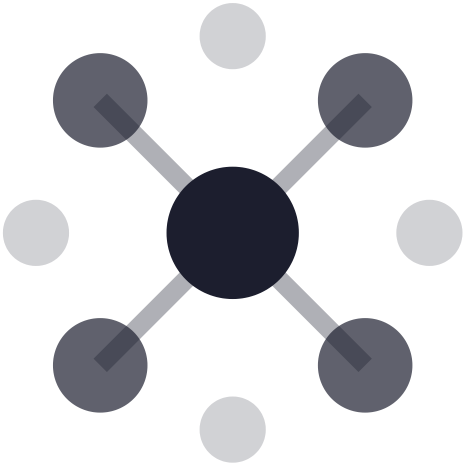
UPG CORE · V0.10.6



Unified Product Graph

Unified Product Graph

An Open Standard for Product Knowledge That Compounds

Faheem Hasan

Arkheiev UG

2026-06-13 COMMIT B24EEB075

UPG CORE · V0.10.6 · 2026-06-13 · COMMIT B24EEB075

Unified Product Graph

An Open Standard for Product Knowledge That Compounds

Faheem Hasan
Arkheiev UG, Berlin, Germany
faheem@theproductcreator.com

ABSTRACT

AI expanded what one person can produce. It did not expand what one person can hold in their head.

A solo builder today can generate fifty product artifacts in an afternoon: personas, jobs-to-be-done, hypotheses, features, pricing tiers, all through conversation with a large language model. Without structure, those fifty artifacts become loosely connected documents. Relationships are implied rather than declared, lifecycle state lives in the author's memory, and there is usually little traceability from evidence to shipped feature. People have long added structure to documents by hand, but AI now generates faster than that hand-structured layer can keep up. The current paradigm for giving AI durable memory, retrieval-augmented generation (RAG) over vector embeddings, helps models find text that resembles a query. But what product builders need to reason about, decisions, relationships, lifecycle, provenance, cannot be retrieved reliably by similarity. It must be declared. The next session starts near zero, because nothing durable was indexed.

This paper introduces the **Unified Product Graph (UPG)**: an open standard that represents product knowledge as a typed, connected graph. UPG defines a product ontology: stable entity types across semantic domains, directed edge types with forward/reverse verb pairs, a parent-child hierarchy grammar, a lifecycle system, and a portable file format (`.upg`, serialisable as UPG Markdown). The specification is organised into conceptual layers with a strict dependency flow, each layer self-contained, so adopters can take what they need.

UPG is designed for the people who build products (designers, engineers, product managers, marketers, growth) and for the AI tools they work with. We describe the ontology, its architecture, and the tooling ecosystem built around it: a local MCP server, a cloud server, import adapters, guided playbooks and approaches that run inside any AI agent, and a visual graph application. The same model scales from a single product to a portfolio of them, where cross-product edges and a canonical shared-entity registry let knowledge compound across products as well as across sessions.

The central claim is simple: the missing layer in AI-assisted product creation is not more generation capability. It is **structured memory**. When product knowledge is declared rather than inferred, it compounds across sessions, roles, and tools instead of dissolving back into prose. The graph is the memory that neither the human nor the AI alone naturally maintains.

KEYWORDS *product design · product engineering · product management · growth · knowledge graphs · open standards · AI-assisted creation · structured memory · ontology · retrieval-augmented generation · Model Context Protocol*

Contents

MAIN

APPENDICES

BACK MATTER

1. Introduction

1.1 The Production-Comprehension Gap

AI expanded what one person can produce. It did not expand what one person can hold in their head. We call this the **production-comprehension gap**: the widening distance between what AI enables a person to generate and what a person can actually reason about, connect, and act on coherently. As AI capabilities grow, this gap continues to widen.

The gap is less about AI capability than about how product knowledge is stored. A founder working alone can speed-draft dozens of product artifacts in an afternoon: research notes, user flows, architecture sketches, feature specifications, pricing models. They may use different terms for each. Within a week, they have fifty of them. Soon after, they cannot remember which artifact connects to which pain point, which hypothesis was validated and which was abandoned, or whether the pricing model drafted earlier aligns with the positioning refined later. The artifacts exist. The agent reading them rarely knows the connections between them, and the founder rarely does either.

Documents, whether in Notion, Confluence, Google Docs, or markdown files, are containers for prose. They are not containers for relationships. Modern AI systems soften this limit: vector search, retrieval-augmented generation, and LLM-indexed wikis can surface the persona document when a question is asked about the feature. But similarity is not the same as structure. The connection between persona and feature still lives, implicitly, in the author's head. It is not declared anywhere a tool can reliably traverse, and that memory fades between sessions and does not transfer to the next person or agent on the project.

What is needed is structure for product work that persists across environments, and a framework-agnostic data model that retains what is true about the product as vocabularies and tools change around it.

1.2 Fragmented by Tool, Scattered by Convention

Product knowledge today lives across tools that each own a fragment of the picture. A typical team uses:

- **Notion or Confluence** for strategy documents and research notes
- **Linear, Jira, or Asana** for delivery tracking
- **Figma or FigJam** for design artifacts and whiteboarding
- **Miro or Whimsical** for journey maps and canvases

- **Productboard or Vistaly** for discovery and opportunity trees
- **Sheets or Airtable** for prioritisation and roadmapping

Each tool captures its own domain well, but no single tool captures the cross-domain relationships that hold product thinking together. Moving between them always requires translation, and that translation has traditionally lived in people: the feature in Linear, the persona in Notion that motivated it, the hypothesis in Miro that validated it, and the pricing tier in Sheets that packages it were juggled by the practitioners who wrote them.

That arrangement held while humans were the only ones producing product work. When AI agents generate artifacts at machine speed, the human-translation layer cannot keep up: connections accumulate faster than any person can catalogue them. The structure has to live somewhere other than a practitioner's head.

Tool-direct retrieval helps but does not close the gap. An AI agent that can pull documents from Notion, issues from Linear, and frames from Figma still receives them in parallel: every tool has its own schema, its own vocabulary, its own notion of what a "feature" or a "user" even is. Without a shared schema across tools, the information arrives in the agent's context window as parallel streams, not as a connected graph.

An agent given access to this corpus can do more than retrieve individual documents. It can even infer connections between them by re-reading prose at query time: an emergent graph assembled on demand from similarity and summarisation. But an emergent graph is not a curated one. The relationships it surfaces are plausible, not declared, and they must be re-discovered on every run. The structure that matters for product work is not declared anywhere it can be reliably traversed, versioned, or trusted.

1.3 The Case for a Product Ontology

Closing the production-comprehension gap calls for a structural layer of its own: a shared foundation humans and AI agents can both use to capture the things a product is made of, the relationships between them, and the reasoning that ties them together. Tools are transient; the knowledge they capture should not be. That foundation is an ontology for product knowledge.

An ontology provides three things that neither documents nor individual tools can:

1. **Typed entities.** *A **type** is a named category of things that share the same shape: the same information, the same lifecycle, and the same place in the graph.* A persona is a different type from a feature, and a feature is a different type from an architectural decision. The ontology defines each type UPG covers: what information it carries, what states

it moves through as work progresses, and where it sits in relation to the others. Once something is recognised as a persona, every tool or AI agent that speaks the ontology knows how to handle it without relearning what a persona is.

2. **Directed relationships.** *A **relationship** between two entities has a name that tells you what the connection means and a direction that tells you which side is the subject.* A hyperlink tells you *that* two documents are related; it cannot tell you *how*. An ontology names the relationship and gives it direction: a persona *pursues* a job, an opportunity *addresses* a need, a hypothesis *requires* an experiment. Each relationship reads coherently in both directions, so a person or an AI agent can ask *"what does this hypothesis require?"* or *"which hypotheses require this experiment?"* and get a real answer.

3. **Portability.** *Knowledge that lives in a plain, open file any compatible tool can read and write, not inside a vendor's database.* The knowledge lives in a plain `.upg` file. The file travels with the product, not with the vendor that happens to be hosting it this year. You can save it to git, see what changed between two versions, hand it to a teammate, and feed it to any AI agent as structured context. Product knowledge stops being trapped inside a tool you may not want to use next year.

This paper describes one such ontology and the open standard built around it: the **Unified Product Graph (UPG)**. UPG is MIT-licensed, implemented as a set of TypeScript packages, and designed to work inside any AI development environment that speaks the Model Context Protocol, with reference implementations in Claude Code, Cursor, and VS Code.

1.4 The Memory Problem

At the root of the production-comprehension gap is a memory problem. Human working memory is limited to approximately 7 ± 2 chunks (Miller, 1956). Large language model context windows, while much larger, reset between sessions and degrade with the position of information within the window (Liu et al., 2024). Neither provides the persistent, structured, queryable memory that long-running product work requires.

A growing landscape of tools addresses this at the seat level. Karpathy's LLM Wiki proposes an incrementally updated markdown collection that the model curates rather than retrieves over, where synthesis is stored once and re-used instead of re-derived on every query (Karpathy, 2026). Graphify turns a repository of code, docs, and diagrams into a queryable knowledge graph for AI coding assistants, built explicitly in the LLM Wiki lineage (Shamsi, 2026). Claude-mem captures session observations in a local SQLite and vector store, re-injecting relevant context into future Claude Code sessions (thedotmack, 2026). These are valuable where they sit, but each is scoped to one operator, one codebase, or one agent. Product work needs something different: a

memory shared across people, agents, stakeholders, and time, independent of any single tool or installation. Structure that lives on one developer's machine is not structure the next collaborator, or the next agent, inherits.

Documents solve memory poorly. A document is a linear container for prose, which means it must be re-read to be used. The next AI session that needs information from a 1,500-word persona document must re-read the entire document, because the relationships inside it are not indexed as typed data. Retrieval-augmented generation and semantic search can surface the right document, but retrieval is not the same as structure: a retrieved document still has to be re-parsed, and the connections between artifacts remain implicit in prose rather than declared as traversable edges. The result is a fuzzy graph at best, re-assembled on demand, different every time.

A typed graph solves memory differently. Every entity becomes a stable, referenceable record. A persona captured in week one has a stable identity, a defined shape, and declared relationships to the jobs, needs, and outcomes it connects to. In week ten, an AI agent retrieving that persona performs a single structured query rather than a document re-read. More importantly, every entity written during the intervening nine weeks is already connected to it. Output from yesterday's session becomes context for today's without re-contextualising, and without depending on any individual operator's local setup.

> **Compounding.** *A property of structured memory in which each new entity added to the graph does not merely sit alongside existing ones but actively connects to and extends them, so the queryable value of the graph grows faster than its storage cost.*
>
> Concretely: a persona captured in week one is referenced by three hypotheses written in week three, which are refuted by a research insight captured in week five, which motivates a feature shipped in week eight. In week ten, asking *"what evidence supports this feature?"* traverses four typed edges and returns the answer. No re-reading, no re-assembly, no drift. The graph gets denser with every session, and every new entity increases the number of questions the graph can answer. This is the opposite of document accumulation, where each new file is another thing to read from the top. Compounding is what makes a structure count as memory rather than storage, and it is the property that qualifies structured memory as the category of solution the production-comprehension gap requires.

To see compounding concretely, contrast two versions of the same ten-week founder workflow:

Week	Without UPG	With UPG
1	Three persona interviews → three 1,500-word Notion documents	Three persona nodes linked to `job` and `need` nodes; every record fits on half a screen
3	New AI session has no memory; documents re-pasted in full	`list_nodes(type='persona')` returns three structured records in one call
6	Hypothesis cross-check requires pasting documents again	AI cross-checks against each persona's validated needs; flags tension with a specific need that explicitly rejects the pricing assumption
10	Feature prioritisation requires an hour of re-reading to support a one-minute decision	`get_graph_digest()` returns chain completeness per feature; twenty candidates rank themselves by evidence density

Each session starts from zero. Every session starts from where the last one left off.

The argument is not that UPG saves time on any one session. The argument is that UPG makes every session cheaper than the last.

Compounding is not autonomous. The founder still runs the guided playbooks and approaches that produce each entity, the AI still emits `mapping_confidence` annotations that a human reviews, and entities still get merged, deleted, and corrected as the model of the product evolves. The graph is a shared memory, curated jointly by the practitioner and the AI agents they work with. The marginal cost of writing an entity is roughly the marginal cost of writing a document; the marginal value is different, because one disappears into a folder and the other accrues into a queryable structure.

1.5 Contributions

This paper makes four contributions, corresponding to the argument structure of the section just closed.

1. **A framing of structured memory as the category of solution.** The production-comprehension gap (§1.1) is, at root, a memory problem (§1.4), and the property that qualifies a structure as memory rather than storage is compounding: each new entity connects to and extends the existing graph rather than sitting beside it. This paper names that property, defines it precisely, and argues that it is what product work requires.
2. **An ontology for product knowledge.** UPG defines a stable vocabulary of entity types covering the full arc of product creation, from the research insights and personas that shape a product, to the features, architectural decisions, and growth

experiments that ship it, to the organisational structures that surround it. Every type carries a defined shape: the information it holds, the lifecycle it moves through, and the place it occupies in a domain hierarchy organised as concentric rings outward from the product nucleus. Relationships between types are typed and directed, with human-readable verb pairs that carry meaning in both directions. The same ontology extends from a single product to a portfolio of them: cross-product edge types and a portfolio-level registry of canonical shared entities (§3.10) let a company's products share personas, metrics, and foundational specifications rather than duplicate them. The full catalogue of types and edges is given in the Technical Appendix.

3. **A layered architecture for the specification itself.** The ontology is organised into conceptual layers with a strict dependency flow: *catalog* (what the types are), *grammar* (how they combine), *properties* (what information they carry), and *output* (how they are serialised and consumed). Layers are self-contained and can be adopted incrementally, so an implementer can read the catalog without the output layer, or build tooling against the grammar without committing to every property. The architecture is what makes the specification evolvable: each layer can grow without breaking the ones above or below it.

4. **A tooling ecosystem that makes the standard usable in practice.** Local and cloud MCP servers that let any AI agent read and write against a `.upg` file or a remote graph. Import adapters that turn prose from Markdown, Notion, Linear, and GitHub into typed entities. Guided playbooks and approaches that run inside any AI agent speaking the Model Context Protocol. A parser for UPG Markdown, the human-readable serialisation of the format. And a visual graph application for the surfaces where a canvas view is the right interface.

Taken together, these contributions close a single loop. Section 1 has argued that product knowledge needs a foundation that lives beneath any single tool, that this foundation must solve a memory problem, and that compounding is the property that makes structured memory work. The ontology, the architecture, and the tooling are the three layers of that foundation. The rest of the paper describes each in detail and examines the design choices that hold them together.

---

2. Background and Related Work

2.1 The Product Tooling Landscape

Section 1 argued for a foundation that lives beneath any single tool. To locate where such a foundation sits, it helps to start with the tools themselves. Product work today is spread across a landscape of categories that coexist and overlap, each with its own purpose and its own gap.

Work trackers. Jira (2002), Linear (2019), Asana (2008), and GitHub Projects model product work as tasks with status, assignment, and hierarchy. Their ontology is minimal and delivery-oriented: issues, epics, sprints, releases. They answer *what are we building?* Not *why are we building it?*

Discovery platforms. Productboard (2014), Vistaly (2020), and ProductPlan introduce structured discovery artifacts: opportunity solution trees, evidence boards, persona canvases. They capture the *why*, typically as a system disconnected from delivery. The discovery graph and the delivery graph do not share a schema. A validated opportunity in Productboard has no typed connection to a shipped feature in Jira. They answer *why are we building it?* Not *is it actually being built?*

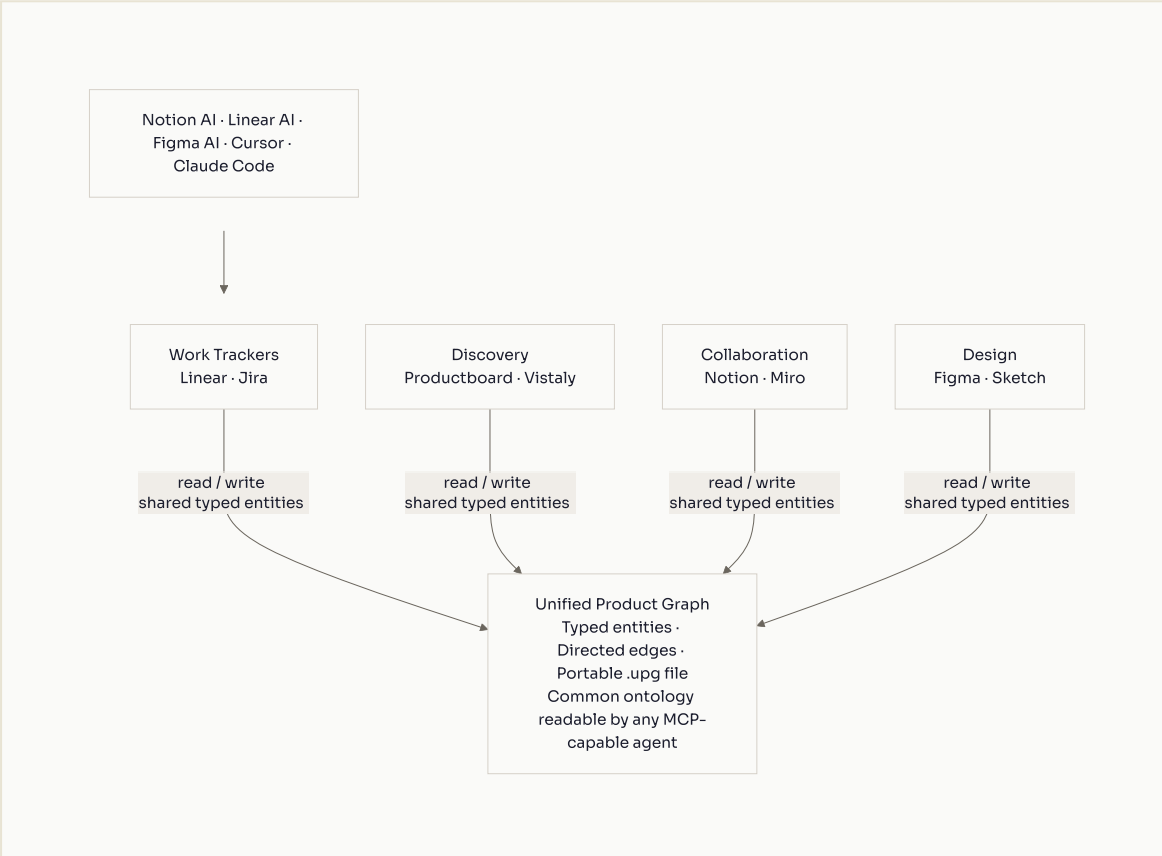
Collaboration surfaces. Notion, Confluence, Google Docs, and Coda on the document side; Miro, Lucidchart, Whimsical, and FigJam on the whiteboarding side. These tools were designed for an era when context was held in people: teams shared a room, a Slack channel, or a whiteboard, and meaning travelled with the humans who wrote and drew. They are deliberately schema-free, and their strength is that flexibility. Their limit is that everything they hold is prose or pixels, not typed data. Each is now adding AI features of its own: Notion AI, Miro AI, FigJam AI. But those features are trapped inside the tool they live in. Without a shared schema across tools, an AI inside Notion cannot reason about a flow in Miro or a design in Figma. They answer *what has the team been thinking?* Not *what has the team decided, and how does it connect to everything else?*

Design tools. Figma, Sketch, and Adobe XD hold the visual artifacts of product work: wireframes, components, flows, prototypes. Their internal ontology (frames, components, variants) is rich for visual composition but blind to the product-level concepts those artifacts represent. A Figma frame named *"Checkout: empty cart"* does not know it depicts a user-journey step that addresses a pain point backed by a research insight. They answer *what will it look like?* Not *what does it address, and why does it matter?*

AI copilots. The most recent layer, which runs across every category above rather than forming its own: Notion AI, Linear AI, Figma AI, Cursor, and Claude Code. Each copilot is a generator, and each generates inside the ontology of the tool it lives in. More

generation, same fragmentation: the AI that drafts a Linear ticket has no structural handle on the persona described in Notion or the component sketched in Figma. They answer *what can we generate next?* Not *how does it fit what already exists?*

Each of these categories solves its problem well within its own scope. What is missing, across the whole landscape, is a shared substrate that describes what a product *is*. The Unified Product Graph (UPG) is designed to sit beneath the landscape rather than within it: a cross-domain ontology spanning strategy, users, discovery, validation, design, engineering, growth, business model, marketing, operations, and organisational structure, so that a work tracker, a discovery platform, a document, a whiteboard, a Figma file, and an AI copilot can all reference the same typed entities rather than each maintain their own.



2.2 Knowledge Graphs and the Formal Tradition

A fair question at this point: if the Unified Product Graph (UPG) is a typed graph with stable identifiers and directed relationships, why does it not use one of the existing standards for knowledge representation? The World Wide Web Consortium has produced a family of such standards over two decades, each with real adoption. It is worth explaining what each is, what UPG implemented in them would look like, and why we chose not to take that path.

RDF: the Resource Description Framework. *A W3C standard (2004) for expressing knowledge as triples of the form subject-predicate-object, where each term is typically a URI.* RDF is the foundation of the Semantic Web vision. In RDF, the statement *"Felix pursues the job of deciding which feature to ship before the holiday window"* would be expressed as three URIs bound together, one for Felix, one for the *pursues* relationship, and one for the job. A graph in RDF is a collection of such triples. RDF Schema (RDFS) adds a light type system on top, letting you declare that *Persona* is a class and *pursues* is a property.

OWL: the Web Ontology Language. *A W3C standard (2004, updated 2012) that extends RDF with richer constructs for formal reasoning.* Where RDF tells you what is, OWL lets you describe what *must* follow. It supports class hierarchies, property restrictions, cardinality constraints, and logical inference. An OWL reasoner can deduce that if *Felix is-a Persona* and every *Persona pursues at least one Job*, then Felix pursues at least one job. OWL comes in three profiles (Lite, DL, Full) that trade expressive power against computational decidability. Biomedical ontologies like SNOMED CT and the Gene Ontology are built on OWL.

schema.org. *A consumer-oriented vocabulary founded in 2011 by Google, Bing, Yahoo, and Yandex, used to annotate structured data on web pages.* schema.org is the most widely deployed structured-data standard in the world. It defines types like *Person*, *Organization*, *Event*, *Product*, *Recipe*, *Review*, with properties for each, and it is typically serialised as JSON-LD embedded in HTML. When a search engine shows a rich result with ratings, dates, or prices extracted from a page, it is reading schema.org annotations. Its scope is deliberately broad and shallow: shared labels any web publisher can use, not formal reasoning.

Honestly sketched, a UPG-in-RDF version of a single persona and a connected feature would translate roughly like this in Turtle syntax:

```
@prefix upg: <https://upg.dev/ontology/> .
@prefix : <https://example.com/products/threadline/> .

:persona-felix
    a upg:Persona ;
    upg:name "Felix, solo builder" ;
    upg:pursues :job-decide-next-feature .

:feature-cross-meeting-search
    a upg:Feature ;
    upg:informed-by :learning-volume-vs-value ;
    upg:in-feature-area :feature-area-search-recall .
```

Layering OWL on top would add class definitions and formal constraints. A minimal OWL treatment of the same types, in OWL's Turtle form, would look like this:

```
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix upg:  <https://upg.dev/ontology/> .

upg:Persona a owl:Class .
upg:Feature a owl:Class .
upg:Opportunity a owl:Class .

upg:pursues a owl:ObjectProperty ;
    rdfs:domain upg:Persona ;
    rdfs:range upg:Job .

upg:implements a owl:ObjectProperty ;
    rdfs:domain upg:Feature ;
    rdfs:range upg:Opportunity .

upg:Feature rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty upg:implements ;
      owl:minCardinality 1 ] .
```

OWL's contribution is the cardinality restriction at the bottom: *every Feature must implement at least one Opportunity*. An OWL reasoner can then catch a graph where a *Feature* exists without a connected *Opportunity*, or infer the missing connection if other constraints imply it. This is the inference machinery UPG does not currently need, since product-work validation is closer to *"does each shipped feature have a research-backed opportunity?"* than to automated theorem proving.

The schema.org treatment of the same information compresses dramatically, because the vocabulary was not designed for product work:

```
{
  "@context": "https://schema.org/",
  "@type": "Person",
  "name": "Felix",
  "description": "Persona pursuing the job of deciding which feature to ship
before the holiday window"
}
```

Persona maps awkwardly to *Person*, which on schema.org is a real human (typically authors, employees, or search-result subjects) rather than a research abstraction. There is no schema.org equivalent for *Feature*, *Opportunity*, *Pain Point*, or *Hypothesis* in the sense a product team uses those words. Capturing the full UPG graph under schema.org would mean either flattening every UPG type into a generic parent like *Thing* or *CreativeWork*, which strips the product-specific semantics, or defining an extension

vocabulary at `upg.dev` and using schema.org only for the small overlap (such as *Organization* for the team). In the latter case the extension does almost all the work, which is the scope-mismatch argument in its concrete form.

