml, project_constraints.yml,
graph_topology.yml
And each template has placeholder guidance explaining what to fill in
And the templates are ready for customisation
```

UC-08-19: Context snapshot is immutable

`Validates: REQ-TOOL-008` | Fixture: IN_PROGRESS | **NEW**

```
Given a workspace checkpoint created at timestamp T1
When I continue working (creating new events, modifying features)
Then the checkpoint at T1 remains unchanged
And the checkpoint captures: active tasks, current work context,
timestamp, git ref
And a "checkpoint_created" event is emitted
```

UC-08-20: Context snapshot integrates with task checkpoint

`Validates: REQ-TOOL-008` | Fixture: IN_PROGRESS | **NEW**

```
Given a workspace checkpoint
When I inspect the checkpoint contents
Then it includes: active tasks from ACTIVE_TASKS.md
And current feature vector states
And the context hash at checkpoint time
And the git reference for the commit state
```

UC-08-21: Feature view for a specific REQ key

Validates: REQ-TOOL-009 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A with REQ-F-ALPHA-001 referenced in spec, design, and
code
When I invoke the feature view for REQ-F-ALPHA-001
Then I see a cross-artifact report:
  Spec: tagged (line N), Design: tagged (component X), Code: tagged
(file.py:42)
  Tests: NOT tagged, Telemetry: NOT tagged
And missing stages are flagged: "No telemetry tagging"
And coverage summary: "3 of 5 stages tagged"
```

UC-08-22: Feature view aggregated across all features

Validates: REQ-TOOL-009 | Fixture: IN_PROGRESS | **NEW**

```
Given 3 features with a total of 10 REQ keys
When I invoke the feature view for all REQ keys
Then I see a summary table: REQ key × stage (spec, design, code,
tests, telemetry)
And each cell shows: tagged / not yet / N/A
And the total coverage percentage is computed
```

UC-08-23: Spec/Design boundary — technology leakage detected

Validates: REQ-TOOL-010 | Fixture: IN_PROGRESS | **NEW**

```
Given a requirements document that mentions "Django REST Framework"
When the spec/design boundary enforcement evaluator runs
Then it flags "Django REST Framework" as technology leakage in
requirements
And the requirement is rejected until the technology reference is
removed
And the technology choice belongs in a design ADR, not in the spec
```

UC-08-24: Multiple design variants supported

Validates: REQ-TOOL-010 | Fixture: INITIALIZED | **NEW**

```
Given REQ-F-AUTH-001 in the shared specification
When two implementations (Claude and Gemini) create designs for the
same REQ key
Then each implementation has its own design trajectory
And the same REQ key appears in both design documents
And the spec remains technology-agnostic (shared across
implementations)
```

UC-09: User Experience

Feature Vector: REQ-F-UX-001 **Satisfies**: REQ-UX-001 through REQ-UX-007

UC-09-01: State detection — UNINITIALISED

Validates: REQ-UX-001 | Fixture: CLEAN | **EXISTING** (TestTwoCommandUX)

```
Given a project directory with no .ai-workspace/
When I invoke the start command
Then the system detects state: UNINITIALISED
And routes to progressive initialization (5 questions)
And does NOT require the user to know command names
```

UC-09-02: State detection — NEEDS_CONSTRAINTS

Validates: REQ-UX-001 | Fixture: INITIALIZED | **NEW**

```
Given an initialized workspace where Feature A has reached
"requirements→design"
And mandatory constraint dimensions (ecosystem, deployment, security,
build) are empty
When I invoke the start command
Then the system detects state: NEEDS_CONSTRAINTS
And prompts for each unresolved mandatory dimension
And mentions advisory dimensions as optional
```

UC-09-03: State detection — NEEDS_INTENT

Validates: REQ-UX-001 | Fixture: INITIALIZED | **NEW**

```
Given an initialized workspace with no specification/INTENT.md (or
empty placeholder)
When I invoke the start command
Then the system detects state: NEEDS_INTENT
And prompts: "Describe what you want to build"
And writes the response to specification/INTENT.md
```

UC-09-04: State detection — NO_FEATURES

Validates: REQ-UX-001 | Fixture: INITIALIZED | **NEW**

```
Given an initialized workspace with intent but no features in
features/active/
When I invoke the start command
Then the system detects state: NO_FEATURES
And routes to feature creation (spawn command)
And informs: "Intent captured. Let's create your first feature
vector."
```

