**KCE+ — System Definition, Reasoning, Validation (Canonical Spec v1)**

**1 — One-line definition**

**KCE+ (Knowledge Cartography & Creative Engine)** is a domain-agnostic, layered engine that represents, navigates, presents, and (optionally) generates knowledge at any level of abstraction while enforcing a rulebook that preserves provenance, prevents collapse, and guarantees representability across presentation modes.

**2 — Purpose & Scope (What it does)**

KCE+ exists to let humans or agents:

* **Discover** — locate and contextualize concepts across domains.
* **Understand** — present explanations and comparisons at appropriate granularity.
* **Navigate** — move up/down/sideways through knowledge without overload.
* **Corroborate** — store and consult provenance to reduce hallucination.
* **Generate** — produce new, verifiable artifacts or proposals under governance.

Scope: universal (any human knowledge domain), but deliberately **explorable**: maps are delivered as bounded slices on demand, not dumps. KCE+ is a *platform for knowledge work* (learning, engineering, research, productization), not a raw datastore.

**3 — Core behaviors (How it does it)**

KCE+ performs a small set of composable transformations in sequence and with governance:

1. **Ingest / Canonicalize** — identify atomic referents (nodes) and normalize them.
2. **Relate** — build typed relations linking nodes (structural, functional, causal, semantic, etc.).
3. **Attribute** — attach typed metadata (ids, descriptions, quantitative, provenance, lifecycle, meaning-tags).
4. **Project** — produce bounded, purpose-driven subgraphs (structural view, workflow view, time slice, attribute filter).
5. **Navigate** — provide controlled traversal operations (up, down, siblings, non-adjacent jumps, temporal).
6. **Present** — render slices in mode requested (text, table, JSON, code, diagram, narrative).
7. **Generate / Execute (optional)** — apply operator pipelines to produce artifacts or trigger actions; all outputs are versioned and provenanced.

All transformations respect the **Rulebook** (see §5), which enforces recursion bounds, representability, provenance, confidence thresholds, and governance checks.

**4 — Structural design rationale (Why it’s the way it is)**

Design principle: **minimal-complete modularity**. The system must be universal across domains while remaining lean and navigable.

* **Layering**: Knowledge tasks naturally factor into transformations: naming (Node), linking (Relation), describing (Attribute), slicing (Projection), traversing (Navigation), rendering (Presentation), and optionally creating/acting (Creative/Execution). Each transformation is necessary and functionally distinct; collapsing them creates ambiguity and functional coupling that's hard to reason about or secure.
* **Rules separate from Layers**: Constraints (recursion limits, representability, provenance) cut across layers and must be enforced globally; therefore rules are a separate system-level component, not a layer.
* **Explorable, not exhaustive**: Cognitive load and trust require that only the relevant slice be exposed; this is enforced at projection + navigation levels.
* **Provenance-first**: Every persistent claim must have provenance and confidence metadata; this is the primary defense against hallucination and the backbone of trust.
* **Representability contract**: A projection must be convertible to requested presentation modes (or explicitly degraded); this prevents "unpresentable" internal forms.

**5 — System-level components (What the system includes) and why each is required**

**A. Layers (the transformation pipeline)**

1. **Node Layer** — atomic referents (Entity, Process, Agent, Artifact, Context).  
   *Why:* Foundations: you cannot relate or reason without clear referents.
2. **Relation Layer** — typed edges (is\_a, part\_of, used\_in, enables, depends\_on, leads\_to, applied\_in, conflicts\_with, temporal qualifiers).  
   *Why:* Captures structure, function, and causality; necessary for reasoning, explanations, and traversal.
3. **Attribute Layer** — typed properties (identifiers, descriptions, quantitative metrics, qualitative states, provenance, lifecycle, meaning-tags).  
   *Why:* Enables analytics, trust, contextualization, and presentation.
4. **Projection Layer** — view generation (structural, functional/pipeline, temporal, contextual, attribute-based, agent-centric, cross-domain).  
   *Why:* Prevents overload and provides task-specific context slices.
5. **Navigation Layer** — traversal operations and constraints (up/down/horizontal/non-adjacent/temporal/attribute/agent-centric).  
   *Why:* Humans and agents need controlled movement through maps; navigation enforces bounded exploration.
6. **Presentation Layer** — concretely renderable modes (textual, tabular, structured JSON/YAML, code/pseudocode, diagram (Mermaid/DOT/ASCII), narrative/hybrid).  
   *Why:* Ensures outputs are consumable by humans, robots, or downstream tools.
7. **Creative/Execution Layer (optional, gated)** — operators and execution adapters producing artifacts or actions (pattern miners, remix/mutate/transpose operators, evaluators, execution adapters guarded by policy).  
   *Why:* Moves the system from descriptive to generative/actionable; necessary for automation and production use-cases. Must be explicit and governed.