All three would work. So why not use them? The decision came down to four factors:

Factor	Why it ruled out RDF/OWL/schema.org
Target consumer	UPG's primary consumers are MCP-integrated AI agents; the tooling surface is TypeScript. RDF and OWL toolchains are Java- and Python-heavy, a translation layer that serves no one
Formal reasoning is not the bottleneck	OWL earns its weight when automated inference is central (biomedical ontologies, regulatory knowledge bases). Product work needs capture, traversal, lifecycle, and validation, all achievable with a TypeScript validator
Practitioner readability	RDF Turtle is specialist-readable; OWL is not readable by practitioners at all. UPG is designed to be readable twice: as JSON for tools, and as UPG Markdown for people
Scope mismatch	schema.org's *Product* type is a consumer-commerce product, not a product-being-built. Expressing UPG in schema.org collapses into generic parents or requires an extension vocabulary that carries almost the whole model

What UPG inherits from this tradition is the core idea: typed entities, typed directed relationships, stable identifiers, and an open serialisable format. What UPG departs from is the implementation substrate, TypeScript rather than RDF/OWL; the lifecycle treatment, baked in rather than patched on; the edge vocabulary, human-readable verb pairs on every relationship; and the scope, a vocabulary shaped by product work rather than by the open web.

2.3 Emergent Graphs from Documents

Section 1.4 introduced a family of tools that extract structure from prose on demand: Karpathy's LLM Wiki, Graphify, and claude-mem, each operating at the seat level. The same pattern scales up. Microsoft's GraphRAG (Edge et al., 2024) applies LLM-driven extraction across enterprise document corpora, building community summaries and cross-document graphs that outperform flat retrieval on comprehensiveness. A broader class of LLM-assisted knowledge-graph extraction tools turns any input (code, documents, papers, screenshots) into a clustered graph in minutes.

These systems share a philosophy with UPG: *the AI should maintain the structure, not the human.* They also produce measurable gains over unstructured retrieval. GraphRAG reports 72–83% comprehensiveness wins over baseline RAG; graph-augmented

benchmarks triple LLM accuracy on schema-intensive questions (data.world, 2024). The case *for* structure is settled. The case for *which* structure is not.

The distinction is between **emergent graphs** and **curated graphs**.

Emergent graphs describe what a corpus contains. They read documents and surface the concepts mentioned, the entities named, and the soft associations between them. They are excellent for personal knowledge synthesis, exploratory reading, and retrieval over unstructured material. They reflect what you *have written*.

Curated graphs describe what a practitioner has decided. Entities are created deliberately, with known types, stable identifiers, typed edges, and lifecycle status. Edges carry verbs with direction. The graph does not reflect what you have written. It reflects what you have *resolved, connected, and committed to building*.

For product creation, five differences matter:

Dimension	Emergent graph	Curated graph (UPG)
Absence	Surfaces corpus-level absence (*"no documents discuss pricing"*)	Surfaces ontological absence: the entities the domain *expects* but the graph doesn't have
Connections	Soft associations based on co-occurrence or similarity	Named, typed edges: `addresses`, `pursued_by`, `requires`, traversable in either direction
Identity	Re-derived on each extraction; `node_42` this week may be `node_87` next	Stable opaque IDs (`n_SuIk0TASeSWJRFaf`) that survive sessions, tool changes, and schema evolution
Interchange	Each extraction reflects its source's vocabulary; two tools produce two incompatible graphs	A common ontology: a Linear issue connects to a Figma component through a typed edge
Semantics	A `feature` is a label; a flat graph	A `feature` carries a lifecycle, evidence requirements, and a traceability chain to user interviews

Emergent and curated graphs are not mutually exclusive. A product team may well want both: emergent retrieval over the raw document corpus (meeting notes, Slack threads, user interview transcripts) and a curated graph of the decisions, hypotheses, and shipped work those documents produced. UPG takes the curated approach because product creation is a domain with lifecycles, evidence chains, and resolutions, not a corpus of interchangeable text.

> **The wiki and the graph.** *The wiki holds what you know. The graph holds what you have decided, validated, connected, and committed to building.*
>
> Both can coexist. A team that writes extensively in Notion or Confluence can extract an emergent graph from its pages and still maintain a curated UPG of its product. The two structures answer different questions. Ask the wiki *"what have we been thinking about?"* and it returns pages and paragraphs that mention the topic. Ask the graph *"what have we decided to build, and what is its evidence?"* and it returns typed entities connected by typed edges, traceable end to end. UPG is the second structure, because product work is the second question.

2.4 The Model Context Protocol

A curated graph is only useful if AI agents can read from it and write into it without custom integration work per tool. The Model Context Protocol (MCP), introduced by Anthropic in 2024, is the emerging standard that makes this possible.

MCP. *An open protocol that defines a uniform client-server interface for AI assistants to discover and invoke external tools and data sources.* Before MCP, integrating an AI assistant with an external system meant writing bespoke glue for that pair. An assistant that could read Jira could not, without more work, read Linear; an assistant that could search Notion could not, without more work, query a vector store. Each integration was a custom contract between one client and one backend. MCP abstracts that contract into a protocol: servers expose tools, resources, and prompts through a standard schema, and any MCP-capable client can discover and call them the same way. The cost of adding a new data source drops from *"a few weeks of integration work per client"* to *"run an MCP server once, any client can use it."*

The protocol grew quickly because the problem it solves is load-bearing for every AI assistant. Claude Code, Cursor, VS Code, Zed, Continue, and a growing list of editors and chat interfaces ship with MCP support. Servers now exist for databases, filesystems, Git hosts, design tools, and most of the category-leading SaaS products. The interoperability dividend is that a single MCP-native tool becomes addressable from every MCP-aware client, without the vendor having to build a separate integration for each.

The Unified Product Graph (UPG) is MCP-native. The specification's primary read/write interface is the UPG MCP server (`@unified-product-graph/mcp-server`), which exposes CRUD operations on nodes and edges, graph traversal queries, batch mutations, schema introspection, and validation against the ontology. A companion cloud server (`@unified-product-graph/cloud-server`) does the same over a remote

graph backed by a Postgres store, for teams that want a shared graph rather than a local file. Any MCP-capable client (Claude Code, Cursor, Zed, Continue, or a custom in-house agent) reads, writes, and reasons about product graphs without bespoke integration. The agent's side of the contract is the MCP tool list; UPG's side is the graph operations those tools map to.

Two design choices follow from the MCP-native posture. First, UPG does not invent a new wire protocol. The specification focuses on the ontology itself, what the types are, how they connect, what their lifecycles look like, without also specifying how agents call into it. Second, the surface UPG presents to AI agents is uniform with the surfaces presented by every other MCP server the agent already knows how to use. From the agent's point of view, UPG is one more tool it can call, distinguishable by what it operates on rather than by how it is invoked.

The trade is acceptance. By building on MCP rather than a proprietary API, UPG inherits the protocol's reach and its constraints. If MCP changes, UPG follows. If an agent cannot speak MCP, it cannot speak UPG without an adapter. For the current AI-agent landscape, that trade is clearly the right direction: MCP is becoming the default way agents reach out into the world, and UPG is designed to be reachable.

2.5 Frameworks as View Definitions

Every discipline that contributes to product creation has its own frameworks, and every framework has its own vocabulary. Product management uses Opportunity Solution Trees, Jobs-to-be-Done, Lean Canvas, and the Business Model Canvas. Design uses empathy maps, user journey maps, and design-thinking stages. Engineering uses domain-driven design, the C4 model, and architecture decision records. Growth uses funnels, loops, and aha-moment frameworks. Each framework is a trusted method for thinking about part of the product, and each brings real clarity within its own domain.

Frameworks collide when they describe the same thing. A *Pain Point* in Lean Canvas is a *Need* in Jobs-to-be-Done is a *Frustration* in an empathy map is an *Unmet Need* in a user research report. A *Feature* in Jira is a *Solution* in an Opportunity Solution Tree is a *Capability* in a capability map is a *Service* in a domain-driven design diagram. The concepts are shared across disciplines. The labels are not.

A cross-discipline map makes the collisions visible. The following table shows a handful of recurring concepts, the names they travel under across product management, design, engineering, and growth, and the canonical UPG type each one maps to:

Concept	Product management	Design	Engineering	Growth	UPG canonical type
The user	Persona (discovery), Buyer	User archetype	Actor (use case diagram)	Segment (marketing), Cohort	`persona`
Unmet user state	Pain Point (Lean Canvas), Need (JTBD)	Frustration (empathy map), Pain (journey map)	—	Friction (funnel analysis)	`need` (with `valence: pain`
Discovery-level bet	Opportunity (OST), Bet	Design question	—	Growth bet, Experiment hypothesis	`opportunity`
Strategic goal	Outcome (OKR), Objective	Journey goal	SLO (SRE)	North Star Metric	`outcome`
Thing to build	Feature (Jira), Solution (OST)	Design (Figma frame), Flow (user flow)	Service (DDD), Component, Module	Experiment (growth)	`feature`
Evidence for a claim	Validated Learning (Lean Startup), Insight (research)	Research finding, Usability observation	Data point, Telemetry signal	Experiment result (A/B)	`insight` (from research), `learning` (from experiment)

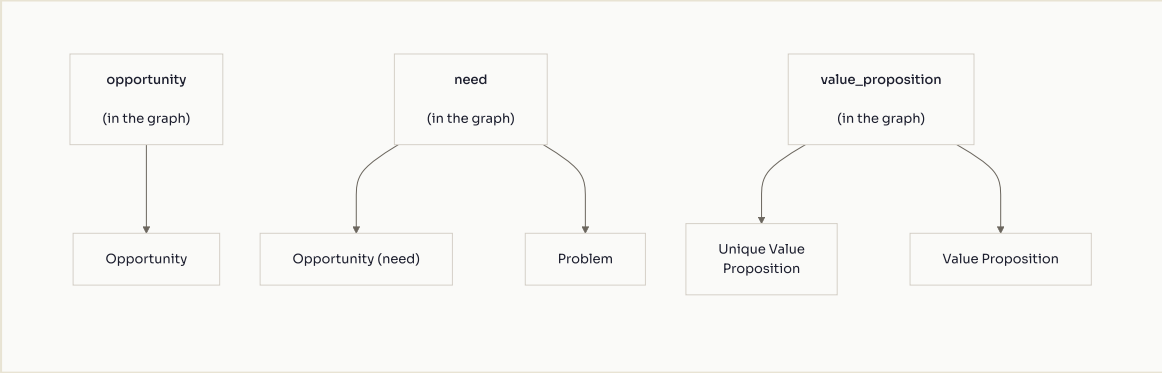
One response to these collisions is to give each framework its own schema and translate between them as needed. That is essentially the practice today, and it is why moving between tools costs so much human-mediated translation: a designer describing a *Frustration* and a product manager describing a *Pain Point* are often looking at the same thing without realising it.

UPG's approach is to treat frameworks as **view definitions** over a canonical graph. The underlying data is framework-agnostic: whatever the practitioner's framework calls it, the entity lands in the graph as a `need` with a declared valence (pain, gap, or constraint), or as a `feature` regardless of whether a Lean Canvas view calls it a *Solution*. The presentation is framework-fluent: the same `need` with `valence: pain` renders as *Pain Point* in a Lean Canvas view, as *Frustration* in an empathy-map view, and as *Friction* in a funnel view. A designer and a product manager applying different frameworks see vocabulary that fits their training, and the graph stays coherent underneath.

This is the move that lets a single UPG describe a product across all the disciplines that touch it. The graph is what a product *is*. The frameworks are how different people choose to read it. The two are not in tension; they are layered.

A concrete example makes the mechanism explicit. The Business Model Canvas (Osterwalder, 2010) defines nine building blocks including Customer Segments, Value Propositions, Channels, and Revenue Streams. In a traditional tool the canvas is an artefact, and the items on it exist inside the artefact. In UPG there is no BMC artefact. The entity types already exist in the graph (`value_proposition`, `revenue_stream`, `cost_structure`, `partnership`, `market_segment`). The BMC is a declarative definition, a JSON object whose `slots` select a subset of those types and arrange them in nine zones. The Lean Canvas is a different definition selecting a partially overlapping subset (including `need` displayed as *Problem*). The Opportunity Solution Tree is yet another, arranging `outcome`, `opportunity`, `solution`, `experiment`, and `assumption` as a tree rather than a grid. The same `value_proposition` node appears in every framework view that selects it; editing it in one view edits the underlying entity everywhere. Framework choice becomes a presentation-layer question rather than a data-modelling one, and a team that adopts Lean Canvas for two years and wants to move to Jobs-to-be-Done only needs to change their framework view; the data is untouched.

The mechanism is shown below. Three frameworks (BMC, Lean Canvas, and Opportunity Solution Tree) each select a different subset of UPG entity types and render them under framework-specific labels. The underlying entities are shared; only the view definition changes.



The same `value_proposition` node appears in the BMC view as *Value Proposition* and in the Lean Canvas as *Unique Value Proposition*. The same `need` node appears as *Problem* in Lean Canvas and *Opportunity (need)* in OST. Editing the entity in any view edits the single underlying record. Framework choice becomes a presentation-layer question rather than a data-modelling one.

Running a framework: the exercise model. A view definition says how to *read* the graph; many frameworks also *evaluate* it, and a RICE pass or a MoSCoW sort produces a result that has to live somewhere. UPG puts it on a relationship, not on the entity. Running a framework over a set of entities creates a `framework_exercise` node, one named pass (a Q3 prioritisation, a canvas for one segment), with a `framework_exercise_includes_node` edge to each entity it covers. The per-entity result rides that edge: a MoSCoW bucket, a RICE score, a Kano class, and the slot role the entity plays in the run. Because the value sits on the relationship, the entity stays what it is. A `feature` carries no scoring fields of its own, the same feature can be a *must* in one exercise and a *could* in another without collision, any entity type can be scored rather than only `feature`, and a re-run never overwrites the last one. Scoring is thus the inverse of presentation: a view reads many frameworks off one entity, while an exercise records one framework's reading of many entities, each on its own edge. The score is computed by the framework's `scoring_method` (Appendix H), evaluated at read time and never persisted to the node.

The full `UPGFramework` type schema, the `scoring_method` and exercise model, and the complete Business Model Canvas definition as worked example live in Appendix H.

---

3. The UPG Model

3.1 Design Goals

UPG was designed against six goals. The first four describe what the graph should be; the last two describe how it survives time and how it meets the world.

1. **Legible to humans and AI agents.** The same choices (self-explanatory names, verb-paired edges, inline property schemas) serve both audiences without translation. A practitioner encountering `job` for the first time needs no glossary; an agent producing a `hypothesis` from conversation knows the expected fields and follow-on relationships without being told.

2. **Framework-neutral and process-neutral.** No single framework is privileged in the data layer. Jobs-to-be-Done, Opportunity Solution Trees, Lean Canvas, and the Business Model Canvas are views over the same typed entities, never competing schemas (§2.5). The same neutrality applies to process: the graph does not require Dual-Track Agile, Lean Startup, or any specific methodology. The layered architecture (§3.2) keeps declarative structure in Layers 1–4 and procedural guidance in Layer 5, so teams can adopt the data model without committing to a playbook, or run playbooks without giving up portability.

3. **Portable** (§1.3). The `.upg` file is the unit of portability; it is JSON, git-friendly, and readable by any compatible tool.

4. **Incrementally adoptable.** A useful graph can begin with three entities. Only `id`, `type`, and `title` are required on any node; the full property schema is available but never demanded. Coverage grows as the team's thinking grows, not the other way around.

5. **Structurally durable across schema evolution.** Entity types carry immutable identifiers (`type_id`) that survive renames, merges, and deprecations. When `pain_point` was merged into `need` with a `valence: pain` property, the old `type_id` was preserved so existing `.upg` files auto-migrate on load without data loss. Names belong to humans; `type_id` values belong to the wire format. Lifecycle transitions, deprecations, and framework additions do not invalidate the graphs already in circulation.

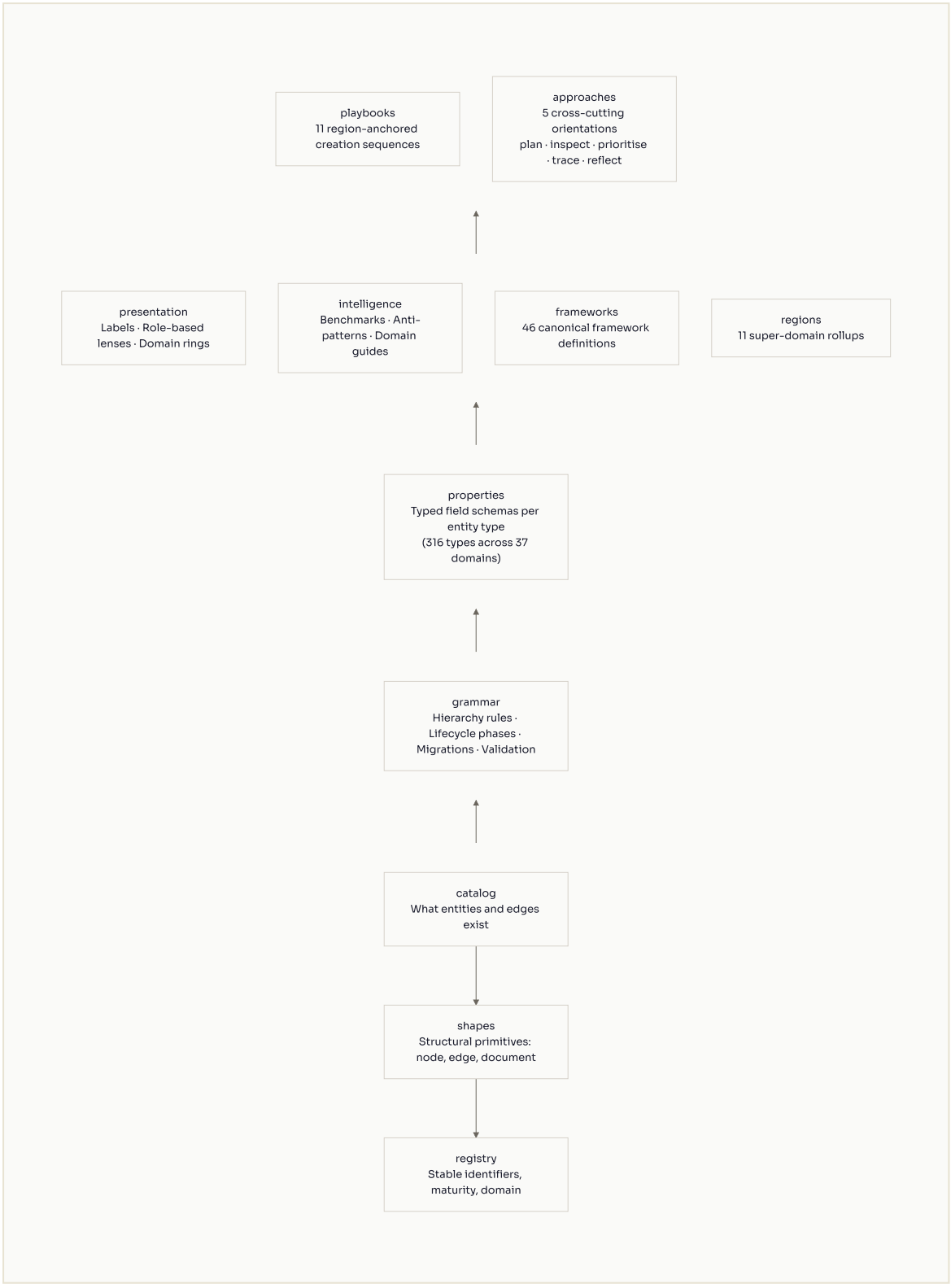
6. **Enactable through playbooks and approaches, not prescribed by them.** The declarative layers say what a graph *is*; Layer 5 (§3.7) says what an agent or practitioner can *do* with it. **Playbooks** provide region-anchored creation sequences: ordered guides for building out a domain of the graph. **Approaches** provide cross-cutting orientations: cognitive engagement modes that frame how to work with

whatever already exists. A team using UPG without any playbooks or approaches still has a valid, compounding graph. Playbooks and approaches are procedural scaffolding that reads and writes the same underlying structure; they do not alter it.

Every design decision later in this section can be traced back to one of these six. Where a decision serves more than one goal, the priority order above decides which wins on trade-off.

3.2 The Layered Architecture

The six goals from §3.1 become concrete first in how the specification is organised. UPG is split into **five layers**. Layers 1 through 4 are declarative: they describe what a graph *is*. Layer 5 is procedural: it describes what to *do* with one. Every layer depends only on the layers below it, never sideways and never upward. The result is a substrate whose foundation cannot be destabilised by changes at the surface.



Layer 1 (Foundation) names what exists and what shape it takes: the catalog of entity and edge types, the shape interfaces every node and edge implement, and the registry that assigns each type a stable identifier, a maturity, and a domain. Layer 2 (Grammar)

encodes what is allowed: hierarchy rules, lifecycle phases, migrations, and the validation routines tools import to check structural legality. Layer 3 (Properties) attaches typed field interfaces to each type (what a `persona` carries, what a `hypothesis` carries) and stays narrow on purpose so that framework-specific fields stay in Layer 4. Layer 4 (Output) holds the four read-time modules: presentation (labels, lenses, rings), intelligence (benchmarks, anti-patterns, domain guides), frameworks (RICE, Kano, BMC, OST as canvas view definitions over canonical types), and regions (the eleven super-domain rollups, §3.3). Layer 5 (Playbooks and Approaches) is the procedural layer: `UPGPlaybook` records carry region-anchored creation sequences, `UPGApproach` records carry cross-cutting cognitive orientations (Plan, Inspect, Prioritise, Trace, Reflect). §3.7 develops Layer 5 in detail; Appendix A walks every module.

The critical design decision. The `.upg` file contains Layers 1 through 3 only. Labels, lenses, benchmarks, frameworks, regions, playbooks, and approaches are never persisted to the file; they are applied at read time, recomputed at will, and replaceable without migration. The same graph data can be rendered through the Product lens or the Engineering lens, scored under RICE or Kano, grouped by ring or by region, and guided by a users-and-needs playbook or a discovery-validation playbook, all without touching what is stored. Storage stays narrow so the graph survives the evolution of everything above it.

Per-module detail, import graph, and representative benchmark, region, playbook, and approach records: Appendix A.

3.3 The Domain Model

The atomic domains named in the registry are the graph's subject headings: user, strategy, discovery, engineering, growth, and so on. UPG carries two groupings over that set, both at Layer 4. The **ring model** arranges the domains as concentric rings radiating outward from the product nucleus, with maturity over time as its axis. The **region model** rolls the domains up into eleven super-domains, with coherence across space as its axis. **When to reach for which:** rings answer *which stage is this product at?* (used by intelligence-module benchmarks that vary by lifecycle stage); regions answer *which area of work is this?* (used by playbooks, the `plan` and `inspect` approaches, and starter templates). Neither is persisted; both are computed at read time from the same atomic-domain assignments. A `competitor`, for instance, projects into Ring 1 (Understand) and Region 4 (Market & Competitive) from a single `market_intelligence` assignment, without either grouping being stored.

The ring model

Seven rings radiate outward from the product nucleus, each holding the domains that answer one question.

Ring	Name	Question
0	Nucleus	The seed
1	Understand	Who are we building for?
2	Define	What are we building?
3	Build	How do we construct it?
4	Grow	How do we make money?
5	Operate	How do we serve?
6	Extend	How do we scale?

Most products activate rings inside-out as they mature. A concept-stage product holds entities in Rings 0 through 2; a growth-stage product activates Rings 3 through 5; a mature organisation reaches into all seven. The intelligence module uses this progression to set healthy entity-count ranges per type per lifecycle stage and surfaces structural observations when the graph diverges from them.

The region model

Eleven regions group the atomic domains into super-domains, each a coherent area of product work.

#	Region	Anchor
1	Strategy & Outcomes	`objective`
2	Users & Needs	`persona`
3	Discovery, Research & Validation	`opportunity`
4	Market & Competitive	`competitor`
5	Experience, Design & Brand	`user_journey`
6	Product & Delivery	`feature`
7	Engineering & Platform	`service`
8	Business, GTM & Growth	`value_proposition`
9	Analytics & Data	`metric`
10	Operations & Quality	`incident`
11	Foundations	`specification`

Each region carries four defining pieces: a set of composed atomic domains, an **anchor entity** (the type that carries the region's design problem most sharply), a **shape archetype** (cascade, convergent hub, cyclic processing graph, DAG, event-driven collage, among others), and **boundary edges** fixing how the region connects to its neighbours.

The region is the primary unit of product work in UPG. When a product team asks *"what should we think about next?"*, the answer is almost always a region ("we need to think more about the business model" or "our discovery region is thin.") Regions are what playbooks (§3.7) bootstrap, what the `plan` and `inspect` approaches (§3.7) operate over, and what the intelligence module uses to scope its coverage analysis. The region is also the unit teams tend to specialise around: a designer primarily works in Region 5, an engineer in Region 7, a growth marketer in Region 8. Role-based lenses (§3.3 above) filter which regions dominate a given view, but the regions themselves are always present in the graph.

The anchor entity is not just a label. Each anchor is the entity type where its region's central question concentrates. `objective` anchors Strategy & Outcomes because the accountability question of strategy (*is this objective being met?*) resolves through an objective node. `persona` anchors Users & Needs because every cross-domain edge that describes a user (from competitive intelligence, from research, from feedback) eventually traces back to a persona. `feature` anchors Product & Delivery because the shipped unit of product value is a feature regardless of whether it came from a research-backed opportunity or a customer request.