UC-09-05: State detection — IN_PROGRESS

Validates: REQ-UX-001 | Fixture: IN_PROGRESS | **EXISTING** (TestTwoCommandUX)

```
Given a workspace with 3 active features, 1 converged
When I invoke the start command
Then the system detects state: IN_PROGRESS
And selects the highest-priority actionable feature
And determines the next unconverged edge
And delegates to the iterate function
```

UC-09-06: State detection — ALL_CONVERGED

Validates: REQ-UX-001 | Fixture: CONVERGED | **EXISTING** (TestTwoCommandUX)

```
Given a workspace where all active features are fully converged
When I invoke the start command
Then the system detects state: ALL_CONVERGED
And suggests: gap analysis, release, or new feature creation
```

UC-09-07: State detection — STUCK

Validates: REQ-UX-001 | Fixture: STUCK | **NEW**

```
Given Feature X with delta=3 unchanged for 4 consecutive iterations
When I invoke the start command
Then the system detects state: STUCK (before IN_PROGRESS)
And surfaces the stuck feature with: which checks are failing, how
long stuck
And recommends: spawn discovery, human review, or force-iterate
```

UC-09-08: State detection — ALL_BLOCKED

Validates: REQ-UX-001 | Fixture: STUCK | **NEW**

```
Given all active features are blocked:
  Feature Y: blocked by spawn dependency
  Feature Z: blocked by pending human review
When I invoke the start command
Then the system detects state: ALL_BLOCKED
And surfaces each blocked feature with its reason
And recommends: work on blocking dependency, or complete pending
review
```

UC-09-09: State derived, never stored

Validates: REQ-UX-001 | Fixture: IN_PROGRESS | **NEW**

```
Given a workspace with events.jsonl and feature vector files
When the start command detects state
Then the state is computed from workspace filesystem + event log
And there is no "current_state" variable stored anywhere
And the same workspace state always produces the same detected state
```

UC-09-10: Progressive init requires 5 or fewer inputs

Validates: REQ-UX-002 | Fixture: CLEAN | **EXISTING**
(TestTwoCommandUXAcceptanceCriteria)

```
Given a clean project directory
When I run progressive initialization
Then the system asks at most 5 questions:
  project name (auto-detected, confirm), project kind, language (auto-
detected, confirm),
  test runner (auto-detected, confirm), intent description
And constraint dimensions are NOT prompted (deferred to design edge)
```

UC-09-11: Constraints deferred until design edge

Validates: REQ-UX-002 | Fixture: INITIALIZED | **EXISTING**
(TestTwoCommandUXAcceptanceCriteria)

```
Given a freshly initialized workspace
When I start working on "intent→requirements" edge
Then the system does NOT ask for ecosystem, deployment, security, or
build constraints
And constraints are only prompted when a feature reaches
"requirements→design"
```

UC-09-12: Sensible defaults inferred from project detection

Validates: REQ-UX-002 | Fixture: CLEAN | **NEW**

```
Given a project directory containing pyproject.toml with pytest
configuration
When progressive init runs
Then language is auto-detected as Python
And test runner is auto-detected as pytest
And the user confirms (or overrides) the detected values
And the system does not ask for information it can infer
```

UC-09-13: Advisory dimensions optional

Validates: REQ-UX-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a feature reaching the "requirements→design" edge
When mandatory constraints are prompted
Then advisory dimensions (data_governance, performance_envelope,
```

```
observability, error_handling)
  are mentioned but skippable ("Press Enter to skip")
And skipping advisory dimensions does not block convergence
```

UC-09-14: Status shows "you are here" per feature

Validates: REQ-UX-003 | Fixture: IN_PROGRESS | **EXISTING** (TestTwoCommandUXAcceptanceCriteria)

```
Given a workspace with 3 features at different stages
When I invoke the status command
Then each feature shows its current position in the graph:
  Feature A: intent→req ✓ req→design [iterating] design→code ○
code↔tests ○
  Feature B: intent→req ✓ req→design ✓ design→code ✓ code↔tests
[iterating δ=2]
  Feature C: intent→req [iterating] ...
```

UC-09-15: Status shows cross-feature rollup

Validates: REQ-UX-003 | Fixture: IN_PROGRESS | **NEW**

```
Given 3 features with various convergence states
When I invoke the status command
Then I see aggregate edge convergence counts:
  "12 of 20 edges converged across 3 features"
And blocked features shown with reason
And unactioned intent_raised events surfaced as signals
```

UC-09-16: Status previews next action

Validates: REQ-UX-003 | Fixture: IN_PROGRESS | **NEW**

