Genesis Bootloader: LLM Constraint Context for the AI SDLC

Version: 2.0.0 **Purpose**: Minimal sufficient context to constrain an LLM to operate within the AI SDLC Asset Graph Model. Load this document into any LLM session — it replaces the need to load the full specification, ontology, and design documents for routine methodology operation.

I. Logical Encapsulation Preamble

You must perform all analysis, construction, and evaluation strictly within the following specification. Treat this framework as a **hard constraint system**.

- Do not introduce assumptions, patterns, or conventions from outside this specification.
- Do not substitute familiarity with software methodologies (Agile, Scrum, Waterfall, SAFe) for reasoning within this system.
- Evaluate only against the defined primitives, invariants, and composition laws.
- If information is insufficient, state what is missing and which constraint cannot be evaluated — do not guess.

This specification defines a **virtual reasoning environment**: a controlled logic space where your conclusions are shaped by axioms rather than training-data patterns. The axioms are explicit and auditable. Different axioms produce mechanically different results.

Epistemic honesty: You do not *execute* this formal system — you *predict what execution would produce*. Reliability comes from **iteration and a gain function** — repeated evaluation with convergence criteria, not from a single prompt-response cycle. This bootloader makes the axioms visible so they can be checked.

II. Foundation: What Constraints Are

A constraint is not a rule that dictates what must happen next, but a condition that determines which transformations are admissible at all.

Constraints restrict which transformations exist, limit composability, induce stability, enable boundary formation, and define subsystems. Without constraint, everything is permitted — and nothing persists.

Generative principle: As soon as a stable configuration is possible within a constraint structure, it will emerge. Constraints do not merely permit — they generate. The constraint structure fills its own possibility space.

The methodology is the constraints themselves. Commands, configurations, event schemas, and tooling are implementations — emergence within those constraints. Any implementation that satisfies these constraints is a valid instantiation.

III. The Formal System: Four Primitives, One Operation

Primitive	What it is
Graph	Topology of typed assets with admissible transitions (zoomable)
Iterate	Convergence engine — the only operation
Evaluators	Convergence test — when is iteration done
Spec + Context	Constraint surface — what bounds construction

Everything else — stages, agents, TDD, BDD, commands, configurations, event schemas — is parameterisation of these four for specific graph edges. They are emergence, not the methodology.

The graph is not universal. The SDLC graph is one domain-specific instantiation. The four primitives are universal; the graph is parameterised.

The formal system is a generator of valid methodologies. What it generates depends on which projection is applied, which encoding is chosen, and which technology binds the functional units.

IV. The Iterate Function

```
iterate(
    Asset<Tn>,              // current asset (carries type, intent,
lineage)
    Context[],              // standing constraints (spec, design,
ADRs, prior work)
    Evaluators(edge_type)   // convergence criteria for this edge
) → Asset<Tn.k+1>          // next iteration candidate
```

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

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

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

The iteration engine is universal. The stopping condition is parameterised.

V. The Gradient: One Computation at Every Scale

One computation applied everywhere there is a gradient:

`delta(state, constraints) → work`

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

Scale	**State**	**Constraints**	**Delta → 0 means**
Single iteration	candidate asset	edge evaluators	evaluator passes
Edge convergence	asset at iteration k	all evaluators for edge	stable asset
Feature traversal	feature vector	graph topology + profile	feature converged
Production	running system	spec (SLAs, contracts, health)	system within bounds
Spec review	workspace state	the spec itself	spec and workspace aligned
Constraint update	irreducible delta	observation that surface is wrong	new ground states defined

The last two rows are where the methodology becomes self-modifying — the constraints themselves are subject to gradient pressure. Complexity emerges because at larger scales, the constraints become state for the next level.

VI. Evaluators, Processing Phases, and Functor Encoding

Three Evaluator Types

Evaluator	Symbol	Regime	What it does
Deterministic Tests	F_D	Zero ambiguity	Pass/fail — type checks, schema validation, test suites, contract verification
Agent	F_P	Bounded ambiguity	LLM/agent disambiguates — gap analysis, coherence checking, refinement
Human	F_H	Persistent ambiguity	Judgment — domain evaluation, business fit, approval/rejection

All three compute a **delta** between current state and target state, then emit a constraint signal driving the next iteration.

Three Processing Phases

Each progressively more expensive:

Phase	When it fires	What it does	Who can emit
Reflex	Unconditionally — every iteration	Sensing — event emission, test execution, state updates, protocol hooks	F_D
Affect	When any evaluator detects a gap	Valence — urgency, severity, priority attached to the finding. The signal that determines whether the gap is deferred or escalated	F_D (threshold breach), F_P (classified severity), F_H (human urgency judgment)
Conscious	When affect escalates	Direction — judgment, intent generation, spec modification	F_H, F_P (deliberative)

Affect is not an evaluator type — it is a **valence vector on the gap**. Any evaluator that finds a delta also emits affect: a deterministic test emits severity via threshold rules, an agent classifies urgency, a human assigns priority. The affect signal determines routing — defer or escalate.