Shape archetypes are meaningful. The shape describes the dominant topology of the region's internal edges, how entities within the region connect to each other. The shape tells an agent (or a practitioner) how to navigate the region: whether to start at the root and flow down, start at the hub and traverse outward, or start anywhere and follow the loop. Eight archetypes appear across the eleven regions: *cascade* (Strategy & Outcomes), *convergent hub* (Users & Needs), *cyclic processing graph* (Discovery), *layered DAG* (Product & Delivery), *layered mesh* (Engineering & Platform), *cyclic value-exchange* (Business, GTM & Growth), *event-driven collage* (Experience / Operations), and *polymorphic-target* (Foundations). Full per-region patterns with mermaid diagrams: Appendix J.

Boundary edges are the graph's joints. Every boundary edge is a registered cross-domain edge type in the catalog. The seam between Region 3 (Discovery) and Region 6 (Product & Delivery) is crossed by `learning_informs_feature` and `opportunity_addressed_by_feature`. The seam between Region 6 and Region 1 (Strategy) is crossed by `feature_drives_key_result`. The seam between Region 2 (Users) and Region 3 (Discovery) is crossed by `persona_has_need` and `job_surfaces_need`. Without these boundary edges, regions would be isolated silos. With them, the full product graph is one connected structure traversable from any starting point.

Full region records with entity rosters, shape archetypes, boundary-edge sets, and per-region profile notes: Appendix J.

3.4 Entities

Every entity in a UPG graph, whatever its type and whichever grouping it belongs to, carries the same structural base. Before the full schema, a concrete three-node example shows the essential idea: two entities connected by a typed, directed edge.

Each box is an entity: a typed, titled, structured record. Each arrow is an edge: a named, directed relationship. The two together form a traversable, queryable fragment of a product graph. The full schema that underpins this is:

Shape.

```
// UPG v0.2: packages/upg-spec/src/shapes/base-node.ts
interface UPGBaseNode {
  id: string                                // Unique identifier within the
graph
  type: UPGEntityType                       // Canonical UPG entity type
  title: string                             // Human-readable title
  description?: string                      // Optional narrative description
  tags?: string[]                           // Freeform tags for filtering
and grouping
  status?: string                           // Current lifecycle phase
  source_id?: string                        // Original ID in the source tool
  source_type?: string                      // Original type name in the
source tool
  mapping_confidence?: UPGMappingConfidence // 'high' | 'medium' | 'low' |
'manual'
  external_tool?: string                    // External tool holding the
canonical artifact
  external_ref?: string                     // URI to the canonical artifact
  external_id?: string                      // Identifier in the external
tool's system
  properties?: Record<string, unknown>      // Type-specific properties
}
```

Design decisions. Four choices are visible in this shape.

Source traceability. Every entity carries its origin: which tool it came from, what it was called there, where it lives externally, and how confident the type mapping is. An adapter importing a Linear issue records the Linear ID, the Linear type string, and a URL back to the issue, so round-trip export is lossless and the imported entity stays linked to its canonical artefact.

Mapping confidence is explicit, never silent. When a Notion page maps to UPG `persona`, the confidence level is recorded. `high` means unambiguous. `medium` means probable. `low` means speculative; human review is recommended. `manual` means a person made the call. Uncertainty is a first-class property of any imported entity.

Progressive structure. Only `id`, `type`, and `title` are required. A valid entity can be three fields. The full property schema for that type is available for deep modelling but never demanded. This supports a pattern observed in user sessions: people mention an entity in passing before they are ready to define it fully, and the format lets that first mention be captured immediately.

Wire-format stability. Every entity type carries an immutable identifier (e.g. `type_id: "ent_016"` for `persona`) that survives renames, merges, and deprecations. When `pain_point` was merged into `need` with a `valence: 'pain'` property, the `type_id` was preserved so existing `.upg` files auto-migrate on load without data loss. Names belong to humans; `type_id` values belong to the wire format. *Full* `EntityTypeMeta` *interface with maturity lifecycle and worked rename example: Appendix B.*

Implications. Taken together, these commitments let the specification evolve without breaking graphs in circulation. Imported entities stay linked to their origins, new fields attach through `properties` without touching the base, and any tool that can render one UPG entity can render any of them.

3.5 Edges

Entities alone describe a set. The graph emerges from the edges that connect them. Edges in UPG are typed, directed, and carry human-readable verbs in both directions; the edge instance is minimal, and the semantics live in a canonical catalog.

Shape.

```
// UPG v0.2: packages/upg-spec/src/shapes/edges.ts
interface UPGEdge {
  id: string                                  // Unique identifier within the
graph
  source: string                              // Source node ID
  target: string                              // Target node ID
  type: UPGEdgeType                           // Key into UPG_EDGE_CATALOG,
e.g. 'persona_pursues_job'
  mapping_confidence?: UPGMappingConfidence // If this edge type was inferred
during import
  properties?: Record<string, unknown>     // Edge-scoped payload, only on
opt-in edge types
}

// UPG v0.2: packages/upg-spec/src/catalog/edge-catalog.ts
interface UPGEdgeDefinition {
  forward_verb: string                        // "source [forward_verb] target"
  reverse_verb: string                        // "target [reverse_verb] source"
  classification: 'hierarchy' | 'causal' | 'semantic' | 'cross-domain'
  source_type: string                         // Entity type permitted at the
source
  target_type: string                         // Entity type permitted at the
target
  carries_properties?: boolean                // Opt-in: may this edge carry a
properties payload?
}
```

Every canonical edge type is registered once in `UPG_EDGE_CATALOG`. Three representative entries:

```
persona_pursues_job: {
  forward_verb: 'pursues',       reverse_verb: 'pursued_by',
  classification: 'semantic',
  source_type: 'persona',        target_type: 'job',
}

experiment_produces_learning: {
  forward_verb: 'produces',      reverse_verb: 'learned_from',
  classification: 'causal',
  source_type: 'experiment',     target_type: 'learning',
}

opportunity_addresses_need: {
  forward_verb: 'addresses',     reverse_verb: 'addressed_by',
  classification: 'cross-domain',
  source_type: 'opportunity',    target_type: 'need',
}
```

Design decisions. Four choices are visible in this shape.

Minimal instances, catalog-carried semantics. A plain edge instance on the wire is four fields plus an optional mapping confidence. Verb pair, classification, and endpoint types live in the catalog entry, not on the edge. This keeps graph files small and means a verb rename in the catalog does not require rewriting every edge that uses it. A deliberately small set of edge types opts into carrying a `properties` payload (the catalog entry sets `carries_properties: true`); the canonical case is `framework_exercise_includes_node`, which stores a framework's per-entity result on the relationship rather than on either endpoint, so the entities stay framework-agnostic (Appendix C, §2.5). The competitive `feature_rivals_competitor_feature` and the classification edges (`competitor_classified_as_classification_value` and its polymorphic sibling) likewise carry their assessment on the relationship; the classification edges additionally declare a typed `property_schema` (confidence on the `confidence_5` scale, plus provenance) that validators range-check. Validators reject `properties` on any edge type that has not opted in.

Every edge reads in both directions. *"Persona pursues job"* reads naturally forward; *"Job pursued by persona"* reads naturally backward. The graph can be traversed and rendered from any entity as origin without the query engine having to invert a verb. This matters when an AI agent is asked *"what is this job connected to?"* and needs to describe incoming as well as outgoing edges in natural language.

Every edge has a classification. Four classifications cover the catalog:

Classification	Purpose	Example
Hierarchy	parent-child containment	`feature_area → contains → feature`
Causal	cause-effect chains	`experiment → produces → learning`
Semantic	meaning associations	`persona → pursues → job`
Cross-domain	bridges across domains	`opportunity → addresses → need`

Classification drives traversal strategy. A traceability query (*"trace this feature back to user evidence"*) follows causal and cross-domain edges backward, never hierarchy edges. A scope query (*"what does this feature area contain?"*) follows hierarchy edges downward only. Without classification, every query degenerates into a full-graph walk.

Every edge pins its endpoints. `source_type` and `target_type` fix which entity types may appear at each end. An invalid endpoint pairing is a structural error at validation time, not a mystery at read time. *Full catalog excerpt across all four classifications: Appendix C.*

Implications. A small, declarative edge catalog lets queries know which edges to follow for which question, lets tools describe any edge in natural language without per-edge code, and lets the catalog evolve (verb renames, new edge types, deprecations) without rewriting graphs in circulation.

3.6 Traceability Through the Graph

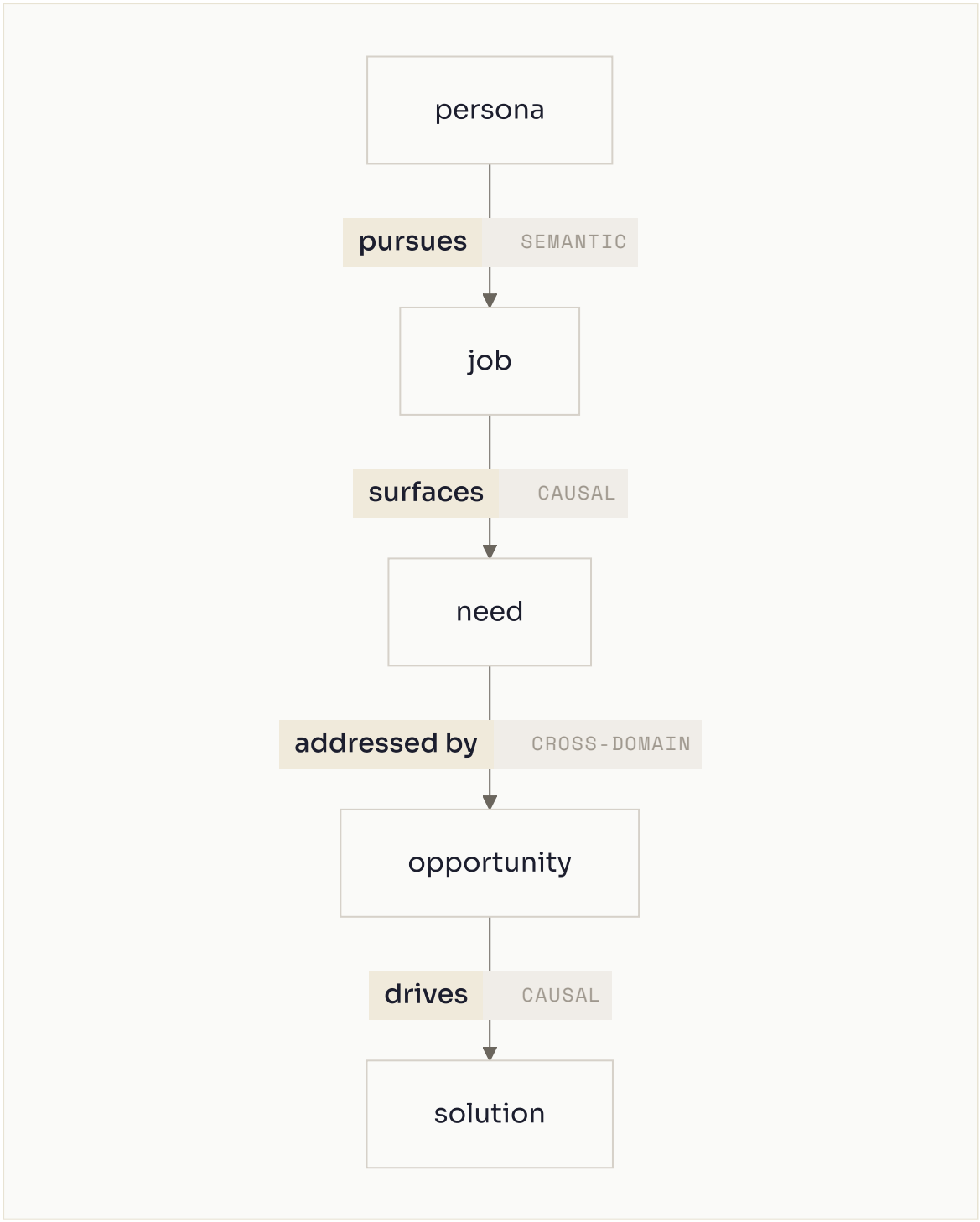
Classification (§3.5) is what makes the graph traversable. A **spine** is a path along typed edges that answers one question: where does a feature come from, what is a metric actually measuring, which user problem is a design decision meant to solve. UPG carries many such spines, and each uses whichever classifications its question needs.

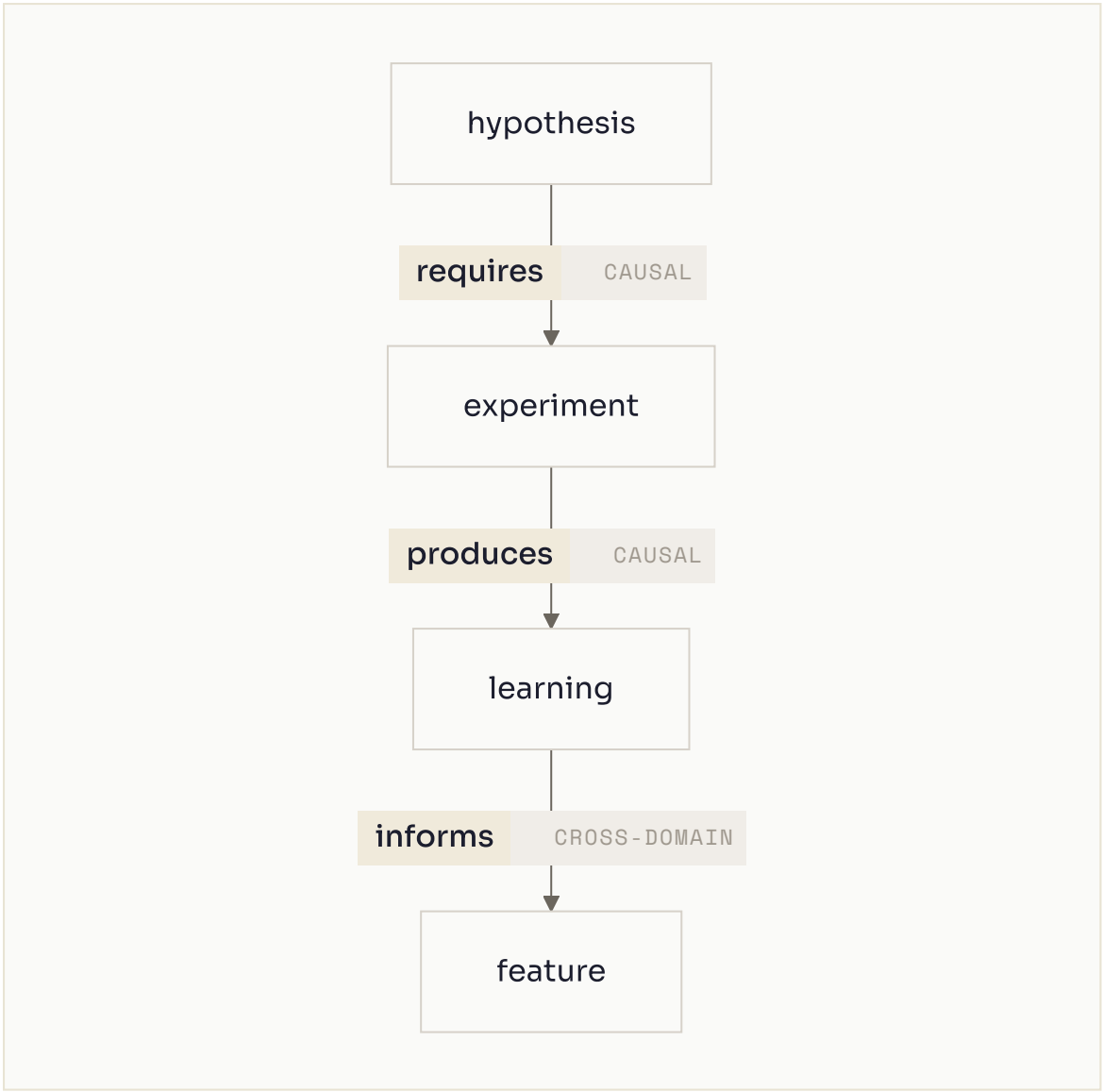
A small selection of spines visible in the v0.2 catalog:

- `competitor` ⟶ `competitor_feature` ⟶ `feature` traces competitive differentiation into shipped work.
- `metric` ⟶ `key_result` ⟶ `objective` ⟶ `strategic_theme` traces measurement back into strategy.
- `design_token` ⟶ `design_component` ⟶ `screen` ⟶ `user_journey` traces design primitives into user-visible flows.
- `persona` ⟶ `job` ⟶ `need` ⟶ `opportunity` ⟶ `solution` ⟶ `hypothesis` ⟶ `experiment` ⟶ `learning` ⟶ `feature` traces user needs into shipped features. This is the spine most often requested in user sessions, and the one walked through below.

What varies across spines is which edges each one follows and what question each one answers.

One spine, walked through. The discovery-to-delivery spine runs from persona to feature through discovery and experimentation. Every arrow is a typed edge in the catalog, with a classification (§3.5) and a verb pair:


persona
pursues
SEMANTIC
job
surfaces
CAUSAL
need
addressed by
CROSS-DOMAIN
opportunity
drives
CAUSAL
solution



The bridge is `solution → proposes → hypothesis` (left column ends, right column begins).

Read end to end:

> *Persona **pursues** job. Job **surfaces** need. Need **addressed by** opportunity. Opportunity **drives** solution. Solution **proposes** hypothesis. Hypothesis **requires** experiment. Experiment **produces** learning. Learning **informs** feature.*

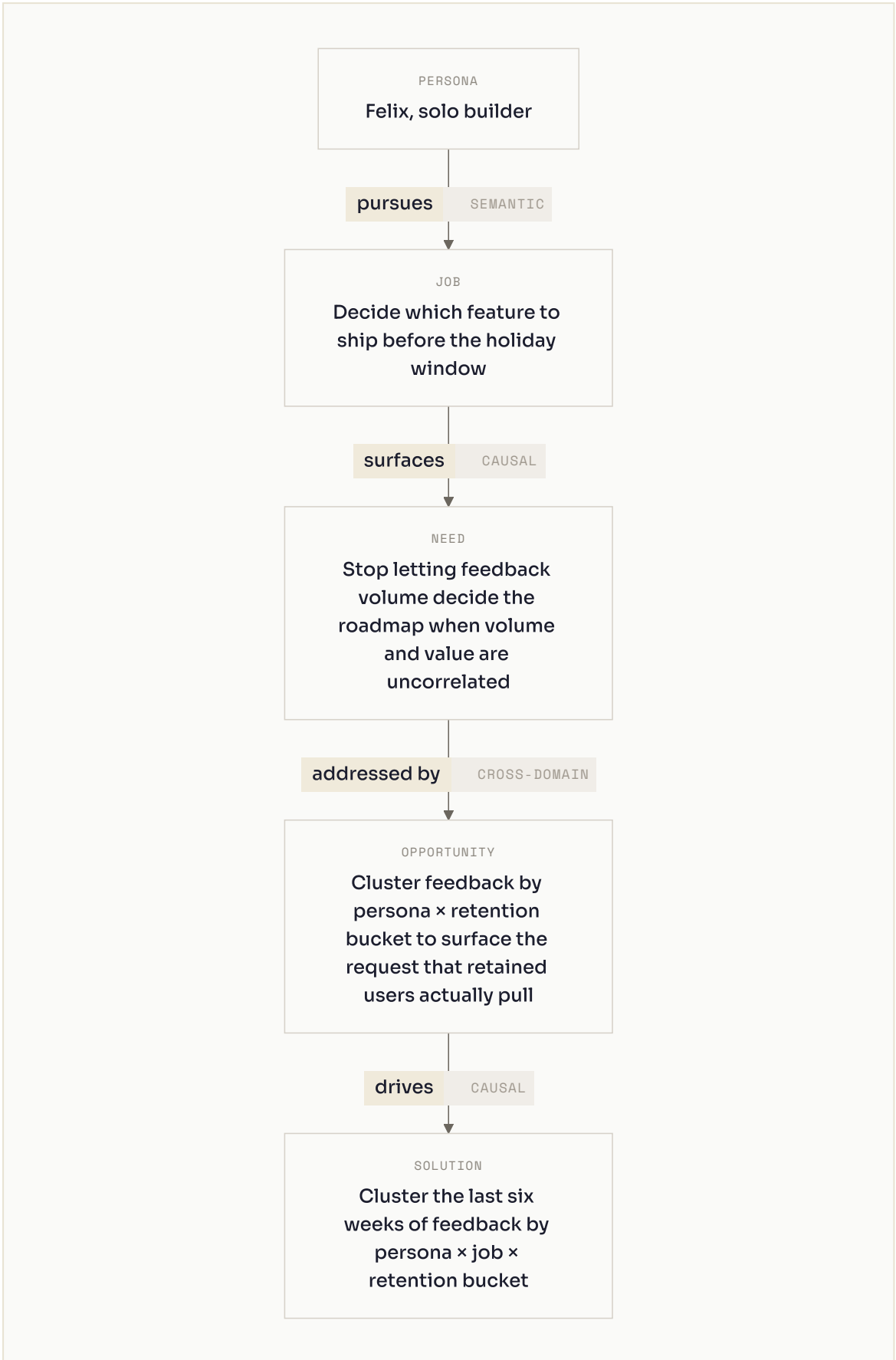
This spine walks causal and cross-domain edges only; hierarchy edges sit outside the traversal. The classification is what makes that distinction possible: the paths that carry *why* a feature exists look different in the catalog from the paths that carry *where* it lives.

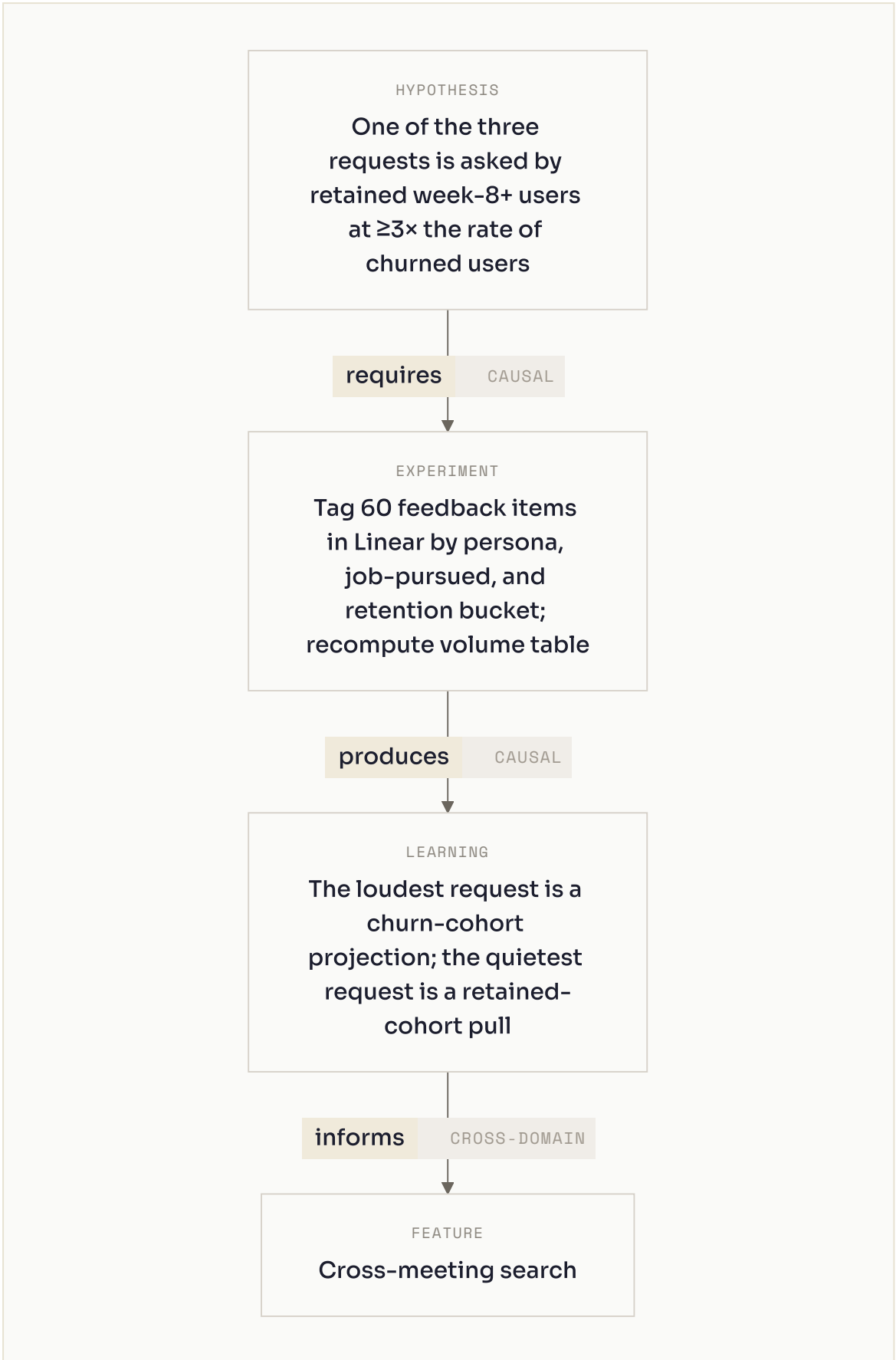
Provenance reads differently from organisation.

A team can capture the chain in whatever order they happen to think about it. Start from a feature idea, from a user interview, or from a market opportunity; the spine is reconstructed from whichever connections are present.