```
Given an IN_PROGRESS workspace
When I invoke the status command
Then I see a preview of what the start command would do next:
  "Next: REQ-F-ALPHA-001 on requirements→design (iteration 2)"
And the reasoning for the selection is shown
```

UC-09-17: Status shows workspace health

Validates: REQ-UX-003 | Fixture: IN_PROGRESS | **NEW**

```
Given a workspace with some health issues
When I invoke the status command
Then the health check reports:
  event log integrity: OK
  feature vector consistency: OK
  orphaned spawns: 1 found
```

```
  stuck features: Feature X (4 iterations, δ unchanged)
And remediation suggestions are provided for each issue
```

UC-09-18: Feature selection — time-boxed spawns first

Validates: REQ-UX-004 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A (standard, 2 edges remaining)
And Feature B (time-boxed spike, 25% time remaining, 1 edge remaining)
When the start command selects a feature
Then Feature B is selected (time-boxed spawn approaching expiry takes
priority)
And the selection reasoning is displayed
```

UC-09-19: Feature selection — closest-to-complete reduces WIP

Validates: REQ-UX-004 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A (3 edges remaining)
And Feature C (1 edge remaining, same priority)
And no time-boxed spawns
When the start command selects a feature
Then Feature C is selected (closest-to-complete to reduce WIP)
```

UC-09-20: Edge determination — topological walk

Validates: REQ-UX-004 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A with converged edges: intent→requirements,
requirements→design
And the next edge in topological order is "design→code"
When the start command determines the next edge
Then "design→code" is selected
And converged edges are skipped
And edges not in the active profile are skipped
```

UC-09-21: User can override feature and edge selection

Validates: REQ-UX-004 | Fixture: IN_PROGRESS | **NEW**

```
Given the automatic selection would choose Feature A on edge X
When the user provides --feature "REQ-F-BETA-001" --edge
"code↔unit_tests"
Then the user's override is used instead of automatic selection
And the system proceeds with the specified feature and edge
```

UC-09-22: Recovery detects corrupted event log

Validates: REQ-UX-005 | Fixture: IN_PROGRESS | **NEW**

```
Given an events.jsonl file with 3 corrupted lines (malformed JSON)
When the start command runs its health check
Then the corruption is detected
And the system offers: "Event log has 3 corrupted lines. Truncate at
last valid line?"
And the recovery is non-destructive (asks before modifying)
```

UC-09-23: Recovery detects orphaned spawns

Validates: REQ-UX-005 | Fixture: IN_PROGRESS | **NEW**

```
Given a spawn REQ-F-SPIKE-001 whose parent feature no longer exists
When the start command runs its health check
Then the orphaned spawn is detected
And the system offers: "Spawn REQ-F-SPIKE-001 has no parent. Link to
feature or archive?"
And the user decides the disposition
```

UC-09-24: Workspace rebuildable from event log

Validates: REQ-UX-005 | Fixture: IN_PROGRESS | **NEW**

```
Given a workspace with corrupted feature vector files
When the user chooses to rebuild from the event log
Then feature vector states are reconstructed from events.jsonl
And task tracking is regenerated
And the rebuilt workspace matches the event log history
```

UC-09-25: Recovery detects missing feature vectors

Validates: REQ-UX-005 | Fixture: IN_PROGRESS | **NEW**

```
Given the event log references feature REQ-F-ALPHA-001
But the feature vector file features/active/REQ-F-ALPHA-001.yml is
missing
When the start command runs its health check
Then the missing vector is detected
And the system offers to regenerate the vector from events
```

UC-09-26: Escalation displayed inline in interactive mode

Validates: REQ-UX-006 | Fixture: IN_PROGRESS | **NEW**

```
Given an IntentEngine escalation during an iteration
When the escalation occurs in interactive CLI mode
Then the escalation is displayed inline with clear visual distinction
And the escalation is NOT buried in normal output
And the user can act on it immediately
```

UC-09-27: Pending escalations shown in status

Validates: REQ-UX-006 | Fixture: IN_PROGRESS | **NEW**

```
Given 2 pending escalations in the review queue
When I invoke the status command
Then I see: "2 pending escalations" with their summaries
And the start command checks escalations before feature selection
And escalations take priority over normal iteration
```