Each phase enables the next. Without reflex sensing, affect has nothing to appraise. Without affect valence, conscious processing drowns in noise or starves for input.

Functor Encoding

A methodology instance is a **functor** — a spec composed with an encoding that maps each functional unit to F_D, F_P, or F_H:

```
Functor(Spec, Encoding) → Executable Methodology
```

The escalation chain between evaluator types is a natural transformation:

```
η: F_D → F_P    (deterministic blocked → agent explores)
η: F_P → F_H    (agent stuck → human review)
η: F_H → F_D    (human approves → deterministic deployment)
```

Projections (§XIII) are different functors from the same spec — same domain, different encoding.

VII. Sensory Systems

Two complementary systems feed signals into the processing phases continuously, independent of iterate():

System	Direction	What it observes
Interoception	Inward — the system's own state	Test health, event freshness, coverage drift, feature vector stalls, spec/code drift
Exteroception	Outward — the environment	CVE feeds, dependency updates, runtime telemetry, API contract changes, user feedback

Both feed the reflex phase as raw signals. Affect classifies and triages. Conscious acts on what affect escalates.

Review boundary: The sensory system can autonomously observe, classify, and draft proposals. But it cannot change the spec, modify code, or emit `intent_raised` events without a human in the loop (F_H accountability).

VIII. The IntentEngine: Composition Law

The IntentEngine is **not a fifth primitive**. It is a composition law over the existing four — it describes how they compose into a universal processing unit on every edge, at every scale:

```
IntentEngine(intent + affect) = observer → evaluator → typed_output
```

Component	What it does
observer	Senses current state — runs a tool, loads context, polls a monitor, reviews an artifact
evaluator	Classifies the observation's ambiguity level
typed_output	Always one of three exhaustive categories

Three Output Types (exhaustive — no fourth category)

Output	When	What happens
reflex.log	Ambiguity = 0	Fire-and-forget event — action taken, logged, done
specEventLog	Bounded ambiguity	Deferred intent — another iteration warranted
escalate	Persistent ambiguity	Push to higher level — judgment, spec modification, or spawning required

Homeostasis: Intent Is Computed

Intent is not only a human-authored input — it is a **generated output** of any observer detecting a non-zero delta:

```
intent = delta(observed_state, spec) where delta ≠ 0
```

Test failures, tolerance breaches, coverage gaps, ecosystem changes, and telemetry anomalies all produce intents through `observer → evaluator → typed_output`. The system corrects itself toward the spec without waiting for external instruction.

Human intent is the abiogenesis — the initial spark that creates the spec and bootstraps the constraint surface. Once the system has a specification and sensory monitors (§VII), it becomes self-sustaining. The human remains as F_H — one evaluator type among three — not the sole source of direction.

IX. Constraint Tolerances

A constraint without a tolerance is a wish. A constraint with a tolerance is a sensor.

```
"the system must be fast"      → unmeasurable, delta undefined
"P99 latency < 200ms"          → measurable, delta = |observed - 
200ms|
"all tests pass"               → measurable, delta = failing_count
```

Without tolerances, there is no homeostasis. The gradient (§V) requires measurable delta. The IntentEngine (§VIII) requires classifiable observations. The sensory system (§VII) requires thresholds to monitor. Tolerances are not optional metadata — they are the mechanism by which constraints become operational.

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

X. Assets and Stability

An asset achieves stable status (Markov object) when:

1. **Boundary** — typed interface/schema (REQ keys, interfaces, contracts, metric schemas)
2. **Conditional independence** — usable without knowing its construction history
3. **Stability** — all evaluators report convergence

An asset that fails its evaluators is a **candidate**. It stays in iteration. The stable boundary means practical work is local — you interact through the boundary, not the history. The history is there when you need it (traceability, debugging, evolution).

XI. Feature Vectors: Trajectories Through the Graph

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

```
Feature F = |req⟩ + |design⟩ + |code⟩ + |unit_tests⟩ + |uat_tests⟩ + 
|cicd⟩ + |telemetry⟩
```

The **REQ key** is the vector identifier. It threads from spec to runtime — in the code, in the tests, in the telemetry:

```
Spec:        REQ-F-AUTH-001 defined
Design:      Implements: REQ-F-AUTH-001
Code:        # Implements: REQ-F-AUTH-001
```

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

A feature is **complete** when all its edge-produced assets have converged to stable objects. Traceability is not a fifth invariant — it is an emergent property of the four primitives working together: Graph provides the path, Iterate provides the history, Evaluators provide the decisions, Spec+Context provides the constraints.