Worked example. Felix runs **Threadline**, a meeting-notes tool that auto-extracts action items and tags each one to the attendees on the calendar invite. Four months post-launch on Product Hunt; around 200 weekly actives, 30 paying at $9/mo (around $270 MRR); week-4 retention sits at 18%. Three feature requests have stacked up: **Slack integration** (25 votes, the loudest), **Calendar back-fill** (12 votes, medium), **Cross-meeting search** (8 votes, the quietest). Felix has bandwidth for one before the holiday window. The decision the graph has to support is not *which is most-requested*, it is *which one moves week-4 retention*.

The discovery chain Felix captures runs from the persona that owns the decision through the learning that resolves it:

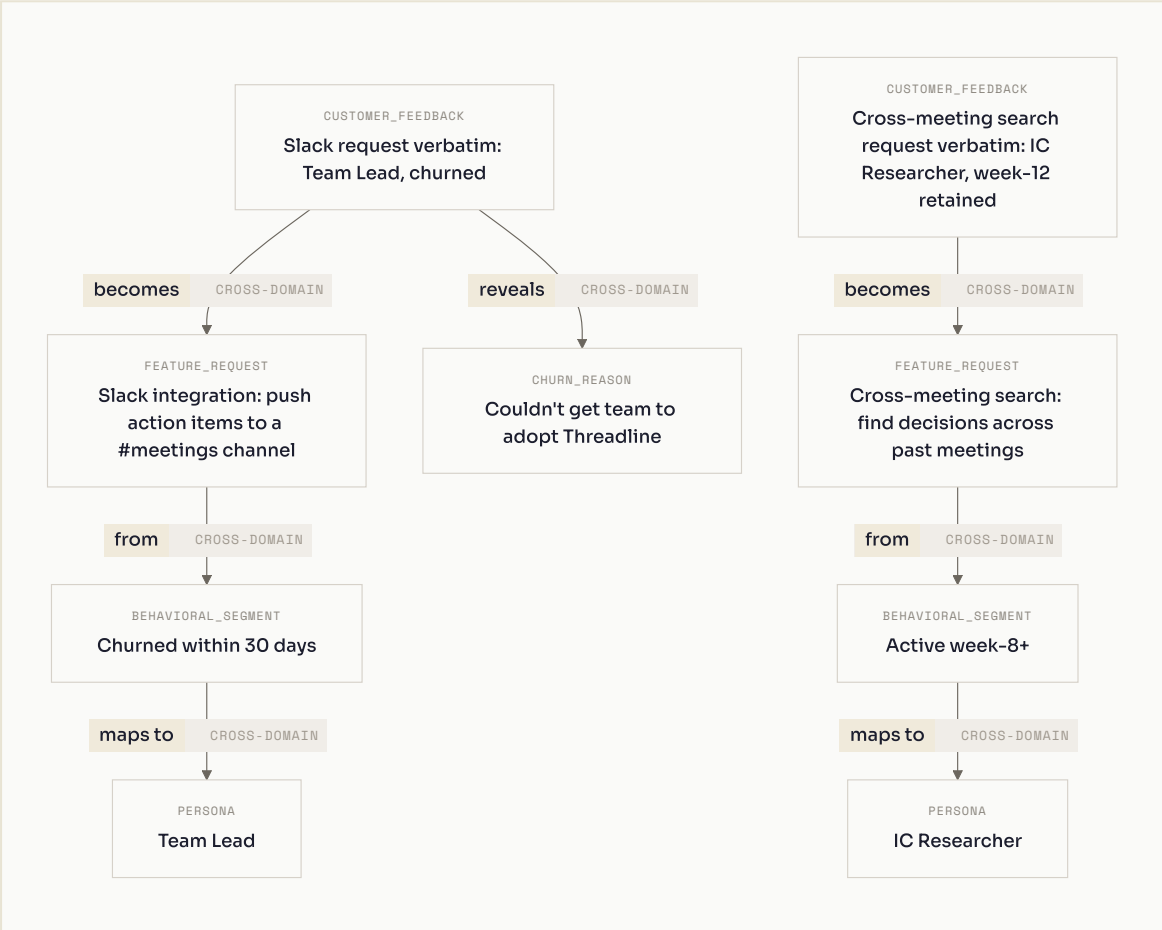

PERSONA
Felix, solo builder
pursues
SEMANTIC
JOB
Decide which feature to ship before the holiday window
surfaces
CAUSAL
NEED
Stop letting feedback volume decide the roadmap when volume and value are uncorrelated
addressed by
CROSS-DOMAIN
OPPORTUNITY
Cluster feedback by persona × retention bucket to surface the request that retained users actually pull
drives
CAUSAL
SOLUTION
Cluster the last six weeks of feedback by persona × job × retention bucket


HYPOTHESIS
One of the three requests is asked by retained week-8+ users at ≥3× the rate of churned users
requires
CAUSAL
EXPERIMENT
Tag 60 feedback items in Linear by persona, job-pursued, and retention bucket; recompute volume table
produces
CAUSAL
LEARNING
The loudest request is a churn-cohort projection; the quietest request is a retained-cohort pull
informs
CROSS-DOMAIN
FEATURE
Cross-meeting search

The left column captures what Felix observed and proposed; the right column captures what the experiment showed and what shipped. The bridge between them is the edge `solution ⟶ proposes ⟶ hypothesis`.

Every link in this chain is a graph edge. The shipped feature (`Cross-meeting search`) has a typed provenance reachable with a single query, rather than one scattered across Slack threads, support tickets, and a Linear backlog. Run the same query on a graph without this chain and it returns nothing: the feature exists, but the reason for it has to be reconstructed from prose.

A second spine, in the same graph. The discovery spine alone does not show why the loudest request was not the right one. That answer lives in a parallel chain in the Customer Feedback domain, which the same graph also carries:



This is the traversal Felix's experiment ran: `feature_request ⟶ from ⟶ behavioral_segment ⟶ maps_to ⟶ persona`. The Slack request resolves to the *Churned within 30 days* segment, which maps to the Team Lead persona; the same edge type takes the Cross-meeting search request to the *Active week-8+* segment and the IC Researcher persona. A parallel `customer_feedback ⟶ reveals ⟶ churn_reason` edge surfaces *why* the Slack signal is loud despite being a poor retention bet: it is a

churn-cohort projection of the missing piece, not a retained-cohort pull. The graph compounds knowledge across two domains (Discovery and Customer Feedback), and the feature decision is the join.

Read end to end, the two spines answer two questions a flat ticket queue cannot answer in one query: *what is the typed provenance of this shipped feature?*, and *what is the persona-and-retention shape of the cohort each request came from?* Volume alone says ship Slack. The graph says ship Cross-meeting search.

Many spines, one mechanism. The discovery-to-delivery spine is one walk through the catalog. The metric-to-strategy, competitor-to-feature, and design-token-to-journey spines work the same way: typed edges, classification-driven traversal, verbalisable at read time. When entities expected along a spine are missing or disconnected, the intelligence module (§3.8) reports it.

3.7 Playbooks and Approaches

Traceability (§3.6) describes how the graph is *read*. Layer 5 describes how it is *produced* and *worked with*, through two distinct primitives. A **playbook** runs in creation mode: it builds out a region of the graph. An **approach** runs in engagement mode: it frames how to work with what already exists. Playbooks compose ordered `Step` records; approaches are definition lookups that orient the LLM's engagement. The same graph is the subject of both; the modes are not interchangeable.

Playbooks

A playbook answers *"I'm populating this region of the graph: what should I create, and in what order?"* One canonical playbook exists per region; a region may also carry specialised playbooks for alternative entry paths. The catalog holds 13 playbooks across 11 regions (11 canonical, 2 specialised).

```
// UPG v0.3: packages/upg-spec/src/playbooks/types.ts
interface UPGPlaybook {
  id: string              // namespace-prefixed: 'playbook:<region>[-variant]'
  name: string            // Human-readable name
  version: string         // Semver
  description: string     // One-sentence description
  region: UPGRegionId     // REQUIRED: the region this playbook anchors
  is_canonical?: boolean // True for the one canonical playbook per region
  framework_id?: string  // Set on framework-anchored specialised playbooks
  target_anchor_entity?: string  // The entity this playbook produces
  creation_sequence: readonly Step[]  // Ordered creation steps
}
```

Design decisions.

Region-scoped, not domain-scoped. Playbooks anchor to a region (one of the eleven super-domains: Users & Needs, Discovery, Business & GTM, etc.) rather than an atomic domain. A region composes several atomic domains; the playbook's creation sequence spans them in the order that makes sense for the region's anchor entity. The Users & Needs playbook, for instance, walks personas, then jobs, then needs, then opportunities: four domains in one coherent sequence.

One canonical per region, W1 invariant. Every region has exactly one canonical playbook. This is an audited constraint, not a type-system guarantee: a CI script enforces it. Specialised playbooks (framework-anchored variants) exist alongside the canonical one; they do not replace it.

Steps are discriminated. Four step kinds compose the creation sequence: `domain_guide` defers to the Intelligence module's usage guide for a named domain; `framework` applies a named framework from the Frameworks module (Layer 4); `entity_sequence` fixes an ordered list of entity types to capture; `sub_sequence` nests another playbook, so longer journeys can reuse shorter ones without duplicating steps.

Structure shared, experience per surface. One playbook definition is authored once in `@unified-product-graph/core`; each surface (a terminal client, an IDE extension, a browser canvas) registers its own `PlaybookBinding`. The binding decides how steps render: it cannot change the sequence, the step kinds, or the prompts. The same playbook runs in a terminal and in a visual canvas without re-authoring.

Approaches

An approach is a cross-cutting cognitive orientation: the *path of arrival* to a region. The cartographic metaphor is precise: like a navigator's approach to a coastline, or an aircraft's final approach to a runway, an approach describes the heading and the angle of engagement with the terrain. It is the orientation that determines which aspects of the graph come into focus and how to move through them.

Five canonical approaches cover the full space of how a practitioner or AI agent can engage with a product graph. The catalog is closed at v0.8.0; adding a sixth would be a coordinated breaking change.

Approach	Question answered	What it surfaces
Plan	What should I build next?	Missing entities against canonical expectations; coverage gaps per region
Inspect	What's broken?	Anti-pattern violations, drift reports, lint passes with severity and fix hints
Prioritise	What's most important?	A ranked candidate set under an explicit framework (RICE, ICE, Kano, Cost of Delay)
Trace	Walk a path through what exists	A typed trail from an anchor entity along named entity types, using canonical edges
Reflect	What should I be questioning?	Structured prompts exposing assumptions, alternatives, blind spots, and load-bearing claims

```
// UPG v0.3, packages/upg-spec/src/approaches/types.ts
type UPGApproachId = 'plan' | 'inspect' | 'prioritise' | 'trace' | 'reflect'

interface UPGApproach {
  id: UPGApproachId           // bare verb, matches the MCP tool name
  label: string               // 'Plan', 'Inspect', etc.
  description: string         // Cartographic framing of the engagement
  question_answered: string   // The user intent in plain language
  signature_hint: string      // '(args) ⟶ return-shape' (v0.3.x execution
contract)
  framework_id_examples: string[]  // Named techniques that live inside this
approach
}
```

Design decisions.

Each approach carries named techniques. The `framework_id_examples` field lists the frameworks that live inside that approach. The five approaches are the engagement categories; the frameworks are the named techniques within each category. Five Whys, Pre-mortem, Red Team, Devil's Advocate, and Second-order Thinking are techniques inside `reflect`. RICE, ICE, Kano, and Cost of Delay are techniques inside `prioritise`. The approach names the path; the framework names the instrument.

Five approaches, five bare-verb MCP tools. Each approach is directly invocable via its id as an MCP tool: `plan`, `inspect`, `prioritise`, `trace`, `reflect`. At v0.8.0 these are definition lookups: the tool returns the approach record and invocation parameters, and the AI agent is the executor. Structured execution lands in a future release as the runtime matures.

3.8 The Intelligence Module

Traceability tells you what chains exist. Playbooks guide you in building them. The **intelligence module** (Layer 4) tells you whether what you have built is structurally healthy, and where it is not.

The intelligence module has three components.

Domain usage guides. One guide per atomic domain. Each guide is a declarative record specifying the canonical creation sequence for that domain: the entity types to create, the order to create them in, the properties most important to fill, and the conditions under which creation of one type should trigger creation of another. Domain usage guides are what playbooks defer to when a step has `kind: 'domain_guide'`: the guide is the intelligence behind the playbook step. An agent calling `get_domain_guide({ domain: 'discovery' })` gets a structured record it can use to drive a creation session without relying on in-context instructions.

Benchmarks. The intelligence module carries three kinds of declarative benchmark, each evaluated at the relevant product lifecycle stage:

Kind	What it checks	Example
Count	Healthy range (`min`, `max`) per entity type	A validation-stage product should have 3–12 hypotheses
Ratio	Healthy ratio between two entity types	Features-to-hypotheses should not exceed 10:1; above that, work runs ahead of validation
Relationship	Structural pattern that should be present	Every shipped feature should connect to at least one learning

Benchmarks are what `get_graph_digest` uses to populate `benchmark_hits`: the digest is the spec's self-measurement at read time, grounded in the ontology's own expectations rather than in an external heuristic.

Anti-patterns. The intelligence module carries a catalog of named anti-patterns, each with a machine-checkable `structured_condition` (detection is data, not code), a `severity` (`high` or `medium`), the lifecycle `stages` it applies to, a `why_it_matters` rationale, a `remediation` hint, and a `source` attribution. They detect broken or missing narrative chains, the failure mode the spec exists to prevent. Anti-patterns are what the `inspect` approach surfaces when it audits a graph. Three representative entries:

Anti-pattern	Severity	What it catches
Features without hypotheses	high	Features exist but no hypotheses connect to them: work runs ahead of validation
Untested-hypothesis pile-up	medium	Hypotheses accumulate with no experiments to test them
Personas without jobs	high	Personas exist but none link into the user chain via a job, need, or desired outcome

`validate_graph` runs all anti-pattern checks across the whole graph and returns a structured violation list.

The intelligence module is what makes the difference between a graph that is a passive record and a graph that actively guides its own evolution. A practitioner can ask *"what should I do next?"* and receive a structured answer grounded in the ontology's canonical expectations: not a recommendation from a language model, but a comparison of what exists in the graph against what the spec says should exist.

Full benchmark definitions and anti-pattern catalog: Appendix E (`list_benchmarks`, `list_anti_patterns`, `validate_graph` tool descriptions).

3.9 Lenses and the Presentation Layer

The same graph can be read from multiple perspectives without changing what is stored. A **lens** is a named projection over the entity catalog: it selects a subset of domains, relabels entity types with role-appropriate vocabulary, and orders the regions that matter most to that role.

Lens	Centred on	Foregrounds
`product`	PM	Users, discovery, features, outcomes
`engineering`	Engineer	Services, APIs, data, deployment, testing
`design`	Designer	Journeys, screens, components, design systems
`growth`	Growth	Funnels, channels, cohorts, experiments, metrics

The four above are illustrative; the canonical set runs to nine — `product`, `design`, `engineering`, `growth`, `business`, `research`, `marketing`, `competitive`, and `full`. The `competitive` lens, added in 0.10.0, is the narrowest of all: it foregrounds the single `market_intelligence` domain, so a product marketer sees only rivals, their offerings, the dated moves they make (§3.11), and where the product leads or trails.

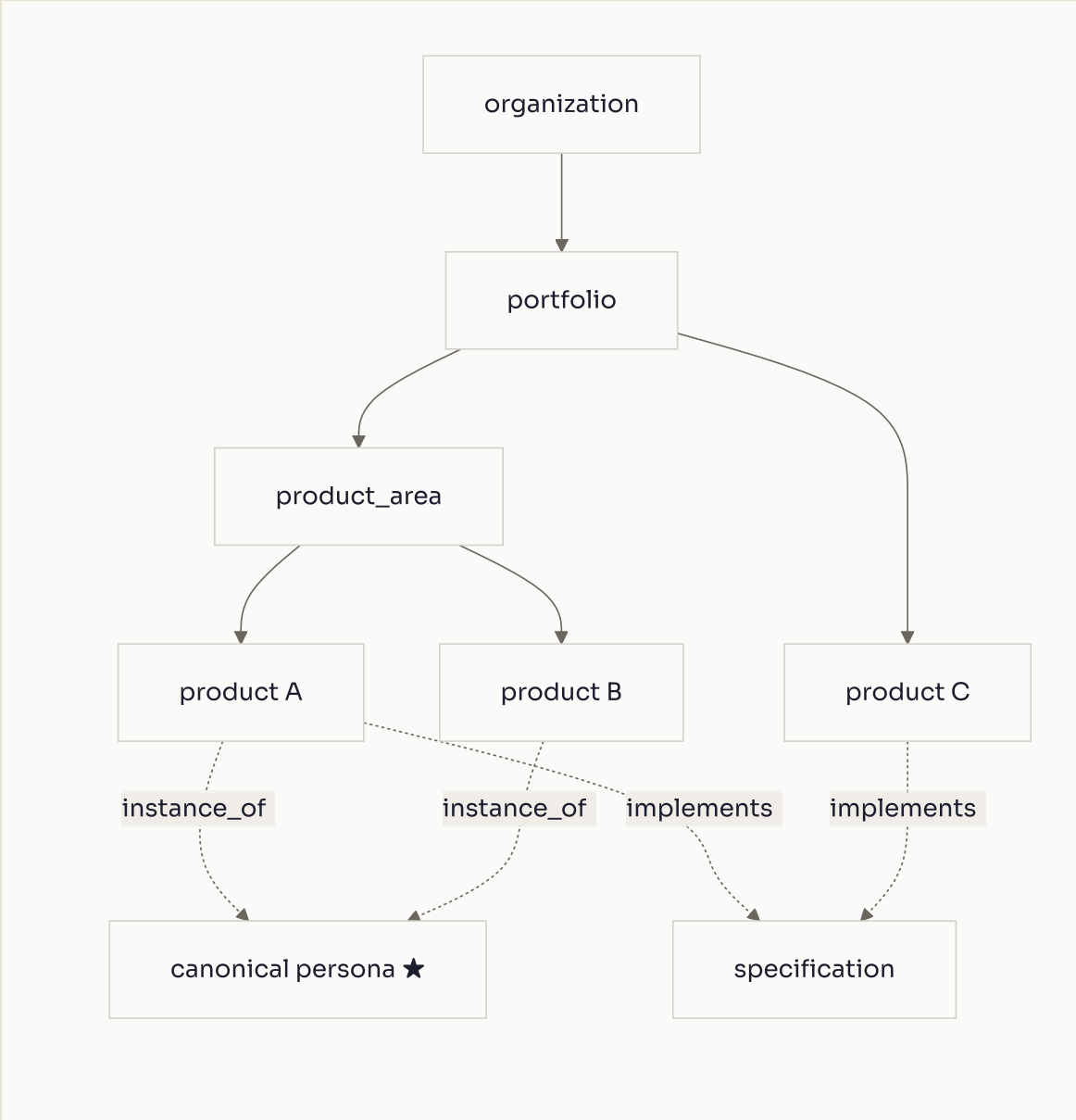
Lenses are what `get_product_context` uses to scope its digest and what `update_session_context` stores as the active lens for an ongoing session. The other presentation-layer outputs, framework labels (§2.5, Appendix M) and the domain ring visualisation (§3.3), sit alongside lenses at the same Layer 4 surface.

3.10 The Portfolio Tier: Multi-Product Graphs

A single `.upg` file holds one product, and §3.1 through §3.9 described how knowledge compounds inside it. But few products exist alone. A company runs several at once; a platform hosts the products built on top of it; a company-level north-star metric is fed by the KPIs of every product beneath it; the same persona buys one product and uses another. The knowledge that compounds *within* a product should also compound *across* a portfolio of them. UPG extends the same layered model to that case rather than inventing a second one.

Three tiers, one ontology. A portfolio is organised into three conceptual tiers, and only the middle tier introduces new vocabulary.

1. **Product graphs** are the instances. Each is an ordinary single-product graph, exactly as described above, stored in its own file and valid on its own. Nothing about a product graph changes when it joins a portfolio.
2. **The portfolio tier** is the organisational structure that holds them. An `organization` invests via one or more `portfolio` nodes, which contain `product` nodes, optionally grouped into nestable `product_area` nodes. These Nucleus-ring types and their hierarchy edges (`organization_invests_via_portfolio`, `portfolio_contains_product`, `product_area_contains_product`, and the nesting variants) are the only structure the tier adds. It answers *what do we have, and how is it grouped?*
3. **The registry** is the shared-vocabulary tier: a portfolio-level home for the entities many products refer to. A canonical persona, a company metric, a shared competitor, a foundational specification: defined once, and pointed at from every product that instantiates it. It answers *what do these products have in common?*



Cross-product edges are the seams. Within a product, typed edges connect typed nodes. Across products, a separate closed union of twenty-four **cross-product edge types** connects nodes that live in different files. Each references its endpoints by qualified id (`{product_id}/{node_id}`, or `registry/{node_id}` for a registry target), reusing the cross-file reference mechanism the file format already carries. They cluster into a few families, each answering a question a single product graph cannot:

Family	Edge types	What it expresses
Peer equivalence	`shares_persona`, `shares_competitor`, `shares_metric`	two products' nodes denote the same thing (symmetric)
Product relationships	`depends_on_product`, `hosts`, `succeeds`, `cannibalises`	runtime dependency, hosting, succession, or overlap between whole products
Strategic rollup	`contributes_to`, `rolls_up_to`	a product objective or KPI feeds a company objective or north-star (the OKR and measurement cascades, across products)
Canonical instance	`instance_of`	a product node is an instance of a canonical registry node (same type)
Area to audience	`area_serves_persona`, `area_targets_market_segment`	a product area serves a canonical persona or targets a segment, with primary/secondary relevance
Foundations	`product_implements_specification`, `product_exposes_specification`, `feature_conforms_to_specification`, `api_contract_speaks_specification`, `product_exposes_primitive`, `feature_manipulates_primitive`, `primitive_stored_as_data_type`	a product, feature, or API contract implements, exposes, or conforms to a shared specification or primitive
Competitive intelligence	`feature_rivals_competitor_feature`, `competitor_signal_maps_to_feature`, `competitor_signal_surfaces_opportunity`, `competitor_classified_as_classification_value`, `node_classified_as_classification_value`	one of our features rivals a competitor's (parity carried on the edge), a dated competitor move maps onto a feature or surfaces an opportunity, and any node (a competitor, or generically) is classified against a canonical registry classification value — spanning our product graph and a watched competitor-

intelligence graph (§3.11)

The registry turns convention into structure. Without a registry, a shared persona is copied into every product graph, and keeping the copies aligned is a convention enforced by discipline rather than by the spec. The registry makes it an architecture. The canonical persona is a normal `persona` node that happens to live in the registry tier, and each product's local persona links to it with an `instance_of` edge. Canonicalness is conferred by *where the node lives*, not by a new type or a flag, so the same mechanism serves personas, metrics, competitors, and market segments without per-type machinery. The payoff is **rollup** (*every instance of the Developer persona, and the per-product jobs each one pursues*), **diff**, and **drift detection**: a `portfolio_validate` pass flags an instance whose title or shape has strayed from its canonical, while distinguishing genuine drift from a *sanctioned* product-local alias carried on the edge itself. `instance_of` and the symmetric `shares_*` edges coexist; they answer different questions (canonical-to-instance versus peer-to-peer), and neither deprecates the other.

Foundations: the deepest shared layer. The most invisible cross-product structure is the foundational technology many products are built on: a query language, a content format, a wire protocol. UPG models these as two Foundations-region entity types. A `specification` is the governed rulebook a product *implements*, *exposes*, or *conforms to*; a `primitive` is a compositional unit a specification *defines*. Registered once and referenced by every product that uses them, they make the deepest layer of a portfolio's moat visible as first-class objects instead of a string repeated across a dozen graphs. Foundations is the eleventh canonical region (§3.3, Appendix J), anchored on `specification`, with a polymorphic-target shape: many products point inward at the same small set of shared specifications.

Portfolio-scoped reads. The single-product read primitives have multi-product counterparts that fan out across every product in scope and roll the results up in one call: `portfolio_query` (the multi-product `query`), `portfolio_digest` (the multi-product `get_graph_digest`), and `portfolio_validate` (the multi-product `validate_graph`, and the home of registry drift detection). A portfolio can therefore answer questions no single product graph holds: *which company objective has no product driving it? which products share the persona we are about to reposition? which products implement the specification we are about to version?* Each is a typed traversal of one connected structure rather than a manual reconciliation across files.

A single graph makes a product's knowledge compound across sessions. The portfolio tier makes it compound across products: a persona learned once is available everywhere, a metric defined once rolls up automatically, and a foundational primitive

named once is visible to every product built on it. The mechanics of the portfolio document that serialises all of this are in §4.7; its full shape and a worked example are in Appendix O.

3.11 Competitive Intelligence: Watched Graphs and Parity

The portfolio tier so far has modelled what an organisation *owns*. A product creator also has to watch what it does not own. UPG 0.10.0 extends the portfolio tier to the competitive landscape with one new entity type, a parity edge family, a role lens, and a member kind that tells the tooling not to judge a rival by the standards of a product under management.

A member kind for graphs you do not own. A portfolio member now carries a `member_kind`: `product` (a shippable product under management), `org_rollup` (the company umbrella graph — org-level vision and OKRs, not a shippable thing), or `watched` (an externally monitored graph, such as a competitor's). The distinction is operational, not cosmetic: a `watched` competitor-intelligence graph must *not* be scored against product-management expectations or drag portfolio health — an empty discovery region in a rival's graph is not a gap in *our* work. `portfolio_validate` and the coverage scorers read `member_kind` and scope themselves accordingly, so a watched graph can sit inside the same portfolio as the products it competes with without distorting their health.