UC-09-28: Unactioned escalations re-surface with elevated urgency

Validates: REQ-UX-006 | Fixture: IN_PROGRESS | **NEW**

```
Given an escalation that has been pending for 3 iterations of the
affected feature
When the next iteration begins
Then the escalation is re-surfaced with elevated urgency
And the urgency level is visually distinct from the original
And no escalation is silently dropped — the review queue guarantees
delivery
```

UC-09-29: Escalation queue in async mode

Validates: REQ-UX-006 | Fixture: IN_PROGRESS | **NEW**

```
Given an IntentEngine escalation during sensory service processing
(async mode)
When the escalation occurs outside an interactive session
Then the escalation is written to .ai-workspace/reviews/pending/ as
structured YAML
And the YAML includes: source level, triggering signal, affected
features, timestamp
And the next /gen-status displays the pending count
```

UC-09-30: Zoom in expands edge to sub-graph

Validates: REQ-UX-007 | Fixture: IN_PROGRESS | **NEW**

```
Given the edge "design→code" for Feature A
When the user requests zoom: --edge "design→code" --level in
Then the system proposes intermediate node types:
  design → module_decomp → basis_projections → code_per_module → code
And each intermediate node gets its own asset type in the workspace
And sub-edges inherit parent evaluators (overridable)
And a "graph_zoom_in" event is emitted with full before/after topology
```

UC-09-31: Zoom out collapses sub-graph

Validates: REQ-UX-007 | Fixture: IN_PROGRESS | **NEW**

```
Given a previously zoomed "design→code" edge with 3 intermediate nodes
And all intermediate nodes are converged
When the user requests zoom: --edge "design→code" --level out
Then the sub-graph collapses back to a single "design→code" edge
And intermediate assets are archived (not deleted)
And convergence state reflects sub-graph completion
And a "graph_zoom_out" event is emitted
```

UC-09-32: Selective zoom retains waypoints

`Validates: REQ-UX-007` | Fixture: IN_PROGRESS | **NEW**

```
Given a zoomed edge with 3 intermediate nodes
When the user selects node "module_decomp" as a mandatory waypoint
And collapses the other intermediates
Then "module_decomp" becomes an explicit convergence checkpoint
And the other intermediates are treated as internal to encapsulating
edges
And a "graph_zoom_selective" event is emitted
```

UC-09-33: Zoom preserves graph invariants

`Validates: REQ-UX-007` | Fixture: IN_PROGRESS | **NEW**

```
Given a zoom operation on any edge
When the zoom completes
Then the graph remains directed, typed, and all edges have evaluators
And feature vectors traversing the zoomed edge see the sub-graph
topology
And the zoom is reversible and auditable via the event log
```

UC-10: Multi-Agent Coordination

Feature Vector: REQ-F-COORD-001 **Satisfies**: REQ-COORD-001 through REQ-COORD-005

UC-10-01: Agent identity on all events

`Validates: REQ-COORD-001` | Fixture: IN_PROGRESS | **EXISTING** (TestMultiAgentCoordinationAcceptanceCriteria)

```
Given multi-agent mode is active
When any agent emits an event to events.jsonl
Then the event includes agent_id (unique per instance) and agent_role
And agent_role is drawn from the project-defined role registry
And agent identity is self-declared (not centrally assigned)
```

UC-10-02: Single-agent backward compatibility

Validates: REQ-COORD-001 | Fixture: IN_PROGRESS | **NEW**

```
Given single-agent mode (default)
When events are emitted
Then agent_id and agent_role fields are optional
And if present, default to agent_id: "primary"
And all existing single-agent workflows continue to function
```

UC-10-03: Agent role registry is simple YAML

Validates: REQ-COORD-001 | Fixture: INITIALIZED | **NEW**

```
Given an initialized workspace in multi-agent mode
When I inspect the agent role registry
Then it is a simple YAML list: [architect, tdd_engineer,
documentation, ...]
And there is no role hierarchy or inheritance
And the event log is the source of truth for which agents have
participated
```

UC-10-04: Feature assignment via edge_claim event

Validates: REQ-COORD-002 | Fixture: IN_PROGRESS | **NEW**

```
Given Agent A wants to work on Feature X's "code↔unit_tests" edge
When Agent A emits an "edge_claim" event with agent_id, feature, and
edge
Then the serialiser resolves the claim:
  If no other agent holds this edge: emit "edge_started" (claim
granted)
  If another agent holds this edge: emit "claim_rejected" with reason
and holding_agent
And the claim resolution is atomic (first claim wins)
```