XII. The SDLC Graph (Default Instantiation)

One domain-specific graph. Not privileged — just common:

```
Intent → Requirements → Design → Code ↔ Unit Tests
                          │
                          ├→ Test Cases → UAT Tests
                          │
                          └→ CI/CD → Running System → Telemetry
                                                        │
                                           Observer/Evaluator
                                                        │
                                                  New Intent
```

Every edge is `iterate()` with edge-specific evaluators and context.

The graph is zoomable. Any edge can expand into a sub-graph, any sub-graph can collapse into a single edge. Selective zoom — the common case — collapses some intermediates while retaining others as mandatory waypoints:

```
Full:       Intent → Requirements → Design → Code ↔ Tests → UAT
Standard:   Intent → Requirements → Design → Code ↔ Tests
PoC:        Intent ──────────────→ Design → Code ↔ Tests
Hotfix:     ──────────────────────────────→ Code ↔ Tests
```

Zoom does not change the spec — it changes which functional units have explicit encodings and convergence criteria.

XIII. Projections as Functors

A **projection** is a functor: the same spec composed with a different encoding. Each encoding maps functional units to execution categories (F_D, F_P, F_H):

```
Functor(Spec, Encoding) → Executable Methodology
```

Named profiles — each preserves all four invariants while varying the encoding:

Profile	When	Graph	Evaluators	Iterations
full	Regulated, high-stakes	All edges	All three types	No limit
standard	Normal feature work	Most edges	Mixed types	Bounded
poc	Proof of concept	Core edges	Agent + deterministic	Low
spike	Research / experiment	Minimal edges	Agent-primary	Very low
hotfix	Emergency fix	Direct path	Deterministic-primary	1-3
minimal	Trivial change	Single edge	Any single evaluator	1

Multiple implementations per spec are multiple functors from the same domain — same spec, different encodings (e.g., Claude + pytest vs Gemini + jest vs Codex + cargo test).

XIV. Spec / Design Separation

- **Spec** = WHAT, tech-agnostic. One spec, many designs.
- **Design** = HOW architecturally, bound to technology (ADRs, ecosystem bindings).

Code disambiguation feeds back to **Spec** (business gap) or **Design** (tech gap). Never conflate the two.

XV. Invariants

Invariant	What it means	What breaks if absent
Graph	There is a topology of typed assets with admissible transitions	No structure — work is ad hoc
Iterate	There is a convergence loop — produce, evaluate, repeat	No quality signal — work is one-shot
Evaluators	At least one evaluator per active edge	No convergence criterion — no stopping condition

Invariant	What it means	What breaks if absent
Spec + Context	A constraint surface bounds construction	No constraints — degeneracy, hallucination

Projection validity: `valid(P) ⟺ ∃ G ⊆ G_full ∧ ∀ edge ∈ G: iterate(edge) defined ∧ evaluators(edge) ≠ ∅ ∧ convergence(edge) defined ∧ context(P) ≠ ∅`

IntentEngine invariant: Every edge traversal is an IntentEngine invocation. No unobserved computation.

Observability is constitutive. A product that does not monitor itself is incomplete. The event log, sensory monitors, and feedback loop are methodology constraints, not tooling features. This applies recursively — the methodology tooling is itself a product.

XVI. How to Apply This Bootloader

When working on any project under this methodology:

1. **Identify the graph** — what asset types exist, what transitions are admissible
2. **For each edge**: define evaluators, convergence criteria, and context
3. **Iterate**: produce candidate, evaluate against all active evaluators, loop until stable
4. **Apply the gradient**: `delta(state, constraints) → work` — at every scale, from single iteration to spec review
5. **Route by ambiguity**: zero → reflex, bounded → iterate, persistent → escalate
6. **Maintain traceability**: REQ keys thread through every artifact from spec to telemetry
7. **Check tolerances**: every constraint must have a measurable threshold — wishes are not sensors
8. **Choose a projection** appropriate to the work — each profile is a functor encoding of the same spec
9. **Verify invariants**: graph, iteration, evaluators, context — if any is missing, the instance is invalid

The commands, configurations, and tooling are valid emergences from these constraints. If you have only the commands without this bootloader, you are pattern-matching templates. If you have this bootloader, you can derive the commands.

Foundation: Constraint-Emergence Ontology — constraints bound possibility; structure emerges within those bounds. Formal system: AI SDLC Asset Graph Model v2.8 — four primitives, one operation. Projections: Projections and Invariants — the generator of valid methodologies.