A signal is a dated move. The new entity type is `competitor_signal` (Region 4, Market & Competitive): a single dated competitor action — a feature launch, a pricing change, an acquisition, a partnership, a market entry — emitted by a `competitor` (`competitor_emits_competitor_signal`) inside the watched graph. Two cross-edges carry the signal back into our own product graph: `competitor_signal_maps_to_feature` (the move lands on a feature we already ship) and `competitor_signal_surfaces_opportunity` (the move reveals a gap worth pursuing). Because they cross from a watched graph into ours, they are dual-registered — a within-graph catalog edge for the degenerate single-graph case, and a cross-product edge for the portfolio case — the same dual-registration the foundations edges use.

Parity as a traversal, not a scan. The parity edge `feature_rivals_competitor_feature` connects one of our features to a competitor's, and carries the assessment *on the edge*: `parity_status`, relative quality, whether it is a gap, when it was assessed, the supporting evidence, and a confidence score. This is the point of modelling it as a typed edge rather than a free-text field — *"where are we behind, and by which area?"* becomes a single traversal over the competitive lens (§3.9), not a manual scan over prose. The competitive surface also exercises a property-type addition from the same release: properties may now be typed `object[]`, letting a `competitive_analysis` carry structured `commitments` and `capabilities` lists, and every competitive record carries a

small **provenance mixin** — `source`, `last_updated`, `observed_by`, and a `confidence` assessment — so a stale, machine-polled signal can be told apart from a fresh, hand-verified one. When `observed_by` is absent, a human, not a poller, was the last writer.

Classification: positioning rivals on the axes that matter. Parity answers *where are we behind on this feature*; classification answers *what kind of competitor is this*. A classification axis is a canonical registry entity (`classification_axis`) that owns a set of `classification_value` children — *AI maturity* with values *agentic, integrated, bolt-on*; *go-to-market* with values *PLG, sales-led, hybrid*. Any node can then be positioned on an axis with a classify edge to one of those values: `competitor_classified_as_classification_value` for the common case of placing a rival, and the polymorphic `node_classified_as_classification_value` for everything else. Because the axes and their values live in the shared registry, every product and every watched graph in the portfolio is positioned against *the same* vocabulary — the competitive map is comparable across the whole organisation rather than re-invented per analyst.

Edge-carried properties become a typed, validated schema. The parity edge already carried its assessment on the edge; 0.10.4 generalised that into a first-class mechanism. An edge type may now declare a `property_schema` — the set of properties it is allowed to carry, each with its own type (an enum, an assessment on a named scale, a date, a provenance mixin) — and edge-property writes are validated against it exactly as node properties are validated against an entity shape (P17). A classify edge carries a `confidence` assessment on the `confidence_5` scale, an `assessed_on` date, a free-text `rationale`, and `evidence`; an unknown key or an off-scale value is rejected at the write boundary rather than surfacing later as silent drift. The assessment is structured data, not a sentence in a notes field, so *"show me every competitor we placed as agentic with at least medium confidence"* is a filter, not a re-read.

The classification is readable as a traversal and a distribution. A classify edge is a cross-product edge — it crosses from a product (or watched) graph into the registry — so the read path has to resolve the registry target rather than stop at a qualified id it cannot follow. `portfolio_query` follows a named classify edge out to its registry value and returns that value as a resolved terminal node, so *what is each rival classified as* is one traversal instead of a manual join; before this the same query reported zero edges, because the target lived in a document the per-product reader never opened. `portfolio_digest` carries a **classification distribution** alongside the structural counts: per axis, how many members fall on each value — the shape of the competitive field at a glance (*four rivals agentic, two integrated, one bolt-on*). Re-classifying is idempotent at the property level: writing the same classify edge again with a fuller assessment **upserts** the properties onto the existing edge rather than creating a duplicate or silently discarding them, so a confidence or evidence backfill across a large competitive set lands cleanly. The whole capability — classify, carry validated properties, query,

distribute — is exposed identically on the MCP tool surface and the `upg` command-line interface (`upg portfolio classify`, and `--properties` on `connect`), because a product creator working a competitive analysis in a terminal should reach the same graph an agent does.

---

4. Architecture and Implementation

§3 described the five layers of the Unified Product Graph (UPG) specification. §4 describes what each of those layers looks like on disk and in flight: the file format that serialises Layers 1 through 3, the MCP server that reads and writes it, the import adapters that bring existing corpora in, the playbook and approach runtime that implements Layer 5, and the hybrid markdown surface that lets prose and graph coexist in one file.

4.1 The `.upg` File Format

The file format has to be readable by hand, diffable in version control, and producible by a language model without a framework to consult. A single JSON document satisfies all three.

A `.upg` file is a JSON document (MIME type `application/vnd.upg+json`). A minimal but complete example, a one-product graph with a persona, a job, a need, and two typed edges connecting them, drawn from the Threadline reference graph:

```
{
  "$upg": {
    "format_version": "1.0.0",
    "spec_version": "0.8.10",
    "product": {
      "id": "n_5KO9z8qsX-pYKVIb",
      "title": "Threadline",
      "stage": "growth"
    },
    "counts": {
      "nodes": 3,
      "edges": 2
    },
    "provenance": {
      "tool": "upg-mcp-server",
      "tool_version": "0.8.10",
      "exported_at": "2026-04-23T12:00:00Z"
    },
    "integrity": {
      "algorithm": "sha256-128",
      "body": "098dbda4962d49fa9ba4d1206f3f69ac"
    }
  },
  "product": {
    "id": "n_5KO9z8qsX-pYKVIb",
    "title": "Threadline",
    "stage": "growth"
  },
  "nodes": [
    {
      "id": "n_XqLDdZOoqrmufgjd",
      "type": "job",
      "title": "Decide which feature to ship before the holiday window",
      "properties": {
        "importance": { "label": "Critical", "value": 5 },
        "job_type": "functional",
        "statement": "Felix has bandwidth for one feature before the December
slowdown. Three requests are stacked. The job is to pick the one that most
moves the metric that matters: week-4 retention, not the loudest one."
      }
    },
    {
      "id": "n_8B1SNCNbDSs4O8Qr",
      "type": "need",
      "title": "Stop letting feedback volume decide the roadmap when volume
and value are uncorrelated",
      "status": "raw",
      "properties": {
        "severity": { "label": "Severe", "value": 4 },
```

```
    },    {      "id": "n_SuIk0TASeSWJRFaf",      "type": "persona",
"title": "Felix, solo builder",      "description": "Builds Threadline
evenings on top of a day job. Ships every two weeks. Four months post-launch.
Faces a recurring 'which request next?' decision and has shipped two of the
last four features chasing loudest-voice asks, both shipped to crickets.",
"properties": {        "experience_level": "intermediate",
"is_primary": true      }    }  ],  "edges": [    {      "id": "edg_1",
"source": "n_SuIk0TASeSWJRFaf",      "target": "n_XqLDdZOoqrmufgjd",
"type": "persona_pursues_job",      "mapping_confidence": "high"    },    {
"id": "edg_2",      "source": "n_XqLDdZOoqrmufgjd",      "target":
"n_8B1SNCNbDSs4O8Qr",      "type": "job_surfaces_need",
"mapping_confidence": "high"    }  ]}
```


A single leading `$upg` header carries everything a reader needs before it walks the graph: the serialisation `format_version`, the `spec_version` the data conforms to, a one-line product summary, entity and edge counts, write provenance, and an integrity checksum over the body. Below it, `product`, `nodes`, and `edges` hold the graph itself. Three nodes, two edges, enough structure to answer *"which needs does this persona's job surface?"* with a typed query. Every property carries its schema: `valence` is an enum, `severity` is an assessment with a qualitative label and a numeric value (P17). Every edge carries a typed relationship with a forward and reverse verb inherited from the catalog. The persona's `is_primary: true` flag establishes which persona is canonical for the product without requiring a prose convention. Node IDs are server-issued opaque identifiers that survive across sessions and tools (P3, P14). The nodes appear ordered by type then title, and object keys are sorted, because the file is written in *canonical form* (below): the same graph always serialises to the same bytes.

Five format decisions follow from the file needing to survive version control, tool changes, and schema evolution.

1. **Git-friendly JSON.** JSON, not a custom format or a binary encoding, so the file opens in any editor, reads into any language, and diffs in git. A product decision three months ago is a `git blame` away.
2. **Canonical form.** The same logical graph always serialises to byte-identical output, regardless of which tool wrote it. Object keys follow a fixed order, the node and edge collections are sorted by stable keys, encoding is pinned (UTF-8, two-space indent, LF), and volatile fields are kept out of the hashed body. The rules adapt RFC 8785 (the JSON Canonicalization Scheme) with two deliberate departures for human review: the output is pretty-printed rather than minified, and the set-like collections are sorted rather than left in write order. The payoff is that adding one edge produces a one-line diff instead of a reshuffled file, and a no-op re-save produces no diff at all. A reference formatter, `upg fmt`, rewrites any file to canonical form and `upg fmt --check` gates it in CI, exactly as `gofmt` or Prettier do for code. Appendix N gives the full specification.

3. **A reserved `$upg` header and an explicit `format_version`.** Metadata that used to sit as loose siblings of the data (version, provenance, integrity) is consolidated into one leading `$upg` object, so a reviewer reads *what product, what version, how big, who last wrote it, and a content checksum* before scrolling a single node. The on-disk `format_version` is distinct from `spec_version`: the *serialisation* can evolve independently of the *ontology*. Portfolio files use the same header with `kind: "portfolio"`.
4. **Integrity and migration on load.** `$upg.integrity.body` is a SHA-256 checksum of the canonical body, truncated to 128 bits (labelled `sha256-128`), so it is stable across no-op re-exports and changes only when the graph changes. On load, tools verify it and, in the same step, auto-migrate deprecated types to their canonical replacements (for example, `pain_point` → `need`, Appendix G) and repair legacy serialisation drift such as double-encoded property values. Downstream tools only ever see current, well-formed types; corruption and structural drift are caught at the file boundary rather than deep in a traversal. A published JSON Schema (Draft 2020-12) lets any editor or CI step validate the structure independently.
5. **Unknown-type preservation, and reads accept both envelopes.** If a writer produces a node with a type not in the reader's spec version, the reader preserves the node unchanged rather than dropping it; a newer writer cannot break an older reader. In the same spirit, readers accept both the `$upg` envelope and the older flat layout, so adoption need not be instantaneous across a fleet of tools.

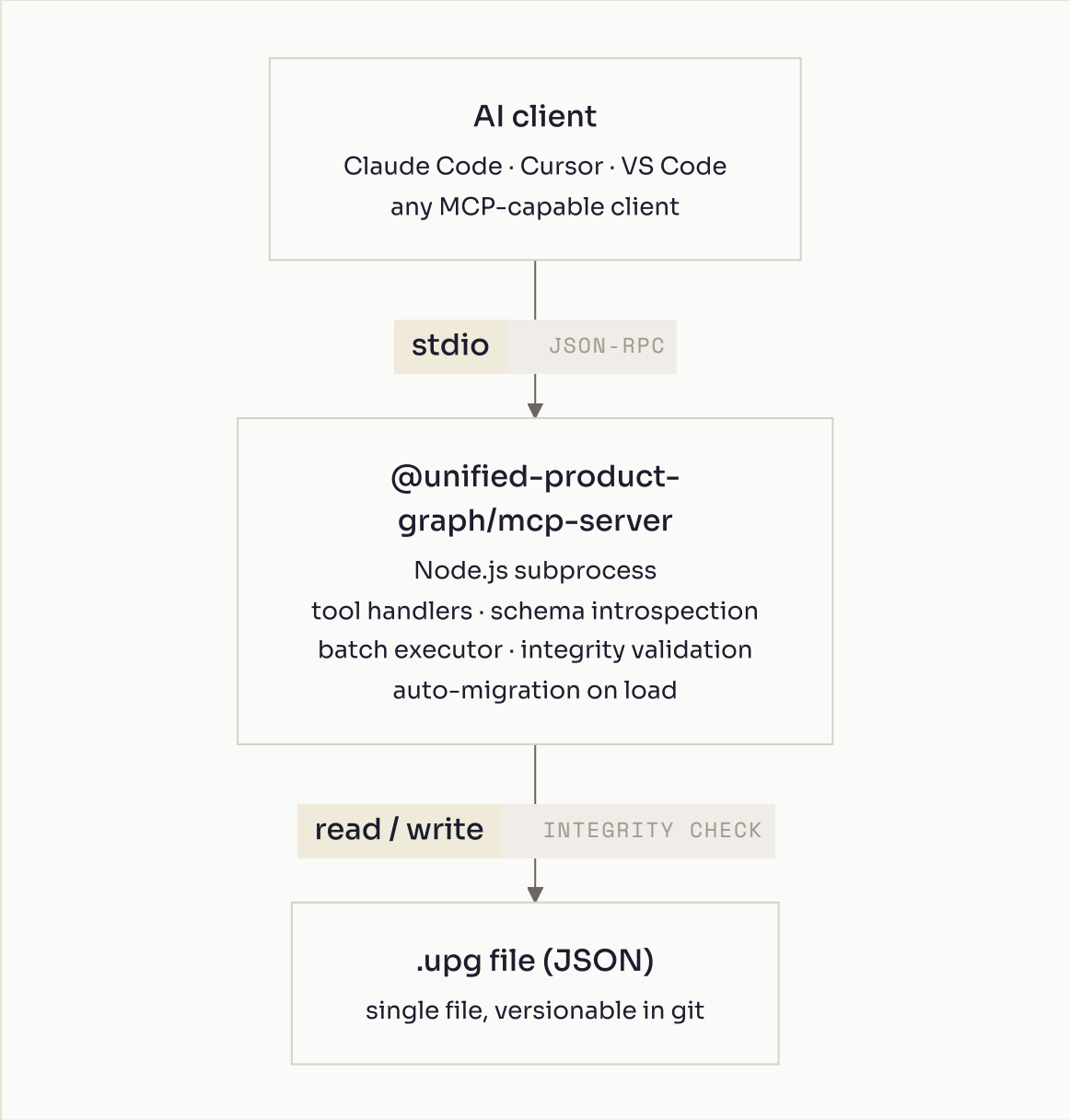
One product per file. The default unit is one product, keeping files scoped and shareable without leaking unrelated work. A portfolio document extends the same `$upg`-headed shape to multiple products with cross-product edges.

Serialisation scope. `.upg` carries Layers 1 through 3 (catalog, shapes, registry, grammar, properties). Layers 4 and 5 (presentation, intelligence, frameworks, regions, playbooks, approaches) are read-time concerns and are never written to the file. The same graph data can be rendered through any output module or guided by any playbook without round-tripping through the file format.

4.2 The MCP Server

The file is the state. Every client (human editor, AI agent, or in-house tool) reaches into it through the same entry point: the UPG MCP server (`@unified-product-graph/mcp-server`). §2.4 introduced the Model Context Protocol; the reference server is its concrete implementation for UPG.

Architecture. The local MCP server runs as a subprocess of the AI client. All transport is stdio; the server never opens a network port.



The `.upg` file is the only persistent state the local server needs. No database, network connection, or separate cache is required, though any of those can be layered on top for teams that want them. Anyone who can run a Node.js subprocess and read a JSON file can run the UPG MCP server. A graph reachable only through one vendor's cloud service is bound to that vendor; the local server ensures the ontology can be adopted, forked, and audited without that binding.

Why MCP as the protocol. §2.4 made the load-bearing argument; the implementation choice follows directly. The major AI coding clients already speak MCP at connection time, so building on MCP is building on the layer the agents are already speaking.

Why not GraphQL. The comparison is natural: GraphQL and UPG MCP both carry a type system with relationships, both expose schema introspection, and both structure client-server interaction around a graph. The MCP design makes three moves GraphQL does not: agents reach for named tools rather than compose queries (no intermediate query language to write); introspection is a call-time safety rail (`get_entity_schema` answers *"what would be valid here, right now?"* before a write) rather than a build-time codegen input; and primitives like triage reads (`get_graph_digest`), typed session context, and diff-based collaboration have no GraphQL-native expression. GraphQL is a query language for data retrieval; UPG MCP is an invitation-driven protocol for agent-graph interaction, and the inversion matters for the agent-facing case.

Four moves beyond CRUD. The MCP server's AI-first shape reduces to four design moves, each of which Appendix E expands with tool-by-tool detail.

1. **Introspection-first.** `get_entity_schema` makes the ontology queryable at write time, so the same agent-side code works against any spec version without retraining. A call to `get_entity_schema({ type: 'hypothesis' })` returns the valid parents, the typed property interface (`we_believe`, `will_result_in`, `we_know_when`, `we_test_by`), the permitted outgoing edges (including `hypothesis_requires_experiment`), and the lifecycle phase model (`untested → testing → resolved`). The agent writes a `create_node` that conforms on the first attempt because the shape was queried rather than guessed.
2. **Triage as a first-class read primitive.** `get_graph_digest` answers *where to focus* rather than *what data to fetch*, distinct from the record-access reads (`get_node`, `list_nodes`, `search_nodes`) that serve retrieval.
3. **Session context as a typed artefact.** `get_session_context` and `update_session_context` carry working memory across conversations as structured data, not conversation history. This is the compounding property of §1.4 realised at the tool layer: a new conversation opens with `get_product_context` and inherits the active playbook step, the recent-change cursor, the active lens, and the open hypotheses.
4. **Change-based collaboration.** `get_changes` returns a diff since a sync point, and `get_sync_state` compares local and cloud heads. Two agents reconcile by diffing cursors rather than by acquiring locks, which makes long-running AI sessions compatible with concurrent edits.

Cloud variant. `@unified-product-graph/cloud-server` backs the same tool surface with PostgreSQL for teams that need multi-user collaboration, incremental sync, and optimistic locking per node. Tool signatures are identical; a client switches by changing the server address, and graphs round-trip between local and cloud.

Client compatibility. Any MCP-capable client works with UPG. At the time of writing, that includes the major AI coding editors.

Full MCP tool inventory, response shapes for introspection, batching, session and product context, graph digest, area-scoped traversal, change-based collaboration, and dual-transport details: Appendix E.

4.3 Import Adapters

The MCP server is how new work enters the graph. Existing work, tickets in Linear, pages in Notion, specs in Markdown, also needs an entry point. Import adapters are that entry point: each reads an external corpus and produces draft UPG nodes and edges a human can review before committing.

Four adapters ship today.

Adapter	Source	Primary inputs	Canonical UPG mapping
Markdown	`.md` files	Headings, bullets, YAML frontmatter	Headings → parent nodes, bullets → children, frontmatter → `properties`
Notion	Databases and pages	Database items, page content, linked-database relations	Database item → node (schema drives `properties`); linked-database relation → typed edge
Linear	Issues and projects	Issues, projects, labels, cross-links	Issue or project → node (label drives type); label → `tags`; cross-link → typed edge
GitHub	Repos, issues, PRs	Issues, PRs, labels, milestones	Issue or PR → node (label drives type); milestone → parent via hierarchy edge

Adapter mapping in practice. A Linear issue labelled `persona`, titled *"Felix, solo builder"*, with body prose describing the archetype, becomes a `persona` node: `title` from the issue title, `description` from the body, `source_id` set to the Linear issue ID, and `mapping_confidence: 'high'` because the label matched a canonical type. The same issue without a UPG-aligned label is imported as a generic `document` with `mapping_confidence: 'low'` and surfaced for human type selection. Source-traceability and mapping-confidence semantics (§3.4) apply to every imported node; the draft is always reviewed before commit.

Pre- and post-import steps. A direct structural map is often not enough for loosely structured sources. An adapter can run pre-import analysis over the source (title clustering, semantic similarity, entity extraction over prose) to surface candidates the source did not explicitly label, and post-import structural checks (missing edges,

orphan nodes, domains the source implies should exist) to flag gaps. Current adapters implement these steps to varying degrees; extending them uniformly is part of the adapter-ecosystem work in §6.2.

An honest gap. Current adapters handle structured sources well (Linear, Notion databases, GitHub) and prose sources crudely (Markdown). A meaningful fraction of product knowledge lives in less structured places: Google Docs, Slack threads, meeting transcripts, interview recordings. A semantic-graph-assisted adapter that ingests prose at scale, proposes a draft UPG structure, and lets a human accept or revise it is a near-term priority (§6.2).

Import adapter mapping tables and pre-processing heuristics: see the adapter source at `packages/upg-adapters/` *in the specification repository.*

4.4 Guided Skills, Playbooks, and Approaches

Adapters bring existing work in. Playbooks and approaches are the other side: structured modes for producing *new* work and engaging with what exists. §3.7 introduced playbooks and approaches as Layer 5 of the specification; this subsection describes how the reference implementation runs today.

The current reference implementation. Slash-command skills inside Claude Code (`/upg-new-persona`, `/upg-new-hypothesis`, `/upg-check-gaps` and others) are the per-surface bindings of canonical `UPGPlaybook` and `UPGApproach` definitions. Playbook structure lives in `@unified-product-graph/core`; the experience lives in a `PlaybookBinding` for the target surface. The same playbook definition binds to a Cursor command, a VS Code extension, or a browser canvas without change: the structure-shared, experience-per-surface split described in §3.7.

Three modes cover the current set.

Mode	Skills	Bound to	What it does
Playbook	`/upg-new-persona`, `/upg-new-strategy`, `/upg-show-journey`, and others	Canonical playbook for each region	Elicits structured knowledge through conversation; persists as typed entities, step by step
Inspect / Plan	`/upg-check-gaps`, `/upg-show-status`	`inspect` and `plan` approaches	Reads the current graph; surfaces missing entities, benchmark violations, broken chains
Reflect	`/upg-reflect`, `/upg-new-hypothesis`	`reflect` approach	Facilitates structured questioning (Five Whys, Pre-mortem, Red Team); stress-tests assumptions

The conversation is the data entry. A skill does not ask the user to draft a persona document and then extract entities later. Every question resolves to a `create_node` or `update_node` call at the moment the user answers; prose never becomes an intermediate artefact. The graph is updated live, one utterance at a time. This is the discipline that keeps capture and structure as the same step: the trap the document paradigm fell into is that writing and structuring were two different steps, and the second one was always skipped.

Playbook and approach schemas: Appendix A. Full MCP tool inventory including `list_playbooks`*,* `list_approaches`*, and the five bare-verb approach tools: Appendix E.*

4.5 UPG Markdown: The Hybrid Format

Guided playbooks capture entities through conversation; UPG Markdown captures them through prose. A `.upg` file is the canonical JSON serialisation of a product graph; a `.upg.md` file is an alternate surface over the same graph: standard markdown with typed entity references embedded inline. Neither format is derived from the other; both serialise the same underlying structure, and a UPG-aware tool can round-trip between them. A reference takes one of three forms: `[[type:id]]` for a bare mention, `[[type:id|property:value]]` for a mention that asserts property values at that point of reference, and `[[+type:id]]` for a creation reference. A paragraph reads as natural prose to a human and parses as structured references to a machine:

> Felix, a PERSONA **`felix-solo-builder`**, repeatedly hits NEED **`volume-vs-value`** every time three feature requests stack up and the loudest one is not obviously the right one.

A memo written in `.upg.md` is simultaneously a document, read linearly and committed to git, and a graph composition whose references resolve to live records. The memo is a curated view over the graph rather than a parallel copy that drifts from it.

Full grammar, resolver behaviour, the longer worked memo, staleness detection, and the rendering implementation: Appendix F.

4.6 Self-Hosting

UPG is used to model the product that defines it. The reference `.upg` file maintained by the authors was built incrementally across dozens of AI-assisted sessions, grew without schema migration as new domains became relevant (engineering entities appeared when architecture work began; growth entities appeared when business-model work started), and is the operational memory for ongoing development of Entopo, the visual graph application introduced in §1.5. Typed features trace back through the discovery-

to-delivery spine (`feature ⟵ hypothesis ⟵ solution ⟵ opportunity ⟵ need ⟵ job ⟵ persona`), and the gaps where chains break are visible in the graph rather than hidden in prose.

This is existence-proof of feasibility, not evidence of generalisation. The specification was designed to model a specific kind of product, and the self-hosted graph demonstrates that the specification is internally coherent enough to support sustained use by the team that wrote it. Controlled user studies across independent product teams, reference graphs built by external collaborators, and cross-domain corpus analysis are open work.

4.7 The Portfolio Document

§4.1 noted that the default unit is one product per file. The portfolio document (§3.10) is the multi-product case, and it reuses the same machinery: the same `$upg` header, the same canonical form, the same cross-file reference syntax. It is distinguished by `$upg.kind: "portfolio"` (the in-memory discriminator is `type: "portfolio"`), and in place of a single product's `product` / `nodes` / `edges` it carries a small set of top-level sections:

- `organization` — the single organisation that owns the portfolio, the same identity summary the header surfaces.
- `portfolios` and `product_areas` — the hierarchy nodes: nestable portfolios (the strategic axis, *where we invest*) and product areas (the organisational axis, *who owns what*).
- `products` — each member product with its own `nodes` and `edges` embedded inline, so a portfolio document is one self-contained, portable snapshot. Each embedded product retains the exact shape it has as a standalone `.upg` file.
- `cross_edges` — the cross-product edges (§3.10), each referencing its endpoints by qualified id.
- `registry` — optional and additive: the shared-vocabulary tier, a `{ nodes, edges? }` object of canonical entities. A portfolio without a registry omits the section entirely and stays byte-identical; an empty registry is never serialised.