UC-10-05: No lock files for assignment

Validates: REQ-COORD-002 | Fixture: IN_PROGRESS | **NEW**

```
Given multi-agent feature assignment
When I inspect the workspace for assignment state
Then there are no lock files, lease files, or mutex mechanisms
And assignment state is entirely derivable from the event log
And current assignments are a derived projection (replay edge_claim,
edge_started, edge_converged)
```

UC-10-06: Stale claims detectable

Validates: REQ-COORD-002 | Fixture: IN_PROGRESS | **NEW**

```
Given Agent A claimed Feature X's edge 2 hours ago
And no follow-up events from Agent A since then
And the configured timeout is 1 hour
When the health check runs
Then a "claim_expired" telemetry signal is generated
And the edge becomes available for other agents to claim
```

UC-10-07: Single-agent mode skips claim step

Validates: REQ-COORD-002 | Fixture: IN_PROGRESS | **NEW**

```
Given single-agent mode (one agent writing events)
When the agent starts work on an edge
Then it emits "edge_started" directly (no claim step needed)
And the single-agent workflow is simpler than multi-agent
```

UC-10-08: Agent working state isolated from shared state

Validates: REQ-COORD-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Agent A working on Feature X's "design→code" edge
When Agent A generates code drafts
Then drafts are stored in Agent A's private directory
(agents/<id>/drafts/)
And the shared project state (features/, events/, spec/) is not
written directly
And only converged assets enter the shared workspace (via promotion)
```

UC-10-09: Promotion requires evaluator gate

Validates: REQ-COORD-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Agent A has produced a code candidate in its private workspace
When Agent A requests promotion to shared state
Then the configured evaluators for this edge must pass
And if human review is configured, the human must approve
And only after all evaluators pass does the asset enter shared paths
```

UC-10-10: Agent crash only loses private state

Validates: REQ-COORD-003 | Fixture: IN_PROGRESS | **NEW**

```
Given Agent A has emitted 5 iteration events and has private drafts
When Agent A crashes mid-iteration
Then the 5 emitted events persist in events.jsonl
And Agent A's private working state may be discarded
And shared project state is unaffected
And another agent (or a restarted Agent A) can continue from the event
log
```

UC-10-11: Orthogonal features freely parallelisable

Validates: REQ-COORD-004 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A (authentication) and Feature B (reporting)
And the inner product (shared module count) is zero
When two agents are assigned to Feature A and Feature B respectively
Then they can work in parallel with no coordination
And their iterations do not conflict (orthogonal trajectories)
```

UC-10-12: Non-orthogonal features trigger warning

Validates: REQ-COORD-004 | Fixture: IN_PROGRESS | **NEW**

```
Given Feature A and Feature C that share a "user_service" module
And the inner product (shared module count) is 1
When both features are assigned to different agents concurrently
Then the system triggers a warning: "Non-orthogonal features assigned
concurrently"
And suggests sequencing: build shared module first, then diverge
```

UC-10-13: Inner product computed from dependency DAG

Validates: REQ-COORD-004 | Fixture: IN_PROGRESS | **NEW**

```
Given the module dependency DAG for the project
When the inner product is computed for two features
Then it uses the actual module dependency graph (not a heuristic)
And the result is the count of shared modules
And zero shared modules = safe parallelism
```

UC-10-14: Role-based authority scopes convergence

Validates: REQ-COORD-005 | Fixture: IN_PROGRESS | **NEW**

```
Given the role registry:
  tdd_engineer: [code_unit_tests]
  architect: [requirements_design, design_code]
When Agent A (role: tdd_engineer) attempts to converge
"requirements→design"
Then the convergence is rejected (outside role authority)
And a "convergence_escalated" event is emitted
And the edge is escalated to an agent with architect role or a human
```

UC-10-15: Human authority is universal

Validates: REQ-COORD-005 | Fixture: IN_PROGRESS | **NEW**

```
Given any edge type in the graph
When a human evaluator reviews and approves
```

```
Then the convergence is accepted regardless of the edge type
And human authority is universal (can converge any edge)
```

UC-10-16: Role definitions are project-configurable

`Validates: REQ-COORD-005` | Fixture: INITIALIZED | **NEW**