**B. Rules & Rulebook (cross-cutting)**

* **Recursion & Reification rules**: how and when nodes/relations/attributes may be reified; collapse rules to prevent infinite regress.
* **Representability rules**: projection → presentation contracts, deterministic degradation pathways.
* **Provenance & Confidence rules**: every persistent change requires provenance; verifier scoring; thresholds for auto-commit vs claimed queue vs reject.
* **Safety/Governance rules**: thresholds, human gates, policy enforcement for external actions.
* **Minimality & Type invariants**: type exclusivity at creation, canonicalization constraints, no ambiguous dual-type nodes without explicit reification.

*Why:* Rules guarantee correctness, auditability, and safety. They make the architecture usable in practice.

**C. State & Memory**

* Graph DB for nodes/edges/attributes; versioning, snapshots, tombstones, audit log, and vector index for embeddings.  
  *Why:* Persistence is necessary for learning, provenance, diffing, rollback, and reuse.

**D. Interfaces**

* Ingest APIs (chat delta, file import), Graph API (CRUD + queries), Presentation API (present slice in mode), Execution API (trigger adapters), Exporters.  
  *Why:* System must interoperate with users, agents, and external tools.

**E. Feedback & Adaptation**

* Verifier pipeline (ensemble), human-in-the-loop queues, automated pruning/merge heuristics, learning loops that promote patterns with high evaluation.  
  *Why:* Prevents decay, improves fidelity, and supports creative/production loops.

**6 — Completeness & minimality argument (Why nothing essential is missing)**

We use a constructive necessity proof pattern:

* **Existence of referents** → If you can’t name things, you can’t map them → Node Layer required.
* **Existence of relations** → If you can’t connect things, no reasoning, no pipelines → Relation Layer required.
* **Need for properties** → If you can’t describe things (metrics, provenance), you can’t evaluate, rank, or present reliably → Attribute Layer required.
* **Human cognition** → Humans need bounded views → Projection Layer required.
* **Traversal & task flow** → Users/agents must navigate → Navigation Layer required.
* **Consumption requirements** → Outputs must be consumable in multiple modes → Presentation Layer required.
* **Action & creation** → To move from mapping to impact (business, research), you need generation/execution primitives (gated) → Creative/Execution required.

For each of these, removing the layer breaks a whole class of requirements (learning, navigation, trust, action). Supplementary systems (vector DBs, LLMs, visualization libs) are *implementations*; the above are the minimal logical primitives.

Therefore the stack is **minimal-complete**: no smaller set offers the same universal capabilities; adding more base layers duplicates function or reduces clarity.

**7 — Failure modes, falsification tests, and acceptance criteria**

Any architectural claim must be falsifiable. Here are tests and acceptance rules.

**A. Unit tests / Acceptance criteria (per component)**

* **Node Layer**: Represent 200 diverse concepts (biology, law, CS). Acceptance: ≤2% need for new node classes.
* **Relation Layer**: Encode 50 canonical process chains and 50 taxonomies; Acceptance: all required connections expressible with existing relation families.
* **Attribute Layer**: Attach provenance + confidence to nodes/edges; Acceptance: every persistent change in test set has provenance traceable to source.
* **Projection Layer**: Given a topic, produce minimal/intermediate/full scopes obedient to user limits. Acceptance: user selects scope with <2 clarifying steps.
* **Navigation Layer**: Perform up/down/non-adjacent traversals with bounded token slices. Acceptance: p95 slice size under an agreed budget; no infinite expansion.
* **Presentation Layer**: Each projection must render to requested mode or return deterministic degradation. Acceptance: conversion success rate >98% across modes.
* **Creative/Execution Layer**: Generate artifacts with provenance and evaluator scores; Acceptance: evaluator correlates with human judgment (Pearson r>0.6) on test tasks and safety gates prevent >99% of disallowed actions.

**B. System-wide falsification tests**

1. **Hallucination injection test**: Inject fabricated claims via LLM-simulated deltas; verifiers must flag and route <threshold auto-commit. If >threshold false commits occur, rule/pipeline broken.
2. **Collapse test**: Attempt to reify attributes ad infinitum; system must enforce recursion depth/collapse rules. If recursion escapes limits, falsified.
3. **Representability test**: Create complex projection that’s not directly renderable; system must either render or provide deterministic downgrade. Failure falsifies representability rules.
4. **Scalability test**: With simulated growth to N nodes/edges (configurable), query latency and slice sizes must remain within architecture budgets. If not, capacity or model needs rework.

**C. Operational KPIs (ongoing health)**

* Map precision (verified edges ÷ total committed). Target: >90% for matured graphs.
* Token reduction ratio per session (raw chat tokens replaced by slice tokens). Target: ≥10× reduction.
* Provenance completeness (fraction of persistent claims with ≥1 source). Target: 100%.
* Auto-commit rejection rate (false commits flagged). Target: <5% after warm-up.
* User correction rate (manual edits per 100 commits). Target: declines over time.

**8 — Canonical versioning & locking statement**

This document is **KCE+ System Definition v1 (canonical)**. It captures the agreed system boundary, layer stack, rulebook outline, component set, falsification tests, and acceptance criteria. Any change to core primitives (layer set, core rule invariants, or component set) requires a *versioned amendment* with explicit rationale and regression-proof tests.