Two properties follow from this design, both deliberate. First, **the format is additive.** A single-product `UPGDocument` remains valid unchanged; a portfolio is a strict superset that wraps products which still open, validate, and version as ordinary graphs. A team can adopt the portfolio tier over an existing fleet of single-product graphs without rewriting any of them, and a local workspace typically maintains the members as separate files and assembles them into one portfolio document only on export. Second, **the registry is reached by a reserved address.** Canonical entities are addressed as `registry/{node_id}`, where `registry` is a reserved pseudo product-id no real product

may claim; the targets of `instance_of`, the area-to-audience edges, and the foundations edges all resolve through it. The registry is therefore a third tier without being a third file format: a namespaced section inside the portfolio document, separating the company's structure (portfolios and areas) from its shared vocabulary (the registry) without adding a loader, a schema, or a tier for every adopter to learn.

The full `UPGPortfolioDocument` interface, the registry shape, and a worked `portfolio.upg` example are in Appendix O.

---

5. Design Principles

The implementations described in Section 4 (file format, MCP server, adapters, guided skills, markdown surface) are surfaces over a single ontology. They cohere only because the ontology itself is constrained. A persona written by a skill has to match a persona parsed from a memo has to match a persona imported from Notion. The Unified Product Graph (UPG) specification governs this convergence through **twenty invariants**, refined across three review cycles and grouped into five clusters.

Each principle earns its place by being an *invariant*: a property of the ontology that, if violated, admits a class of failure we have seen cost projects time, interoperability, or trust. Some are enforced by automated tests against `@unified-product-graph/core`; others by manual review against the governance document. One principle runs through the other nineteen. **P16** states it directly: *UPG is machine-written, human-readable*. No human writes a `.upg` file by hand. The format is produced by agents, adapters, and tools; consumed by AI inference, renderers, and other agents. Every entity type name, property key, and edge verb is parseable from natural language. The clusters that follow all serve that commitment.

How types are named (Vocabulary): P1, P2, P7, P9

Vocabulary determines whether a practitioner describing a `job` in prose and an agent writing a `job` node converge without translation. The penalty for getting this wrong is small per session and compounds badly over time: every jargon translation is a place where agent output diverges from the canonical type. P1 keeps type names readable without a glossary; P2 keeps the data layer framework-neutral so framework vocabularies remain display labels; P7 collapses overlapping types into one canonical type with a discriminator property; P9 anchors naming inside each domain to that domain's universal practitioner vocabulary.

P7 in practice. Early versions of the specification carried framework-specific types for the same underlying concepts. The discriminator property that replaced each consolidation set carries the information the framework-specific types were carrying implicitly: `pain_point`, `user_need`, and `desired_outcome` became `need` with `valence: pain | gap | constraint`; `kpi`, `north_star_metric`, and `input_metric` became `metric` with `designation`; `design_decision`, `product_decision`, and `architecture_decision` became `decision` with `layer`. Made explicit, one canonical type suffices and every framework view still renders its preferred label. P7 does not forbid distinctions between concepts: it forbids creating a type when a property would do.

How types compose (Grammar)

P4, P5, P18, P6

Grammar is what keeps the graph structurally meaningful. P4 makes hierarchy a first-class spec concept, so an agent cannot place a `database_schema` under a `persona`: the validator catches it against the spec alone. P5 and P18 give every edge a semantic verb and a reverse form, so the relationship reads naturally from either endpoint. P6 keeps domains as coherent semantic groupings rather than buckets of convenience.

When something earns being an entity (Graph Shape): P14, P19, P20

This cluster is where UPG diverges most sharply from ad-hoc modelling. It asks, for any concept, whether it is an entity, an edge, or a property. P14 keeps the same relationship from being stored in two places (edge and foreign key) that can disagree. P19 upgrades repeatedly-referenced values (statuses, categories, metric names) from strings to first-class nodes. P20 gives the ontology an operational test for whether a candidate type deserves to exist at all.

P20 operates on a different axis from P7. Where P7 asks whether two candidate types should collapse into one, P20 asks whether a single candidate type earns the right to exist at all. The operational test is mechanical: remove the entity and rewire its edges directly between its neighbours. If the graph loses no information, the entity was a pass-through. If it loses a hub role, a container role, or narrative meaning that did not belong on any single edge, the entity earns its place.

What entities carry (Properties)

P3, P13, P17, P15

Properties are where judgment becomes data. P3 forbids escape hatches so the schema cannot degrade into a catalog of titles. P13 prevents two entity types from inventing divergent enum values for the same conceptual axis. P17 keeps qualitative meaning separate from computational convenience: a slider reading 3/5 is not the same as "Severe pain," and the ontology refuses to let that distinction collapse. P15 makes status portable across tools; the universal phases hold everywhere, and granular states extend without breaking interop.

How the spec evolves (Governance): P8, P10, P11, P12

A standard fractures the moment two implementations quietly disagree. The governance principles are what let the ontology grow over years without splintering. P8 prevents forking of the canonical vocabulary into app-specific dialects. P10 prevents

the specification from becoming an opinion paper. P11 keeps the extension path from becoming an escape hatch that breaks interoperability. P12 keeps documentation and types in the same place, so the spec cannot drift from its docs.

A note on numbering. Principle identifiers reflect the order in which each invariant was recognised and added to the specification; the clusters above are a later semantic grouping applied once the full set was stable. A cluster-ordered reference index is provided at the end of Appendix D.

Appendix D expands each of the twenty principles with the specific failure mode it prevents and the mix of automated tests and manual review that enforces it.

---

6. Expanding UPG

The specification today covers the software-product-creation graph. The sections below collect the open work: four domains staged as the next ontology expansions, and several areas where the standard will grow through contributions from outside the authors.

6.1 New Domains

Four domains sit outside UPG's current coverage. Each needs native entity types, edge semantics, and lifecycle models rather than workarounds through the types that exist today.

Domain	Key missing entity types	Why current spec is awkward
Hardware products	`component`, `assembly`, `bill_of_materials`, `tolerance_spec`, `manufacturing_process`, `supplier_part_number`	The discovery-to-delivery spine shifts: hardware's *learning validates feature* involves physical prototyping, certification cycles, and production ramp stages that software does not have
Non-profit organisations	`programme`, `beneficiary`, `theory_of_change`, `outcome_indicator`, `funding_cycle`, `grant_application`	Revenue streams and pricing tiers are structurally awkward for organisations measured on mission impact rather than profit
Creative agencies	`client_engagement`, `billable_phase`, `deliverable_package`, `creative_brief`	Treating each client as a separate product is a workaround; client engagements have their own lifecycle distinct from product iteration
Government services	`policy_objective`, `statutory_requirement`, `citizen_cohort`, `service_standard`	Compliance domain coverage is thin; government delivery cycles and accountability structures differ structurally from commercial product teams

Each is a multi-month ontology effort. The path forward uses the `UPGEntityTypeMaturity` lifecycle (`draft ⟶ proposed ⟶ stable ⟶ deprecated ⟶ removed`) so community proposals can ship in `.upg` files and accumulate usage data before entering the stable specification. We invite collaborators in each sector to co-propose the entity catalog, lifecycle model, and edge semantics for their domain.

6.2 Contributors Welcome

Four areas where the standard benefits from work outside the authors:

Area	Current state	What's open
Schema evolution	Single-type migrations (rename, merge-with-default): see Appendix G	Multi-type structural migrations: splitting one type into two, or merging two with field-reconciliation rules
Collaborative editing	One file per product; cloud server provides optimistic locking	True CRDT-based collaborative graph for simultaneous multi-user editing without lock contention
Interoperability testing	Reference implementation in TypeScript; format is stable, typed, and documented	An independent implementation that round-trips `.upg` files against the reference: the real portability test
Adapter ecosystem	Markdown, Notion, Linear, GitHub	Figma, Jira, Productboard, Miro/FigJam, Google Docs, Slack threads, interview recordings: a semantic-graph-assisted adapter that ingests prose at scale, proposes draft UPG structure, and lets a human accept or revise

Governance for all of this runs through a public RFC process in the specification repository. The standard grows when other people's work lands in it.

---

7. Conclusion

AI expanded what one person can produce. It did not expand what one person can hold in their head. The gap widens with every model release, and the tools piling more output into folders, chat histories, and scattered documents widen it further rather than closing it.

The Unified Product Graph gives product knowledge a shape: typed entities, directed relationships with forward and reverse verbs, lifecycle phases and states, a portable JSON file format, and a migration system that lets the ontology evolve without breaking files already in circulation. The `.upg` file is agnostic about which tool writes it and which tool reads it, so long as product knowledge carries structure, relationships stay explicit, and no context is lost when data moves between systems or between AI sessions.

The standard is early. The ontology will grow, shift, and occasionally shrink as use cases clarify what was wrong. The evidence is preliminary, the community is small, and the hard questions of governance, independent implementation, and formal evaluation are ahead rather than behind.

The thesis is what the paper has argued and what the authors have run in production long enough to trust: structured memory, curated jointly by practitioners and the AI agents they work with, compounds across sessions, roles, and tools where unstructured output dissolves. A product's memory is built once and written into by every session thereafter. That is what makes the next session cheaper than the last.

UPG is open source, MIT-licensed, and available at **unifiedproductgraph.org** (https://unifiedproductgraph.org).

References

Placeholder citations for the final version, to be expanded with proper bibliographic entries.

- Anthropic (2024). *Model Context Protocol specification.* https://modelcontextprotocol.io (https://modelcontextprotocol.io)
- Christensen, C. M. et al. (2003). *The Innovator's Solution.* Harvard Business Review Press.
- data.world (2024). *Benchmark: Increasing the Accuracy of LLMs in the Enterprise with a Knowledge Graph.* ACM GRADES-NDA.
- Doerr, J. (2018). *Measure What Matters.* Portfolio/Penguin.
- Edge, D. et al. (2024). *From Local to Global: A Graph RAG Approach to Query-Focused Summarization.* Microsoft Research. arXiv:2404.16130.
- Grove, A. (1983). *High Output Management.* Random House.
- Intercom. *RICE: Simple Prioritization for Product Managers.* https://www.intercom.com/blog/rice-simple-prioritization-for-product-managers/ (https://www.intercom.com/blog/rice-simple-prioritization-for-product-managers/)
- Karpathy, A. (2025). *LLM-maintained knowledge wikis.* Referenced informally via public writings and demonstrations.
- Liu, N. F. et al. (2024). *Lost in the Middle: How Language Models Use Long Contexts.* TACL. https://direct.mit.edu/tacl/article/doi/10.1162/tacl_a_00638/119630/ (https://direct.mit.edu/tacl/article/doi/10.1162/tacl_a_00638/119630/)
- Maurya, A. (2012). *Running Lean.* O'Reilly Media.
- Miller, G. A. (1956). *The Magical Number Seven, Plus or Minus Two.* Psychological Review, 63(2), 81–97.
- Osterwalder, A. & Pigneur, Y. (2010). *Business Model Generation.* Wiley.
- Torres, T. (2021). *Continuous Discovery Habits.* Product Talk LLC.
- W3C (2004). *Resource Description Framework (RDF).* https://www.w3.org/RDF/ (https://www.w3.org/RDF/)
- W3C (2012). *OWL 2 Web Ontology Language.* https://www.w3.org/TR/owl2-overview/ (https://www.w3.org/TR/owl2-overview/)

Technical Appendix

The appendix carries the specification detail referenced from the main body. Each appendix is self-contained; they can be read in any order.

Appendix A

The Layered Architecture in Detail

The specification (`@unified-product-graph/core`) is organised into five layers composed of self-contained modules with a strict dependency flow. Lower layers know nothing about higher layers. Higher layers compose lower layers without modifying them.

Layer 1. Foundation

Three modules answer *what exists in the graph and what shape does it take?*

- `catalog/` enumerates what the ontology knows: the union of stable entity types (including deprecated) and the full map of edge types. Zero imports. It is the true foundation, depended on by every other module.
- `shapes/` defines structural primitives: the `UPGBaseNode` interface every entity implements, the `UPGEdge` interface every relationship implements, the `UPGDocument` format for serialisation, and the `UPGPortfolioDocument` format for multi-product graphs.
- `registry/` provides identity and organisation: stable `type_id` values (see Appendix B), maturity flags, `since` and `deprecated_in` version tracking, domain membership, and the semantic domain definitions (name, description, maturity).

Layer 2. Grammar

`grammar/` defines the rules of valid graphs: which entity types may be children of which parents (hierarchy), what lifecycle phases each type progresses through (lifecycles), how schema evolves across versions (migrations), and validation routines that tools can import to check structural legality without running the MCP server.

Layer 3. Properties

`properties/` provides typed property schemas per entity type. Domain interface files (`properties/domains/engineering.ts`, `properties/domains/users.ts`, etc.) declare TypeScript interfaces that are auto-compiled into a runtime `UPG_PROPERTY_SCHEMA` map. Tools validate incoming property bags against these schemas; MCP servers expose them through `get_entity_schema` (Appendix E).

Layer 4. Output

Four modules answer *how is the graph shown, evaluated, and projected?*

- `presentation/` provides labels (framework-specific display names per entity type), role-based lenses that filter domains and relabel entities, and concentric `domain-rings` groupings for spatial navigation.
- `intelligence/` encodes graph health with declarative benchmarks:
- { entity_type: 'hypothesis', stage: 'validation', healthy_range: { min: 3, max: 12 }, warning_below: 1, message_below: 'Features are being built without recorded hypotheses. ' + 'Traceability from evidence to shipped code is breaking.', }, { ratio: { numerator: 'feature', denominator: 'hypothesis' }, stage: 'validation', healthy_ratio_max: 5, warning_ratio_above: 10, message_above: 'More than 10 features per hypothesis. Most work is running ' + 'ahead of validation, expect late-stage rework.', }

  Benchmarks run on any compliant graph and produce structural observations, not prescriptions. The intelligence is the ontology's self-measurement, not an AI layer on top. The module also holds domain usage guides that MCP agents consult when constructing entities.
- `frameworks/` holds declarative framework definitions (Appendix H) that remap canonical entity types to framework-specific vocabularies and canvas layouts.
- `regions/` rolls the atomic domains up into the eleven super-domains described in §3.3. The module is topology-only: each region declares its entity roster, edge flow, anchor entity, and shape archetype. Rendering concerns (UCS pattern assignments, co-rendering scenarios, design implications, workspace archetypes) sit in application code rather than the spec, so the wire format stays presentation-neutral.

Layer 5. Playbooks and Approaches

Two modules compose the procedural layer.

`playbooks/` holds region-anchored creation sequences. A `UPGPlaybook` answers *"what should I create, and in what order, to populate this region?"* The module exports:

- `UPGPlaybook` carries the structure: `id` (namespace-prefixed `playbook:<region>[-variant]`), `region` (required: the `UPGRegionId` this playbook anchors), `is_canonical` (true for the one canonical playbook per region), an optional `framework_id` for specialised framework-anchored playbooks, and an ordered `creation_sequence` of steps.

- `PlaybookBinding` carries the surface contract. Structure lives in `@unified-product-graph/core`; surface experience (terminal, IDE, canvas) binds to the same playbook through its own `PlaybookBinding`, so a playbook runs identically across surfaces while each surface retains its native idiom.

The catalog holds 13 playbooks across 11 regions: one canonical per region and 2 specialised variants.

`approaches/` holds cross-cutting cognitive orientations. A `UPGApproach` answers *"how should I engage with what already exists in this graph?"* The catalog is closed at v0.8.0: five canonical entries (`plan`, `inspect`, `prioritise`, `trace`, `reflect`), each a bare-verb MCP tool. Their per-approach semantics, signatures, and the cartographic framing are tabled in §3.7 and detailed in Appendix K.

Steps are discriminated by `kind` across both playbooks and approaches. Four kinds compose the shared step grammar:

- `domain_guide` defers to the Intelligence module's domain usage guide for a named domain.
- `framework` applies a framework from the Frameworks module.
- `entity_sequence` fixes an ordered list of entity types to capture.
- `sub_sequence` nests another playbook or approach, so longer journeys reuse shorter ones without duplicating steps.

Because Layer 5 composes Layer 4 and Layers 1 through 3, a team using UPG without any playbooks or approaches still has a valid, compounding graph. Playbooks and approaches are scaffolding for guided production and engagement; they do not alter the underlying structure.

Dependency Flow

```
catalog/       ← zero imports (true foundation)
    ↓
shapes/        ← imports from catalog/
    ↓
registry/      ← imports from catalog/
    ↓
grammar/       ← imports from catalog/, shapes/, registry/
    ↓
properties/    ← imports from shapes/, grammar/
    ↓
presentation/  ← imports from registry/, grammar/, frameworks/
intelligence/  ← imports from catalog/, shapes/, registry/
frameworks/    ← self-contained
regions/       ← self-contained
    ↓
playbooks/     ← imports from intelligence/, frameworks/, regions/
approaches/    ← imports from regions/, frameworks/
```

Critical design decision. Layers 1 through 3 are storage concerns. Layers 4 and 5 are read-time concerns. The `.upg` file contains only Layers 1 through 3. Lenses, frameworks, benchmarks, regions, playbooks, and approaches are never written to the file. The same graph data can be presented, evaluated, projected, and guided through any number of output, playbook, and approach modules without data migration.

Appendix B

Entity Model

UPGBaseNode

```
interface UPGBaseNode {
  id: string                    // Unique identifier (canonical)
  type: UPGEntityType           // Canonical UPG entity type
  title: string                 // Human-readable name
  slug?: string                 // Stable handle for inline [[type:slug]]
chips
                                //   in .upg.md; unique within (product, type)
  aliases?: string[]            // Past slugs, so renamed chips still resolve
  description?: string          // Narrative description
  tags?: string[]               // Freeform tags
  status?: string               // Lifecycle phase (per P15)
  source_id?: string            // ID in originating tool (round-trip
fidelity)
  source_type?: string          // Type name in originating tool
  mapping_confidence?:          // Reliability of the type mapping
    'high' | 'medium' | 'low' | 'manual'
  external_tool?: string        // Tool holding the canonical artifact
  external_ref?: string         // URI to external artifact
  external_id?: string          // ID in the external tool (sync / round-trip)
  properties?: Record<string, unknown>  // Type-specific fields per Layer 3
}
```

EntityTypeMeta

Every entity type carries an immutable identifier alongside its name. The `type_id` (`ent_001`, `ent_003`...) never changes even as names do; this is the wire-format stability contract.

```
export interface EntityTypeMeta {
  name: string                              // 'need' (may change across
versions)
  type_id: string                           // 'ent_313' (immutable, never
changes)
  maturity:
    | 'draft'
    | 'proposed'
    | 'stable'
    | 'deprecated'
    | 'removed'
  since: string                             // UPG version introduced
  deprecated_in?: string                    // UPG version deprecated
  removed_in?: string                       // UPG version removed
  replacement?: string                      // Canonical replacement (if
deprecated)
}

// Examples from the registry:
{ name: 'product',  type_id: 'ent_001', maturity: 'stable',      since:
'0.1.0' }
{ name: 'outcome',  type_id: 'ent_002', maturity: 'stable',      since:
'0.1.0' }
{ name: 'kpi',      type_id: 'ent_003', maturity: 'deprecated',
                    since: '0.1.0', deprecated_in: '0.1.0', replacement:
'metric' }
{ name: 'metric',   type_id: 'ent_006', maturity: 'stable',      since:
'0.1.0' }
{ name: 'persona',  type_id: 'ent_047', maturity: 'stable',      since:
'0.1.0' }
{ name: 'need',     type_id: 'ent_313', maturity: 'stable',      since:
'0.1.0' }
```

When `pain_point` was merged into `need` with a `valence: 'pain'` property (Appendix G), the `type_id` of the original `pain_point` was preserved in the migration record so old `.upg` files auto-migrate on load without data loss. Names belong to humans; `type_ids` belong to the wire format.

Entity Types by Domain Ring

Representative canonical types per ring. The Types column gives the stable count at the current spec version; the sample column is illustrative, not exhaustive.

Ring	Types	Sample Canonical Entity Types
0 Nucleus	5	organization, portfolio, product_area, workspace, framework_exercise
1 Understand	41	persona, job, need, opportunity, solution, hypothesis, experiment, learning, research_study, insight, competitor, competitor_signal, market_trend
2 Define	68	product, outcome, objective, key_result, feature, feature_area, user_story, release, user_journey, screen, design_component, design_token, business_model, value_proposition, revenue_stream, brand_identity
3 Build	94	specification, primitive, bounded_context, service, api_contract, service_level_indicator, test_suite, threat_model, event_schema, dashboard, ai_model, prompt_version, workflow_template, agent_definition
4 Grow	47	pricing_strategy, pricing_tier, gtm_strategy, positioning, ideal_customer_profile, funnel, acquisition_channel, growth_loop, deal, marketing_campaign_plan
5 Operate	23	support_ticket, customer_feedback, churn_reason, content_piece, knowledge_base_article, tutorial, walkthrough
6 Extend	38	team, role, stakeholder, program, project, milestone, locale, translation_key, partner_program, compliance_requirement, risk

Appendix C

Edge Model

UPGEdge and UPGEdgeDefinition

```
interface UPGEdge {
  id: string
  source: string                  // Source node ID
  target: string                  // Target node ID
  type: UPGEdgeType               // Catalog key, e.g. 'persona_pursues_job'
  mapping_confidence?: UPGMappingConfidence
  // Edge-scoped payload. Permitted ONLY on edge types whose catalog
  // definition sets carries_properties: true (currently
  // framework_exercise_includes_node, which stores a framework's per-entity
  // result here: a MoSCoW bucket, a RICE score, a canvas slot). A value that
  // exists only within a specific exercise is a fact about the relationship,
  // not about either endpoint. Plain edges stay payload-free; validators
  // reject properties on edges that do not opt in.
  properties?: Record<string, unknown>
}

interface UPGEdgeDefinition {
  forward_verb: string            // Active voice: "source [forward_verb]
target"
  reverse_verb: string            // Reverse reading: "target [reverse_verb]
source"
  classification: 'hierarchy' | 'causal' | 'semantic' | 'cross-domain'
  source_type: string             // Which entity type may originate this
edge
  target_type: string             // Which entity type it may terminate on
  carries_properties?: boolean    // Opt-in to the gated edge-property model.
                                  //   When true, instances MAY carry
properties
                                  //   (validated); when absent, validators
                                  //   reject any properties on the edge.
}
```

Edge Catalog Excerpts Across Classifications

The edge catalog enumerates every legal relationship in the ontology. Every definition specifies both directions of the verb pair.

```
export const UPG_EDGE_CATALOG = {

  // ——— Semantic: meaning associations ———————————————————————

  persona_pursues_job: {
    forward_verb: 'pursues',        reverse_verb: 'pursued_by',
    classification: 'semantic',
    source_type: 'persona',         target_type: 'job',
  },
  persona_experiences_need: {
    forward_verb: 'experiences',    reverse_verb: 'experienced_by',
    classification: 'semantic',
    source_type: 'persona',         target_type: 'need',
  },

  // ——— Causal: cause-effect chains ——————————————————————————

  job_surfaces_need: {
    forward_verb: 'surfaces',       reverse_verb: 'surfaces_from',
    classification: 'causal',
    source_type: 'job',             target_type: 'need',
  },
  solution_proposes_hypothesis: {
    forward_verb: 'proposes',       reverse_verb: 'tests',
    classification: 'causal',
    source_type: 'solution',        target_type: 'hypothesis',
  },
  hypothesis_requires_experiment: {
    forward_verb: 'requires',       reverse_verb: 'validates',
    classification: 'causal',
    source_type: 'hypothesis',      target_type: 'experiment',
  },
  experiment_produces_learning: {
    forward_verb: 'produces',       reverse_verb: 'learned_from',
    classification: 'causal',
    source_type: 'experiment',      target_type: 'learning',
  },
  learning_updates_hypothesis: {
    forward_verb: 'updates',        reverse_verb: 'updated_by',
    classification: 'causal',
    source_type: 'learning',        target_type: 'hypothesis',
  },

  // ——— Cross-domain: bridges between rings ——————————————————

  opportunity_addresses_need: {
    forward_verb: 'addresses',      reverse_verb: 'addressed_by',
    classification: 'cross-domain',
    source_type: 'opportunity',     target_type: 'need',
```

```
    forward_verb: 'pursues',        reverse_verb: 'pursued_by',
classification: 'cross-domain',    source_type: 'opportunity',
target_type: 'outcome',  },  insight_informs_opportunity: {    forward_verb:
'informs',        reverse_verb: 'informed_by',    classification: 'cross-
domain',    source_type: 'insight',         target_type: 'opportunity',  },
// The one property-carrying edge: a framework exercise's per-entity result
// rides this relationship (see "edge-property model" above).
framework_exercise_includes_node: {    forward_verb: 'includes',
reverse_verb: 'included_in',    classification: 'cross-domain',
source_type: 'framework_exercise', target_type: 'node',  // polymorphic
carries_properties: true,  },  // ——— Hierarchy: parent-child containment
——————————————————————————————————————————————————————————————————————
feature_area_contains_feature: {    forward_verb: 'contains',
reverse_verb: 'contained_in',    classification: 'hierarchy',    source_type:
'feature_area',    target_type: 'feature',  },
feature_refined_by_user_story: {    forward_verb: 'refined_by',
reverse_verb: 'refines',    classification: 'hierarchy',    source_type:
'feature',         target_type: 'user_story',  },
research_study_produces_insight: {    forward_verb: 'produces',
reverse_verb: 'produced_by',    classification: 'hierarchy',    source_type:
'research_study',  target_type: 'insight',  },  // ... additional
definitions} as const
```

Classification counts

Classification	Count	Purpose
Hierarchy	436	Parent-child containment
Cross-domain	382	Bridges between rings
Causal	89	Cause-effect chains
Semantic	78	Meaning associations

The 985 within-product catalog edges divide across four classifications. The hierarchy-heavy distribution reflects that most edges in the catalog define legal parent-child relationships per P4. The 89 causal edges and 382 cross-domain edges carry the narrative spines (§3.6); the 78 semantic edges carry associations that are neither hierarchical nor causal.