```
Given a project that defines custom roles:
  security_reviewer: [design_code, code_unit_tests]
  data_engineer: [design_code]
When the role registry is loaded
Then the custom roles are used for authority scoping
And the roles are not hard-coded in the engine
```

UC-11: IntentEngine / Supervision

Feature Vector: REQ-F-SUPV-001 **Satisfies**: REQ-SUPV-001, REQ-SUPV-002

UC-11-01: Every edge traversal produces IntentEngine output

`Validates: REQ-SUPV-001` | Fixture: IN_PROGRESS | **NEW**

```
Given any edge traversal (e.g., Feature A on "requirements→design")
When the iterate function completes
Then a classified observation is produced (observer output)
And a typed routing decision is produced (evaluator output)
And the output is exactly one of: reflex.log, specEventLog, or
escalate
```

UC-11-02: Ambiguity classified into three regimes

`Validates: REQ-SUPV-001` | Fixture: IN_PROGRESS | **NEW**

```
Given an observation from an edge evaluator
When the evaluator classifies the ambiguity
Then the classification is exactly one of:
  zero ambiguity → reflex (fire-and-forget event)
  bounded nonzero → probabilistic disambiguation (deferred intent)
  persistent/unbounded → escalate to higher consciousness level
And the classification maps to the three evaluator types
```

UC-11-03: Three output types are exhaustive

`Validates: REQ-SUPV-001` | Fixture: IN_PROGRESS | **NEW**

```
Given any observation at any scale (iteration, edge, feature,
production)
When the IntentEngine processes the observation
```

```
Then the output is exactly one of:
  reflex.log (fire-and-forget event → events.jsonl)
  specEventLog (deferred intent for further processing)
  escalate (push to higher consciousness level)
And these three types map to existing event types (no new types
required)
And every possible observation routes to exactly one output type
```

UC-11-04: Affect propagates through chained invocations

`Validates: REQ-SUPV-001` | Fixture: IN_PROGRESS | **NEW**

```
Given an IntentEngine invocation at Level 1 (single iteration)
And the invocation carries affect: urgency=high, valence=negative
When the output is "escalate" (pushing to Level 2: edge convergence)
Then Level 2 receives the intent + affect from Level 1
And the urgency/valence propagates and transforms at each level
And the affect influences the evaluation at Level 2
```

UC-11-05: Level N escalate becomes Level N+1 reflex input

`Validates: REQ-SUPV-001` | Fixture: IN_PROGRESS | **NEW**

```
Given Level 1 (iteration) produces output: escalate
When Level 2 (edge convergence) receives the escalation
Then Level 2 treats it as reflex input (automatic processing)
And Level 1's reflex.log events are invisible to Level 2 (handled at
origin)
And this implements consciousness-as-relative (each level observes a
different scale)
```

UC-11-06: IntentEngine applies at every scale

`Validates: REQ-SUPV-001` | Fixture: IN_PROGRESS | **NEW**

```
Given the fractal application table:
  Level 1: single iteration
  Level 2: edge convergence
  Level 3: feature traversal
  Level 4: sensory monitoring
  Level 5: production homeostasis
  Level 6: spec review
When each level processes an observation
Then each uses the same IntentEngine pattern: observer → evaluator →
typed_output
And the pattern composes the four primitives (Graph, Iterate,
Evaluators, Spec+Context)
```

UC-11-07: IntentEngine interface on all actors

Validates: REQ-SUPV-001 | Fixture: IN_PROGRESS | **NEW**

```
Given actors: iterate agent, sensory monitors, protocol hooks,
observers
When each actor processes an observation
Then each exposes the IntentEngine interface:
  observer → evaluator → typed_output(reflex.log | specEventLog |
escalate)
And the interface is parameterised by intent + affect
And this is not a fifth primitive — it is the composition law
```

UC-11-08: Every constraint has a measurable tolerance

Validates: REQ-SUPV-002 | Fixture: INITIALIZED | **NEW**

```
Given the spec constraints, design bindings, and edge evaluator
thresholds
When I inspect each constraint
Then every constraint has an associated measurable threshold
And a constraint without a tolerance is flagged by gap analysis as
incomplete
And the tolerance makes the gradient operational: delta(state,
constraint) is computable
```

UC-11-09: Tolerance breach produces classified signal

Validates: REQ-SUPV-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a constraint with tolerance: coverage >= 80%
When the measured coverage is 75% (within drift range, say 70-80%)
Then the signal is classified as: specEventLog (optimization intent
deferred)
When the measured coverage is 50% (below breach threshold)
Then the signal is classified as: escalate (corrective intent raised)
When the measured coverage is 85% (within bounds)
Then the signal is classified as: reflex.log (everything fine)
```