Cross-Product and Registry Edges

The 985 catalog edges above connect typed nodes *within* one product. The portfolio tier (§3.10) adds a further set that operates across products.

Cross-product edges. A separate closed union of twenty-four cross-product edge types (`UPG_CROSS_EDGE_TYPES`) connects nodes that live in different product files, referencing endpoints by qualified id (`{product_id}/{node_id}`, or `registry/{node_id}`). They are not part of the within-product catalog and do not count toward the 985. The full set, by family:

Family	Edge types
Peer equivalence (symmetric)	`shares_persona`, `shares_competitor`, `shares_metric`
Product relationships	`depends_on_product`, `hosts`, `succeeds`, `cannibalises`
Strategic rollup	`contributes_to`, `rolls_up_to`
Canonical instance	`instance_of`
Area to audience	`area_serves_persona`, `area_targets_market_segment`
Foundations	`product_implements_specification`, `product_exposes_specification`, `feature_conforms_to_specification`, `api_contract_speaks_specification`, `product_exposes_primitive`, `feature_manipulates_primitive`, `primitive_stored_as_data_type`
Competitive intelligence	`feature_rivals_competitor_feature`, `competitor_signal_maps_to_feature`, `competitor_signal_surfaces_opportunity`, `competitor_classified_as_classification_value`, `node_classified_as_classification_value`

`instance_of` carries an optional `alias` flag (a sanctioned product-local name); `area_serves_persona` and `area_targets_market_segment` carry optional `relevance` (primary/secondary) and `audience_role` qualifiers, the matrix nuance an area record cannot hold on its own.

Registry-internal and portfolio-hierarchy edges. Two sets of edges that *look* portfolio-scoped are in fact ordinary catalog entries, counted in the 985 above. The portfolio hierarchy edges (`organization_invests_via_portfolio`, `portfolio_contains_product`, `product_area_contains_product`, and their nesting variants) connect Nucleus-ring structure nodes; the Foundations registry-internal edges (`specification_extends_specification`, `specification_competes_with_specification`, `primitive_defined_by_specification`, `primitive_composes_primitive`, `specification_governed_by_organization`) connect canonical specifications and primitives to each other. The distinction is directional, not arbitrary: an edge that

crosses *from* a product graph *into* the registry is a cross-edge; an edge *between* two registry or structure nodes is a catalog edge, validated by the same `source_type`/`target_type` rules as any other.

Appendix D

The 20 Principles in Full

The specification is governed by 20 principles, locked after three review cycles. Enforcement is a mix of automated tests against the specification package and manual review against the governance document. For each, we state what the principle *prevents*, not merely what it *is*.

Vocabulary

P1. Readability Over Brevity. Every entity type name is understandable by a non-specialist without a glossary. The type is `job`, not `jtbd`. Industry numeronyms are permitted only when universally understood (`a11y`, `api`). *Prevents: AI agents translating natural language into jargon. Readable names mean the agent's output and the type system converge; the conversation and the graph speak the same language.*

P2. Framework-Neutral Naming. Entity types represent universal concepts, not framework-specific terminology. Framework vocabularies are display labels resolved at render time by the Rosetta Stone label map. The type is `need`, not `pain_point` (Lean), `frustration` (Empathy Map), or `barrier` (Design Thinking). *Prevents: The ontology from picking winners.*

P7. One Concept, One Type. No two types represent the same underlying concept. Overlapping types are consolidated with a discriminator property: `pain_point + user_need + desired_outcome ⟶ need` with `valence`. `kpi + north_star_metric + input_metric ⟶ metric` with `designation`. *Prevents: Type proliferation where adjacent frameworks each contribute their own near-duplicate type.*

P9. Naming Follows Domain Language. Within each domain, naming follows that domain's *universal* practitioner vocabulary, not one framework's house style. Engineering uses `bounded_context`, `aggregate`, `domain_event` (DDD). SRE uses `sli`, `slo`, `error_budget` (Google SRE). `jtbd` is framework-specific, not domain-universal. *Prevents: Over-indexing on one framework inside a domain, or ignoring the domain's real vocabulary.*

Grammar

P4. Hierarchy Is Grammar. Valid parent-child relationships are a first-class specification concept, not an application convention. The spec defines which types can be children of which parents; tools validate tree structure using only the spec. *Prevents: An AI agent placing a* `database_schema` *as a child of a* `persona`*. Grammar prevents structural nonsense at the schema level.*

P5. Edges Must Have Verbs. Every edge type carries a human-readable verb. The `_has_` pattern is reserved for strict containment; all other relationships use semantic verbs. *Prevents: Edges becoming meaningless joins.*

P18. Edges Have Forward and Reverse Verbs. Every edge type carries a forward verb (source → target) and a reverse verb (target → source). Data stores one direction; display renders the appropriate verb based on the reading context. *"Persona pursues job" ↔ "Job pursued by persona." Prevents: Tools inverting verbs at render time, and AI agents guessing how a relationship reads when asked about the opposite endpoint.*

P6. Domains Are Semantic Boundaries. Domains are not arbitrary groupings. Entity types within a domain relate to each other more than to types in other domains; cross-domain relationships are expressed through edges, not mixed membership. *Prevents: Domains becoming bucket-of-convenience collections.*

Graph Shape

P14. Foreign Keys Are Edges. Relationships between entities are expressed through edges, not through foreign-key properties. A `persona_id` field on a node duplicates the relationship (once as an edge, once as a property). The canonical form is the edge. *Prevents: Contradictory state where the edge and the foreign key disagree.*

P19. Referenceable Values Are Entities. When a property value needs to be referenced, grouped by, or shared across multiple entities, it is an entity type with edges, not an inline string. If two entities need to agree on a value, that value is a relationship. *Prevents: Fragile string matching across the graph.*

P20. Entities Earn Their Existence. An entity with zero typed properties is valid; its value comes from the relationships it anchors, not the properties it carries. An entity earns its place when it plays at least one role:

- **Hub.** Connects three or more entities that would not otherwise relate (e.g. `positioning` hubs `value_proposition`, `competitor`, `persona`, `market_segment`).
- **Container.** Has children in the hierarchy (e.g. `positioning` parents `messaging`, `objection`, `proof_point`).
- **Narrative.** Carries meaning in its title or description that does not belong on any single connected entity.

An entity is questioned when it has zero properties, connects only two entities, has no children, and its title adds no meaning beyond the edge itself. *Prevents: Entity-proliferation by intuition. Operational test: remove the entity, rewire its edges directly between neighbours; does the graph lose information?*

Properties

P3. Complete Property Schemas. Every entity type has a typed property interface. `Record<string, unknown>` is not acceptable for any standard type. *Prevents: The schema from being a catalog of titles.*

P13. Shared Vocabulary Axes. When the same conceptual axis (e.g., functional/emotional/social) applies to multiple entity types, it uses the same values and vocabulary. The axis is shared, not duplicated with different names per type. *Prevents: Semantic drift where shared axes have divergent enum values across types.*

P17. Assessments Are Qualitative With Numeric Encoding. Judgments like reach, frequency, severity, importance, and pain carry *both* a numeric value (for computation) and a qualitative label (for display). Scale definitions provide the mappings. Computed scores are normalised to 0–1 for cross-tool comparison. *Prevents: A slider reading `3/5` where the user meant "Severe pain."*

P15. Two-Level Lifecycle. Status is two levels: phases (universal vocabulary fixed by the spec) and states (granular positions within a phase, extensible by tools). See §5 *What entities carry (Properties)* in the main body for the TypeScript shape. *Prevents: Status from being the boundary where interop breaks.*

Machine-Writable by Design

P16. UPG Is Machine-Written, Human-Readable. No human writes a `.upg` file by hand. The format is produced by agents, tools, and adapters; consumed by AI inference, rendering engines, and other tools. Every entity type name, property key, and edge verb is parseable from natural language without ambiguity. When choosing between a compact form and a self-describing form, choose self-describing. *Prevents: The agent becoming a source of invalid data.*

Governance

P8. The Spec Is Canonical. The UPG specification (`@unified-product-graph/core`) is the single source of truth. Applications derive from it. When an app and the spec conflict, the spec wins. *Prevents: Forking of the canonical vocabulary into app-specific dialects.*

P10. Research Before Definition. No domain is defined by guessing. Every property schema cites: what existing tools track, what practitioners need, what enables AI context, what enables queryability. *Prevents: The specification from being an opinion paper.*

P11. App Extensions Are Additive, Never Contradictory. Applications may extend entity types with app-specific properties but must never contradict a spec property's type or semantics, rename a spec property, or promote an optional field to required. *Prevents: The extension path from becoming an escape hatch that breaks interoperability.*

P12. JSDoc Is the Spec Documentation. Every property carries a JSDoc comment explaining what it is, what its values mean, and (where applicable) examples and constraints. A developer importing `@unified-product-graph/core` gets the documentation inline in their editor. *Prevents: Documentation drift.*

Cluster-Ordered Reference Index

Principle identifiers (P1–P20) reflect the chronological order in which each invariant was recognised and added to the specification. The clusters defined in §5 are a later semantic grouping applied once the full set was stable. For readers using those clusters, the principles group as follows.

Cluster	Principles
Vocabulary (How types are named)	P1, P2, P7, P9
Grammar (How types compose)	P4, P5, P18, P6
Graph Shape (When something earns being an entity)	P14, P19, P20
Properties (What entities carry)	P3, P13, P17, P15
Governance (How the spec evolves)	P8, P10, P11, P12

P16 (Machine-Written, Human-Readable) is the through-line: every other principle exists to make that commitment hold at scale.

Appendix E

MCP Tool Reference

The UPG MCP server is the primary interface between AI agents and product graphs. It runs locally against a `.upg` file or remotely against a cloud-hosted graph; the tool surface is identical, only the transport and the backend differ. An AI client speaking Model Context Protocol (Anthropic, 2024) discovers the tool set at connection time and invokes each tool by name. This appendix documents the full surface, the introspection primitive that lets agents write correctly on the first attempt, the batch semantics that collapse multi-entity composition into single calls, the session and product-context primitives that carry state across conversations, the area-scoped and change-based primitives that support collaborative workflows, and the boundary-time integrity guarantees.

Tool surface

Every MCP tool exposes a JSON-Schema signature that the AI client reads at connection time. The server advertises each tool's name, description, input schema, and output shape, so the client can drive the tool without any per-server integration code.

```
// create_node signature (as advertised to the MCP client)
{
  name: 'create_node',
  description: 'Create a new node in the product graph. Use get_entity_schema
first ' +
               'to discover what type, parent, and properties are valid.',
  inputSchema: {
    type: 'object',
    required: ['type', 'title'],
    properties: {
      type:        { type: 'string', description: 'Canonical UPG entity type'
},
      title:       { type: 'string' },
      description: { type: 'string' },
      parent_id:   { type: 'string', description: 'Parent node (must satisfy
P4 hierarchy)' },
      properties:  { type: 'object', description: 'Type-specific properties
per Layer 3 schema' },
      tags:        { type: 'array',  items: { type: 'string' } },
      status:      { type: 'string', description: 'Lifecycle phase per P15'
},
    },
  },
}
```

The complete tool set at v0.10.6 (125 tools across 12 concern groups):

Group	Tool	What it does
Context	`get_product_context`	Product metadata + digest + session in one call, the agent onboarding pill
	`get_graph_digest`	Compact triage summary: counts, chain completeness, benchmark violations
	`get_session_context`	Read structured working memory (active playbook, cursor, open hypotheses, lens)
	`update_session_context`	Write back patched session record
	`get_area_context`	Digest + session scoped to one product area
	`start`	Zero-state on-ramp: recommends the first canonical playbook + the anchor `create_node` call for an empty graph
Node reads	`get_node`	Fetch one node by id
	`get_nodes`	Batch fetch up to 50 nodes by id
	`list_nodes`	Filtered list by type, parent, status, property (paginated)
	`search_nodes`	Fuzzy text search over titles and descriptions
	`query`	Structured graph traversal via declarative filter DSL
	`get_tree`	Assemble a canonical tree pattern (OST, OKR, user, product, validation, strategy, feature areas) server-side as nested data + structural gaps
	`export_edges`	Flat enumeration of all edges (optionally filtered by type)
Node writes	`create_node`	Create a node; validates P4 hierarchy + Layer-3 property schema
	`update_node`	Patch properties and status on an existing node
	`delete_node`	Remove a node; cascades to attached edges
	`move_node`	Re-parent a node to a new parent

	`batch_create_nodes`	Create up to 50 nodes atomically; supports `$N` parent-ref chaining
	`batch_update_nodes`	Update up to 50 nodes atomically
	`batch_delete_nodes`	Delete up to 50 nodes atomically
	`batch_move_nodes`	Re-parent up to 50 nodes atomically
	`deduplicate_nodes`	Identify and merge nodes that reference the same concept
Edge writes	`create_edge`	Create a typed edge; validates source/target types against edge catalog
	`delete_edge`	Remove an edge by id
	`batch_create_edges`	Create up to 50 edges atomically
	`batch_delete_edges`	Delete up to 50 edges atomically
	`rename_edge_type`	Rename an edge type across all edges in the graph in one transactional pass
	`repair_dangling_edges`	Find and remove edges whose source or target no longer exists
Areas	`create_area`	Create a `product_area` node + parent edge in one call
	`list_product_areas`	Enumerate areas and their entity counts
	`get_area_graph`	Full node-and-edge subgraph of one area
	`assign_product_to_area`	Place an existing product under a product area (portfolio.upg membership)
	`update_area`	Edit a product area (title, priority, owner) or re-parent it via `parent_area_id`
	`remove_product_from_area`	Remove a product from a product area's membership
	`delete_area`	Delete a product area (guarded while non-empty; un-nests child areas)
	`move_product_to_area`	Move a product from one product area to another

Workspace	`init_workspace`	Create a new UPG workspace in the current directory
	`get_workspace_info`	Workspace metadata
	`create_product`	Create a new product record in the workspace
	`update_product`	Update the product header: stage, title, and metadata
	`list_local_products`	Enumerate products known to the workspace
	`switch_product`	Change the active product for the session
	`list_portfolios`	Enumerate portfolios (multi-product graphs)
	`create_cross_product_edge`	Edge between nodes in two products within the same portfolio
	`create_parity_edge`	Typed writer for the `feature_rivals_competitor_feature` parity edge (our feature vs a competitor_feature), carrying the assessment as edge metadata; routes within-graph or cross-product
	`create_classification_edge`	Typed writer for the classification edges (a node placed in a classification cell), carrying confidence and provenance as edge metadata; picks competitor vs polymorphic edge by source type, routes within-graph or cross-product
	`delete_cross_product_edge`	Delete a cross-product edge by id
	`batch_create_cross_product_edges`	Create up to 50 cross-product edges in one atomic write
	`link_area_to_audience`	Link a product area to a canonical registry persona or market segment, with primary/secondary relevance and audience role
	`attach_product_to_portfolio`	Place an existing product under a portfolio (portfolio.upg membership)
	`detach_product_from_portfolio`	Remove a product from a portfolio's membership

`list_portfolio_cross_edges`	List all cross-product edges in a portfolio
`portfolio_query`	Traverse the graph across every product in scope in one call (multi-product `query`)
`portfolio_digest`	Roll up every product's counts, health, and stage-coverage in one call (multi-product `get_graph_digest`)
`portfolio_validate`	Run `validate_graph` across every product in scope in one call (the audit counterpart to `portfolio_digest`)
`clone_structure`	Stamp the shape of one product (typed nodes + canonical edges, placeholder titles) into another, without re-authoring the skeleton
`define_canonical_entity`	Define a shared entity once in the portfolio registry (the shared-vocabulary tier) that product instances link to
`register_instance`	Link a product node to a canonical registry entity via an `instance_of` edge, enforcing the same-type constraint
`list_registry`	List the canonical shared entities in the portfolio registry, optionally with their product instances
`update_canonical_entity`	Edit a canonical registry entity (title, description, audience_role, tags, properties) without disturbing its instance edges
`batch_define_canonical_entity`	Define up to 50 canonical registry entities in one atomic call
`batch_register_instance`	Register up to 50 product instances against canonical entities in one atomic call
`promote_to_canonical`	Lift an existing product node into the registry as its canonical, optionally registering the source as the first instance

	`create_registry_edge`	Author a canonical-internal edge between two registry entities (e.g. a specification governed_by an organization), validated against the edge catalogue
	`get_organization`	Organisation that owns the workspace's portfolio (the `portfolio.upg` org singleton)
Schema	`get_entity_schema`	Valid parents, outgoing edges, property interface, lifecycle for a type
	`list_entity_types`	All entity types with maturity and domain
	`get_entity_meta`	Maturity, since, deprecated_in, type_id for one entity type
	`get_valid_children`	Which entity types may be children of a given parent type
	`resolve_edge_for_pair`	Canonical edge type for a source→target entity type pair
	`get_spec_version`	Current spec version string
Spec introspection	`list_domains`	All atomic domains with maturity
	`get_domain_guide`	Usage guide for one domain: creation sequence, property hints
	`list_frameworks`	Framework catalog (id, name, category)
	`get_framework`	Full framework definition by id
	`list_framework_categories`	All 35 framework categories
	`list_framework_structure_patterns`	All structural layout patterns (tree, matrix, funnel, ...)
	`list_edge_types`	All edge types with verb pairs and classification
	`get_edge_type`	Full edge definition by id
	`list_cross_edge_types`	Edge types that cross domain boundaries
	`list_regions`	All 11 canonical regions

	`get_region`	Full region definition: entities, anchor, shape, boundary edges
	`get_region_for_entity_type`	Which region owns a given entity type
	`list_tree_patterns`	All get_tree patterns: region, anchor, depth, gap policy
	`get_tree_pattern`	Full pattern record: child map resolved to concrete edges
	`list_lenses`	All 9 role-based lenses (product, design, engineering, growth, business, research, marketing, competitive, full)
	`get_lens`	Full lens definition: included domains, relabelled types
	`list_type_labels`	All canonical type labels with framework aliases (Rosetta Stone)
	`get_type_label`	Labels for one entity type across all frameworks
	`list_domain_rings`	The 7-ring model with domain membership
	`get_domain_ring`	Full ring definition by id
Frameworks, Playbooks & Approaches	`list_playbooks`	All 13 playbooks (filterable by region)
	`get_playbook`	Full playbook definition by id
	`list_approaches`	All 5 canonical approaches
	`get_approach`	Full approach definition by id
	`plan`	Approach: surfaces missing entities and coverage gaps
	`inspect`	Approach: anti-pattern audit + lint pass with violation list
	`prioritise`	Approach: rank a candidate set under an explicit framework
	`trace`	Approach: follow a typed path from an anchor entity
	`reflect`	Approach: structured prompts for assumptions and blind spots

	`apply_framework`	Run a framework over a set of entities: creates a `framework_exercise` node and an `includes` edge to each (the exercise model)
	`score_entity`	Record a framework's per-entity result on the exercise's `includes` edge (a MoSCoW bucket, a RICE score, a canvas slot)
Governance	`list_anti_patterns`	All registered anti-patterns with severity, lifecycle stages, and remediation
	`get_anti_pattern`	Full anti-pattern definition by id
	`get_anti_pattern_violations_for`	Reverse lookup: anti-pattern violations implicating a given entity (drill-down after `validate_graph`)
	`list_benchmarks`	All graph health benchmarks (count + ratio + relationship)
	`list_product_stages`	All product lifecycle stages
	`list_type_migrations`	All single-type migrations in the migration map
	`list_edge_migrations`	All edge type migrations
	`list_split_migrations`	All type-split migrations
	`list_lifecycles`	All entity lifecycle models
	`get_lifecycle`	Full lifecycle for one entity type
	`list_status_values`	Valid status values (+ initial / terminal) for one entity type
	`list_scales`	All assessment scales (severity, frequency, confidence, ...)
	`get_scale`	Full scale definition by id
	`validate_graph`	Run structural validation + anti-pattern audit on the whole graph
	`skill_audit`	Audit a UPG skill for source-vs-deployed integrity (symlink + body match)
Migrations	`migrate_type`	Apply a named migration on demand across all matching nodes
	`migrate_properties`	Apply property migrations graph-wide (drop, rename, lift) with no type rename; previews by default

	`migrate_status`	Rewrite legacy lifecycle status values to canonical phase ids graph-wide; previews by default
	`migrate_cross_edges`	Migrate cross-product edge types to their canonical replacements
Sync	`get_changes`	Changes since a named cursor (create/update/delete)
	`get_sync_state`	Local vs cloud head comparison
	`apply_pull_changeset`	Apply a remote changeset to the local graph
	`push_to_cloud`	Upload local changes to the cloud graph

Schema introspection

Before an AI agent creates an entity, it can ask the server *what would be valid here?* The response enumerates valid parents, permitted outgoing edges, the typed property interface, the lifecycle, and the maturity status of the type.

```
// Request: get_entity_schema({ type: 'feature' })

{
  entity_type: 'feature',
  description: 'A unit of user-visible product functionality.',
  maturity: 'stable',
  since: '0.1.0',

  valid_parents:  ['feature_area'],
  valid_children: ['epic', 'bug'],

  properties: {
    priority:     { type: 'enum', values: ['urgent', 'high', 'medium', 'low',
'none'] },
    owner:        { type: 'string', description: 'Person or team responsible
for this feature' },
    start_date:   { type: 'iso_date', description: 'When work on this feature
is targeted to start' },
    target_date:  { type: 'iso_date', description: 'When this feature should
be complete' },
    health:       { type: 'enum', values: ['on_track', 'at_risk',
'off_track', 'blocked'] },
  },

  permitted_outgoing_edges: [
    { type: 'feature_addresses_job',      target_types: ['job'] },
    { type: 'feature_tests_hypothesis',   target_types: ['hypothesis'] },
    { type: 'feature_drives_key_result',  target_types: ['key_result'] },
    // ...
  ],

  lifecycle: {
    phases: ['proposed', 'in_progress', 'shipped', 'archived'],
    initial_phase: 'proposed',
    terminal_phases: ['archived'],
  },
}
```

An LLM can produce valid UPG output on the first attempt: it does not have to memorise the ontology, it queries it. The same call answers *"can this entity be created here?"* and *"what edges can I attach once it exists?"* in one round trip.

Batch semantics with parent-ref chaining

A multi-entity structure (feature area plus features plus user stories plus acceptance criteria) composes in a single call. Parent references use `$N` notation to reference the Nth node in the same batch; the server resolves references during execution.

```
batch_create_nodes({
  nodes: [
    { type: 'feature_area', title: 'Auth',                              // $0
      parent_ref: 'root' },
    { type: 'feature',      title: 'Magic-link login',  parent_ref: '$0' },
// $1
    { type: 'feature',      title: 'OAuth (Google)',    parent_ref: '$0' },
// $2
    { type: 'user_story',   title: 'As a user, I can log in with email',
                                                        parent_ref: '$1' },
// $3
    { type: 'acceptance_criterion', title: 'Link expires after 15 min',
                                                        parent_ref: '$3' },
// $4
  ],
})
```

The server validates each creation against P4, resolves the parent chain, and returns the assigned IDs in creation order. If any creation fails validation, the entire batch is rolled back; the `.upg` file is atomic per call. The call above collapses what would be five round-trips in a naive CRUD surface into one, and the agent never has to await an ID to compose the next node in the structure.

Session context

structured memory between conversations

Most MCP servers are stateless; the MCP client carries conversation history. UPG's server adds a structured session record that persists across conversations. The record is not a chat log. It is a typed artefact an agent reads and writes as data, covering the state an agent needs to resume mid-work without re-priming.

```
// Request: get_session_context()

{
  session_id: 'ses_2026_04_23_abc',
  opened_at:  '2026-04-23T09:15:00Z',
  last_write: '2026-04-23T11:47:22Z',
  active_playbook: {
    id:       'playbook:discovery-validation-hypothesis-cycle',
    step:     3,                                  // Currently mid-playbook
    bindings: { need_id: 'n_8B1SNCNbDSs4O8Qr' } // Stop letting volume decide
the roadmap
  },
  recent_changes_cursor: 'chg_2026_04_23_00014', // For get_changes
  active_lens: 'product',                         // Agent is in the product
lens
  persona_filter: 'n_SuIk0TASeSWJRFaf',           // Scoped to Felix, solo
builder
  open_hypotheses: ['n_DMlhLiZ3a5goK1cf'],        // Retained users ask for it
≥3× more
  session_notes: 'Tagging 60 feedback items by persona × retention; Slack vs
Search call.'
}
```

`update_session_context` writes back a patched record. The next conversation, whether an hour or a month later, opens by calling `get_session_context` and reading this structure rather than re-building context from prose. The typed shape makes it safe to reason over (an enum-valued `active_playbook.step`, an id-valued `persona_filter`) where a free-form chat log would require re-interpretation. The session record is the persistent handoff between one AI conversation and the next.