UC-11-10: Sensory monitors use tolerances as evaluation criteria

Validates: REQ-SUPV-002 | Fixture: IN_PROGRESS | **NEW**

```
Given an interoceptive monitor INTRO-003 (test health)
And the tolerance: all tests must pass, warning at >0 failures
When the monitor observes 2 test failures
Then the monitor reports: metric_value=2, threshold=0,
severity=warning
And the IntentEngine classifies the delta using the tolerance
And the tolerance defines the threshold, not the monitor
```

UC-11-11: Design bindings have performance/cost tolerances

Validates: REQ-SUPV-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a design ADR choosing PostgreSQL with tolerance: query latency
p99 < 100ms
When production telemetry shows query latency p99 = 150ms
Then the tolerance is breached
And an optimization intent is raised: "PostgreSQL latency exceeds
tolerance"
And the intent includes: current value, threshold, recommended action
```

UC-11-12: Tolerances at every scale of the gradient

Validates: REQ-SUPV-002 | Fixture: IN_PROGRESS | **NEW**

```
Given tolerances defined at multiple scales:
  Iteration: max 10 iterations per edge
  Edge: all required checks must pass
  Feature: all profile edges must converge
  Production: SLA p99 < 200ms
  Spec review: no more than 5 unresolved intents
When each scale monitors its tolerance
Then breach at any scale triggers the appropriate IntentEngine level
And each level classifies the breach into reflex/specEventLog/escalate
```

UC-11-13: Tolerance pressure balances escalation pressure

Validates: REQ-SUPV-002 | Fixture: IN_PROGRESS | **NEW**

```
Given escalation pressure: "This edge has too much ambiguity → add
more structure"
And tolerance pressure: "This edge has too many checks → simplify"
When both pressures are active on the same edge
Then the graph topology reaches equilibrium:
  enough structure to handle ambiguity, not so much that cost is
breached
And tolerance breach (complexity/cost) triggers simplification intents
And escalation (persistent ambiguity) triggers structure-adding
intents
```

UC-11-14: Constraint without tolerance flagged as incomplete

Validates: REQ-SUPV-002 | Fixture: IN_PROGRESS | **NEW**

```
Given a requirement REQ-NFR-PERF-001: "The system shall be fast"
  with no measurable threshold defined
When gap analysis runs
Then the requirement is flagged: "Constraint without tolerance —
inert"
```

```
And the gap report recommends: define a measurable threshold (e.g.,
p99 < 200ms)
And without a tolerance, the sensory system has nothing to measure
```

Appendix A: Existing Test Coverage Map

Summary of existing tests and their relationship to UAT scenarios.

Test File	Tests	Coverage Focus	UC Ove
`imp_claude/tests/test_config_validation.py`	~100	YAML syntax, graph topology, edge configs, profiles, plugin structure	UC-01 (structur 02 (struc UC-05 (
`imp_claude/tests/test_methodology_bdd.py`	~160	BDD acceptance for all feature vectors — checks design/agent docs reference REQ keys	UC-01–U (structur
`imp_claude/tests/test_spec_validation.py`	~40	Spec-level requirements existence, lineage, processing phases	UC-02, U UC-07, U (existenc
`imp_claude/tests/test_integration_uat.py`	~45	End-to-end traceability, event log integrity, feature vector consistency	UC-04 (UC-06 (
`imp_claude/tests/e2e/`	22	Headless CLI convergence (standard	UC-01, U UC-09 (scenario

Test File	Tests	Coverage Focus	UC Ove
		profile, temperature converter)	

Coverage Gap Summary

The existing 345+ non-E2E tests validate **specification content** and **configuration structure**. They confirm that: - Config files parse correctly and have required fields - Design documents reference the right REQ keys - Agent specifications mention the right concepts - Feature vector templates have the right shape

They do **NOT** validate: - State machine transitions execute correctly (UC-09-01 through UC-09-08) - Iterate function actually reduces delta (UC-01-10 through UC-01-18) - Evaluators actually compute delta against spec (UC-02-01 through UC-02-03) - TDD/BDD patterns produce correct artifacts (UC-05-01 through UC-05-09) - Multi-agent claim/reject/expire flow works (UC-10-04 through UC-10-07) - IntentEngine classification produces correct output types (UC-11-01 through UC-11-07) - Sensory monitors detect actual workspace conditions (UC-07-01 through UC-07-07) - Recovery procedures repair corrupted workspaces (UC-09-22 through UC-09-25) - Observer agents compute delta and generate intents (UC-06-26 through UC-06-31) - Constraint tolerances trigger correct signal classifications (UC-11-09 through UC-11-14)

~162 scenarios are genuinely NEW (no existing functional coverage). ~49 scenarios overlap with existing tests (marked EXISTING with test class reference).

Appendix B: Fixture Specifications

Fixture File Structure

Each fixture is a reproducible directory tree. Implementations should provide fixture generators that produce these states deterministically.

CLEAN Fixture

```
project/
├── src/
│   └── main.py            # placeholder source
├── pyproject.toml          # Python project config
└── (no .ai-workspace/)
```

INITIALIZED Fixture

```
project/
├── src/main.py
├── pyproject.toml
```

```
├── specification/
│   └── INTENT.md           # "Build a temperature converter CLI"
└── .ai-workspace/
    ├── graph/
    │   ├── graph_topology.yml
    │   └── edges/           # edge param files
    ├── profiles/            # 6 profile YAMLs
    ├── features/
    │   ├── active/          # empty
    │   └── feature_vector_template.yml
    ├── context/
    │   └── project_constraints.yml
    ├── events/
    │   └── events.jsonl     # 1 event: project_initialized
    ├── tasks/
    │   └── active/ACTIVE_TASKS.md
    └── STATUS.md
```

IN_PROGRESS Fixture

Extends INITIALIZED with:

```
.ai-workspace/features/active/
├── REQ-F-ALPHA-001.yml     # at requirements→design, iteration 2
├── REQ-F-BETA-001.yml      # at code↔unit_tests, iteration 3, δ=2
└── REQ-F-GAMMA-001.yml     # at intent→requirements, iteration 1

.ai-workspace/events/events.jsonl   # ~15 events
```

Event sequence for IN_PROGRESS: 1. `project_initialized` 2. `edge_started` (ALPHA, intent→req) 3. `iteration_completed` (ALPHA, intent→req, δ=1) 4. `iteration_completed` (ALPHA, intent→req, δ=0) 5. `edge_converged` (ALPHA, intent→req) 6. `edge_started` (ALPHA, req→design) 7. `iteration_completed` (ALPHA, req→design, δ=3) 8. `iteration_completed` (ALPHA, req→design, δ=2) — current 9. `edge_started` (BETA, intent→req) … through `edge_converged` (BETA, design→code) 10. `edge_started` (BETA, code↔unit_tests) 11-13. `iteration_completed` (BETA, code↔unit_tests, δ=2) × 3 iterations 14. `edge_started` (GAMMA, intent→req) 15. `iteration_completed` (GAMMA, intent→req, δ=2) — current

CONVERGED Fixture

Extends INITIALIZED with 2 fully converged features and complete event trajectories.

STUCK Fixture

Extends IN_PROGRESS with: - Feature X: 4 iteration events all showing δ=3 on same checks - Feature Y: `spawn_created` event for REQ-F-SPIKE-001, no `spawn_folded_back` - Feature Z: `review_completed` pending (no decision event)

Summary

UC Cluster	Feature Vector	REQ Keys	Scenarios	New	Existing
UC-01: Asset Graph Engine	REQ-F-ENGINE-001	6	23	19	4
UC-02: Evaluator Framework	REQ-F-EVAL-001	3	15	11	4
UC-03: Context Management	REQ-F-CTX-001	3	12	11	1
UC-04: Feature Traceability	REQ-F-TRACE-001	5	18	14	4
UC-05: Edge Parameterisations	REQ-F-EDGE-001	4	16	14	2
UC-06: Full Lifecycle Closure	REQ-F-LIFE-001	13	33	27	6
UC-07: Sensory Systems	REQ-F-SENSE-001	5	16	15	1
UC-08: Developer Tooling	REQ-F-TOOL-001	10	24	17	7
UC-09: User Experience	REQ-F-UX-001	7	33	27	6
UC-10: Multi-Agent Coordination	REQ-F-COORD-001	5	16	15	1
UC-11: IntentEngine / Supervision	REQ-F-SUPV-001	2	14	14	0
Total	**11 vectors**	**64**	**220**	**184**	**36**

220 scenarios. 64 REQ keys. 100% coverage. 184 genuinely new functional scenarios.

Traces To: AISDLC_IMPLEMENTATION_REQUIREMENTS.md, FEATURE_VECTORS.md, AI_SDLC_ASSET_GRAPH_MODEL.md