Product context

the onboarding pill

A single tool call returns everything a new AI conversation needs to orient itself against a product graph. `get_product_context` composes product metadata, the graph digest (below), and the active session record in one payload.

```
// Request: get_product_context()

{
  product: {
    id:        'n_5KO9z8qsX-pYKVIb',
    title:     'Threadline',
    stage:     'growth',
    portfolio: 'prt_arkheiev',
    lens_set:  ['product', 'engineering', 'design', 'growth']
  },
  digest:  { /* see graph digest below */ },
  session: { /* see session context above */ },
  meta: {
    upg_version:        '0.8.0',
    last_modified:      '2026-04-23T11:47:22Z',
    integrity_verified: true
  }
}
```

This is the onboarding pill an agent takes as the first action in a new conversation. A few thousand tokens of structured context in, and the agent knows which product it is looking at, which playbook step was active, what chains are incomplete, and which changes have landed since the last session closed. A conversation that would otherwise spend most of its first turn re-pasting documents and re-asking clarifying questions starts productive work immediately.

Graph digest

triage, not retrieval

`get_graph_digest` returns a compact structural summary of the graph, not its contents. The digest is how an agent decides *where to focus*, which is a different kind of question from *fetch me this record*. The two read primitives serve different phases of reasoning.

```
// Request: get_graph_digest()

{
  summary: {
    total_nodes:        714,
    total_edges:       1029,
    active_domains:      25,
    most_recent_change: '2026-04-23T11:47:22Z'
  },
  by_type: {
    persona:               { count:   5, with_lifecycle_issues: 0 },
    feature:               { count: 152, with_lifecycle_issues: 3 },
    hypothesis:            { count:   9, with_lifecycle_issues: 1 },
    learning:              { count:  14, with_lifecycle_issues: 0 },
    architecture_decision: { count:  31, with_lifecycle_issues: 0 },
    // ...
  },
  chain_completeness: {
    discovery_to_delivery: { complete: 108, broken: 44, rate: 0.71 },
    evidence_to_feature:   { complete:  62, broken: 90, rate: 0.41 },
    outcome_to_feature:    { complete: 134, broken: 18, rate: 0.88 }
  },
  benchmark_hits: [
    { check:       'feature / hypothesis ratio',
      observed:    16.9, healthy_max: 10,
      severity:    'warning',
      message:     'Most features run ahead of validation.' }
  ],
  recent_changes: [
    { at: '2026-04-23T11:47Z', kind: 'create_node', type: 'feature',
      id: 'n_Pobm-G9CKhm38L5w', title: 'Cross-meeting search' }
    // ... last 20
  ]
}
```

Three kinds of signal in one response. Where the graph has density, where its spines are broken, where a benchmark flags an at-risk pattern. An agent entering a product graph for the first time reads the digest and knows whether to run the `plan` approach (chain completeness low, coverage gaps to fill), the `inspect` approach (`hypothesis` count low against `feature`, anti-pattern violations present), or a cleanup-oriented playbook (lifecycle issues on `feature`). The digest is the answer to *what would be most useful to do next*, and it arrives without loading a single entity record.

Area-scoped traversal

Products grow beyond what an agent can usefully hold in context. Area-scoped reads let the agent work inside one bounded subgraph without loading the whole thing. The scoping primitive is the `product_area` node, and the reads follow the hierarchy down from there.

```
// Request: list_product_areas({ product_id: 'n_5KO9z8qsX-pYKVIb' })

[
  { id: 'area_capture',         title: 'Capture',           node_count: 37,
edge_count: 52 },
  { id: 'area_search_recall',   title: 'Search & Recall',   node_count: 41,
edge_count: 63 },
  { id: 'area_integrations',    title: 'Integrations',      node_count: 28,
edge_count: 39 },
  { id: 'area_billing_account', title: 'Billing & Account', node_count: 18,
edge_count: 24 }
]

// Request: get_area_context({ area_id: 'area_search_recall' })

{
  area:    { id: 'area_search_recall', title: 'Search & Recall', owner:
'n_SuIk0TASeSWJRFaf' },
  digest:  { /* scoped to this area */ },
  session: { /* if the session is bound to this area */ }
}

// Request: get_area_graph({ area_id: 'area_search_recall' })

{
  area_id: 'area_search_recall',
  nodes: [ /* 41 nodes in this area */ ],
  edges: [ /* 63 edges inside or crossing the boundary */ ]
}
```

The area scope is the MCP-layer counterpart of the role-based lenses in the Presentation module (§3.3, Layer 4). Lenses filter by role; area scopes filter by topology. An agent can compose both: *give me the Auth area through the engineering lens.* A full-graph read (`list_nodes(type='*')`) is almost never what an agent wants; area scope is the everyday primitive.

Change-based collaboration

Multiple agents or humans can work against the same cloud graph at once. Lock-based contention is brittle for long-running AI sessions, so UPG exposes a change primitive: each create, update, or delete generates a change record with a monotonically increasing cursor. An agent reconciles by diffing from its last known cursor rather than by acquiring a lock.

```
// Request: get_changes({ since: 'chg_2026_04_23_00014' })

{
  since: 'chg_2026_04_23_00014',
  head:  'chg_2026_04_23_00021',
  changes: [
    { cursor: 'chg_2026_04_23_00015', kind: 'create_node',
      type:   'learning', id: 'n_mtjqbFnY0hg3fQPK',
      at:     '2026-04-23T10:12Z', by: 'agent_geordi' },
    { cursor: 'chg_2026_04_23_00016', kind: 'create_edge',
      type:   'experiment_produces_learning',
      source: 'n_DMsfelprtcx5jfQC', target: 'n_mtjqbFnY0hg3fQPK',
      at:     '2026-04-23T10:12Z', by: 'agent_geordi' }
    // ...
  ]
}

// Request: get_sync_state()

{
  local_head:        'chg_2026_04_23_00014',
  cloud_head:        'chg_2026_04_23_00021',
  direction:         'behind',
  divergent_changes: 0,        // No local changes after the last common
ancestor
  can_fast_forward:  true
}
```

`apply_pull_changeset` applies a remote changeset locally; `push_to_cloud` uploads local changes. Two agents working in different areas operate concurrently without conflict; two agents touching the same entity see divergent changes reported by `get_sync_state`, and the conflict surfaces to a human rather than being silently resolved. The full CRDT-based collaborative graph flagged as contributor-welcome in §6.2 builds on this primitive.

Integrity and migration on load and on demand

Every save computes a SHA-256 checksum in `_integrity`. Every load verifies the checksum and runs auto-migration (Appendix G) against the migration map before exposing the graph to the client, so downstream tools only ever see current canonical types. On-demand migration is also available: `migrate_type({ from: 'pain_point', to: 'need' })` walks the graph, applies the migration to any nodes still carrying the old type, and returns a count of affected nodes plus any that could not be migrated cleanly.

The boundary-time guarantee (integrity checksum plus auto-migration on load) shields agents from schema drift between tool versions. The on-demand tool lets agents drive explicit migration when a graph is being upgraded ahead of the spec.

Dual transport

local and cloud

The same tool surface runs against two backends. `@unified-product-graph/mcp-server` is the local implementation; it listens on stdio (the conventional MCP transport) and reads and writes a `.upg` file on disk. `@unified-product-graph/cloud-server` is the cloud implementation; it listens on HTTPS with OAuth and stores the graph in PostgreSQL. An MCP client switches between them by changing the server address; the tool set, JSON schemas, and response shapes are identical. A team can start on the local server with a single `.upg` file checked into git, and move to the cloud server when multi-user editing becomes necessary, without changing any agent-side code.

Appendix F

UPG Markdown Grammar

UPG Markdown (`.upg.md`, parsed by `@unified-product-graph/markdown`) is a markdown extension that embeds typed entity references inline in prose.

Reference syntax

```
## Our hypothesis for Threadline's holiday feature

Our primary user is [[persona:felix-solo-builder]]. His core job is
[[job:decide-which-feature-to-ship-before-the-holiday-window]], which
surfaces a
critical [[need:stop-letting-volume-decide-the-
roadmap|valence:pain|severity:4]].

We think [[opportunity:cluster-feedback-by-persona-x-retention]] addresses
this need, and we are testing [[hypothesis:retained-users-ask-3x-more]]
via [[experiment:tag-60-feedback-items-by-persona-and-retention]].
```

Reference format

```
[[<type>:<slug>]]                       // simplest reference
[[<type>:<slug>|<label>]]               // reference with display label
[[<type>:<slug>|<key>:<value>|…]]       // reference with inline properties
```

Resolver behaviour

- **Resolved:** `<type>:<slug>` matches a node in the graph. Resolver returns the node record; renderer may display title, lifecycle phase, or any requested property.
- **Unresolved:** `<type>:<slug>` has no matching node. Renderer shows the literal text and flags it for the author (create entity? fix slug?).
- **Stale:** `<type>:<slug>` resolved to a node whose referenced property (e.g., `status:shipped`) no longer matches. Renderer shows the current value and flags the drift.

Two-way editing

A `.upg.md` file is both a document (edit in any text editor, commit in git) and a graph composition. Edits to referenced entities propagate to every document that references them; edits to the document's prose are pure markdown and do not affect the graph. Entity renames and type migrations update every reference on load (Appendix G).

A longer document, and what a reader sees

The syntax above is the grammar. The value appears when a full document (a strategy memo, an architecture decision record, a research synthesis) uses the grammar at scale. The example below is a condensed strategy memo with inline references to persona, job, need, opportunity, hypothesis, experiment, learning, feature, and feature_area nodes, followed by how a UPG-aware reader renders it.

Source `.upg.md` file

```
# Threadline: holiday-feature decision memo

**Author:** Felix
**Date:** 2026-04-23
**Status:** Draft

## The core bet

Our primary user is [[persona:felix-solo-builder|Felix]], the solo builder
behind Threadline. His core job this month is
[[job:decide-which-feature-to-ship-before-the-holiday-window]], and it
surfaces a recurring
[[need:stop-letting-volume-decide-the-roadmap|valence:pain|severity:4]]
that the last two shipped-to-crickets features confirmed as the top
predictor of wasted build cycles.

The opportunity we are pursuing,
[[opportunity:cluster-feedback-by-persona-x-retention]], reframes the
roadmap question from *which is most-requested* to *which is most-requested
by users who stick around*. The hypothesis we ran,
[[hypothesis:retained-users-ask-3x-more|status:testing]], was tested via
[[experiment:tag-60-feedback-items-by-persona-and-retention|status:done]]
on the last six weeks of feedback in our Linear inbox.

## What we learned this quarter

The experiment produced one strong learning,
[[learning:loudest-is-churn-projection-quietest-is-retained-
pull|result_direction:positive|confidence_impact:strengthens]]:
the loudest request (Slack integration, 25 votes) came overwhelmingly
from the [[behavioral_segment:churned-within-30-days]] cohort, while the
quietest request (Cross-meeting search, 8 votes) came almost entirely
from the [[behavioral_segment:active-week-8-plus]] cohort. Calendar
back-fill split evenly across both. The Slack ask is also the surface form
of [[churn_reason:couldnt-get-team-to-adopt-threadline]], a churn-cohort
projection of the missing piece, not a retention lever.

That learning directly informs our next build:
[[feature:cross-meeting-search|status:proposed|target_date:2026-05-15]],
under the [[feature_area:search-recall]] area. Slack integration is parked
until there is an explicit team plan; Calendar back-fill is scheduled
after Search ships.

## Open questions for next quarter

- Does [[need:stop-letting-volume-decide-the-roadmap]] change valence once
  Cross-meeting search ships and we re-tag the next six weeks of feedback?
- Should we promote
  [[hypothesis:retained-users-ask-3x-more]] from a one-off check to a
```

```
  standing pre-roadmap ritual, with the persona × retention table as a
required input to every feature decision?
```

What a human reader sees when the document is parsed and rendered

The same memo, rendered by a UPG-aware reader (the first paragraph, to show the behaviour concretely):

The core bet

> Our primary user is **Felix, solo builder** `persona`, the solo builder behind Threadline. His core job this month is **Decide which feature to ship before the holiday window** `job`, and it surfaces a recurring **Stop letting feedback volume decide the roadmap when volume and value are uncorrelated** `need` ⟨pain · severity 4⟩ that the last two shipped-to-crickets features confirmed as the top predictor of wasted build cycles.

Every `[[type:slug]]` reference has resolved against the graph. The displayed label is the entity's canonical title (or the explicit display label if the reference supplied one). The small tag after each label shows the entity type. Inline property assertions (`valence:pain`, `severity:4`, `status:done`) render as compact chips and are live-checked against the current graph; if the graph has drifted, the chip shows the current value and flags the mention as stale.

What an AI agent sees when the document is parsed

The same document parsed by `@unified-product-graph/markdown` yields a CommonMark AST annotated with resolved references. A single pass over the AST returns the subgraph the memo composes:

- one `persona` (Felix), one `job`, one `need`, one `opportunity`, one `hypothesis`, one `experiment`, one `learning`, one `feature`, one `feature_area`, two `behavioral_segment` references (active week-8+ and churned within 30 days), one `churn_reason` reference
- typed edges implied by the reference context across the Discovery and Customer Feedback domains
- a set of property assertions that the memo makes about those entities (`valence:pain`, `severity:4`, `status:testing`, `status:done`, `result_direction:positive`, `confidence_impact:strengthens`, `status:proposed`, `target_date:2026-05-15`)

An agent re-reading the memo next quarter does not have to re-extract any of these entities; they already exist in the graph with stable identity. The memo is a curated view over a subgraph that persists across sessions. If the underlying entities change, every mention in every `.upg.md` file across the repository updates on load. If the memo's prose changes, the graph is untouched. This is the two-way-editing property stated in the previous subsection, demonstrated end-to-end.

Rendering implementation

The reference implementation (`@unified-product-graph/markdown`) emits a CommonMark AST annotated with resolved entity references. Downstream renderers (the UPG site, the TUI, the visual canvas, any third-party reader) consume the AST and choose their own chip, link, or inline-card presentation. Colour-coding by entity type, stale-mention highlighting, assertion-mismatch warnings, and entity hover cards are renderer concerns, not grammar concerns. The grammar guarantees that every reference is unambiguously resolvable; the renderer decides what the resolved reference looks like on screen.

Appendix G

Schema Evolution

Migration shape

```
export interface UPGTypeMigration {
  from: string                    // The old type name being retired
  to: string                      // The new canonical type name
  defaults?: Record<string, unknown>  // Default property values to set on
migrated nodes
  reason: string                  // Human-readable explanation
}

export const UPG_MIGRATIONS: Record<string, UPGTypeMigration[]> = {
  '0.1.0': [
    {
      from: 'pain_point',
      to: 'need',
      defaults: { valence: 'pain' },
      reason: 'Consolidated into neutral "need" type with valence property. '
+
              'Framework labels provide context-specific display names ' +
              '(Problem in Lean Canvas, Struggle in JTBD, etc.).',
    },
    {
      from: 'user_need',
      to: 'need',
      defaults: { valence: 'pain' },
      reason: 'Same rationale: consolidated into neutral "need" type.',
    },
    {
      from: 'kpi',
      to: 'metric',
      defaults: { designation: 'north_star' },
      reason: 'Consolidated into neutral "metric" type with designation
property.',
    },
    // ... additional migrations
  ],
}
```

Auto-migration on load

When an MCP server or client loads a `.upg` file written before a migration was introduced, the migration map auto-converts old types to canonical ones *before* the graph is exposed to the client. No data is lost. Old files remain readable; new readers

see canonical types. The migration record itself is retained in the node's `source_type` field (Appendix B), so round-trip export to a legacy tool can reconstruct the original label.

Forward and backward compatibility

- **Forward compatibility** is guaranteed for type names: a reader on an older spec version loading a file written by a newer writer will preserve unknown types as-is (the `source_type` field records what the type was, even if the current reader doesn't recognise it).
- **Backward compatibility** is guaranteed for files: a reader on a newer spec version will auto-migrate older files through the accumulated migration map. The file's `upg_version` field determines which migrations run.
- **What is not guaranteed:** semantic compatibility for multi-type structural changes. Splitting one type into two based on runtime property values, or merging two types with field reconciliation rules, is not yet supported and is the next item on the migration roadmap.

Appendix H

Framework Reference

Framework catalog

UPG ships **46 canonical frameworks** spanning 18 of its 35 framework categories. They are bundled inside `@unified-product-graph/core` as the `UPG_FRAMEWORKS` array: the famous, battle-tested frameworks that anchor the public catalog, curated for editorial confidence over breadth. Each is a declarative `UPGFramework` record (schema below), so the same canonical entity types can be read through RICE, Kano, a Business Model Canvas, or an Opportunity Solution Tree without copying the underlying data. A larger research catalog of additional definitions lives in the package's `definitions/` directory and is promoted into the canonical set as each entry is reviewed and validated.

The full, current catalog is browsable at **unifiedproductgraph.org/frameworks** (`https://unifiedproductgraph.org/frameworks`). A representative sample, one per category cluster, follows.

Representative frameworks

ID	Name	Category	Origin
`opportunity-solution-tree`	Opportunity Solution Tree	discovery	Teresa Torres
`persona-canvas`	Persona Canvas	user_understanding	Alan Cooper
`rice-scoring`	RICE Scoring	prioritization	Intercom
`kano-model`	Kano Model	prioritization	Noriaki Kano
`now-next-later`	Now-Next-Later	planning	Janna Bastow
`okr-framework`	OKR Framework	strategy	Andy Grove / John Doerr
`wardley-map`	Wardley Map	strategy	Simon Wardley
`business-model-canvas`	Business Model Canvas	business_model	Osterwalder & Pigneur
`north-star-metric`	North Star Metric	metrics	Sean Ellis / Amplitude
`pirate-metrics-aarrr`	Pirate Metrics (AARRR)	growth	Dave McClure
`build-measure-learn`	Build-Measure-Learn	validation	Eric Ries
`c4-model`	C4 Model	engineering	Simon Brown
`atomic-design`	Atomic Design	design	Brad Frost
`raci-matrix`	RACI Matrix	team_process	—
`three-horizons`	Three Horizons of Growth	portfolio	McKinsey

Framework categories (35)

The 35 categories fall into five discipline clusters; a separate set of seven structure-pattern tags describes a framework's canvas topology:

Cluster	Categories
Core Product	`prioritization` · `strategy` · `discovery` · `business_model` · `metrics` · `validation` · `planning` · `competitive`
Design & Research	`design` · `ux_research` · `user_understanding` · `research` · `accessibility` · `feedback_voc`
Engineering	`engineering` · `devops` · `security` · `qa_testing` · `ai_ml` · `agentic`
Growth & Revenue	`growth` · `marketing` · `go_to_market` · `sales` · `pricing` · `data_analytics`
Organisation & Ops	`legal_compliance` · `customer_success` · `team_process` · `program_mgmt` · `content` · `education` · `partnerships` · `localisation` · `portfolio`
Structure patterns	`tree` · `table` · `matrix` · `funnel` · `collection` · `quadrant` · `flow`

Structure patterns are not discipline categories; the seven values (`UPG_STRUCTURE_PATTERNS`) describe the visual topology of a framework's canvas. Used by `FrameworkStructureSpec.pattern`.

Framework definition schema

A `UPGFramework` is a declarative JSON object that remaps canonical UPG entity types to framework-specific vocabularies and canvas layouts. Every framework in the library conforms to this schema. Only data, no code.

Top-level type

```
export interface UPGFramework {
  id: string                                // 'business-model-canvas'
  name: string                              // 'Business Model Canvas'
  version: string                           // '1.0.0'
  description: string
  category: FrameworkCategory               // 'business_model', 'discovery',
...
  origin: FrameworkOrigin                   // attribution, year, license, URL
  tags: string[]
  slots?: FrameworkSlot[]                   // named panels/zones (data
binding)
  data: FrameworkDataSpec                   // entity types, required +
computed
                                            //   properties, constants, scoring
  structure: FrameworkStructureSpec         // topology (tree, matrix, funnel,
...)
  presentation: FrameworkPresentationSpec // layout + visual behaviour
  education: FrameworkEducation             // guidance for first-time users
  composable_with?: string[]                // frameworks this one composes
with
  extends?: string                          // parent framework this one
extends
  approach_ids?: readonly string[]          // approaches served
(plan/inspect/...)
}
```

Data layer

Declares which UPG entity types participate, their roles, required and computed properties, any framework-scaffolded constants (e.g. quadrant labels), and, for frameworks that score, a `scoring_method` that declares the inputs and formula once.

```ts
export interface FrameworkDataSpec {
  entity_types: FrameworkEntityTypeSpec[]
  required_properties: Record<string, FrameworkPropertyRequirement[]>
  computed_properties?: FrameworkComputedProperty[]    // e.g. RICE score
  constants?: FrameworkConstant[]                      // e.g. quadrant labels
  scoring_method?: FrameworkScoringMethod              // declare-once scoring
(below)
}

export interface FrameworkEntityTypeSpec {
  type: string                                 // UPG entity type
  role: string                                 // 'root' | 'item' | 'branch' |
                                               //   'leaf' | 'bucket' |
'scored_item' | ...
  min_count?: number
  max_count?: number
  auto_scaffold?: boolean
}

export interface FrameworkComputedProperty {
  property: string                             // e.g. 'rice_score'
  expression: string                           // '(reach * impact * confidence) /
effort'
  entity_type: string
  label?: string
  format?: 'number' | 'percentage' | 'currency'
}

export interface FrameworkPropertyRequirement {
  property: string                             // key on the entity's properties
object
  type: 'number' | 'string' | 'enum' | 'boolean' | 'assessment'
  required: boolean
  scope?: 'entity' | 'framework'               // UPG-595: where the property
lives
  scale_id?: string                            // assessment scale, e.g. 'reach_5'
  enum_values?: string[]                       // valid values when type is 'enum'
  default_value?: unknown
  label?: string
  description?: string
}
```

The `scope` distinction (UPG-595). Each requirement declares where its property lives. A `scope: 'entity'` requirement (the default) is an intrinsic property of the entity type and *must* exist in the entity schema; the framework-shape audit treats an entity-scoped requirement absent from the schema as a referential-integrity bug. A `scope: 'framework'` requirement is a framework-scoped scoring input the framework declares *for itself* (RICE's `reach`/`impact`/`confidence`/`effort`, MoSCoW's `moscow` bucket, Kano's functional/dysfunctional responses) — it is not asserted as an intrinsic entity property,

so the entity schema stays noise-free and the audit exempts it by design. This is the structural seam that lets a framework carry its own scoring vocabulary without polluting the canonical types.

A scoring framework (RICE, ICE, WSJF, Cost of Delay, Kano) declares its inputs and formula **once** via `scoring_method`, listing the entity types it `applies_to`. A build-time expander derives the per-type `required_properties` and `computed_properties` from it, so the source stays DRY while the public surface ships fully expanded: a consumer that ignores `scoring_method` sees the same expanded fields it always did. One declaration is why RICE can score a `feature`, an `opportunity`, or a `need`.

```
export interface FrameworkScoringMethod {
  applies_to: string[]                    // entity types this method scores
  inputs: FrameworkPropertyRequirement[]  // inputs collected, declared once
  computed?: FrameworkComputedProperty[]  // derived value(s); entity_type
filled per applies_to
}
```

Structure layer

Describes the topology: the pattern by which entities are arranged. Seven patterns are supported: `tree`, `table`, `matrix`, `funnel`, `collection`, `quadrant`, `flow`.

```
export interface FrameworkStructureSpec {
  pattern: 'tree' | 'table' | 'matrix' | 'funnel' | 'collection' | 'quadrant'
| 'flow'
  levels?: FrameworkLevel[]      // tree:    depth + label + allowed types
  slots?: MatrixSlot[]           // matrix:  grid position + spanning
  stages?: FunnelStage[]         // funnel:  ordered stages + metrics
  groups?: NamedGroup[]          // collection: named logical groups
  edge_types?: string[]          // which UPG edge types connect entities
}

// Tree example: Opportunity Solution Tree
{
  pattern: 'tree',
  levels: [
    { depth: 0, label: 'Outcome',     entity_types: ['outcome'],
      edge_from_parent: '', description: 'The desired result' },
    { depth: 1, label: 'Opportunity', entity_types: ['opportunity'],
      edge_from_parent: 'outcome_reveals_opportunity', ... },
    { depth: 2, label: 'Solution',    entity_types: ['solution'],
      edge_from_parent: 'opportunity_drives_solution', ... },
    { depth: 3, label: 'Experiment',  entity_types: ['experiment'],
      edge_from_parent: 'hypothesis_requires_experiment', ... },
  ],
}
```

Presentation layer

A discriminated union over eight layout strategies. Each layout carries its own configuration fields.

```
export type FrameworkLayout =
  | { type: 'tree';     direction: 'TB' | 'LR'; engine?: 'dagre' | 'elk' }
  | { type: 'table';    columns: TableColumn[] }
  | { type: 'matrix';   rows: number; cols: number; template?: string }
  | { type: 'funnel';   orientation: 'vertical' | 'horizontal' }
  | { type: 'kanban';   columns: string[] }
  | { type: 'quadrant'; x_axis: string; y_axis: string;
                        x_label?: string; y_label?: string }
  | { type: 'grid';     groupBy: string }
  | { type: 'flow';     direction: 'LR' | 'TB' }

export interface FrameworkPresentationSpec {
  layout: FrameworkLayout
  sort_by?: { property: string; direction: 'asc' | 'desc' }
  colour_by?: 'type' | 'status' | 'score' | 'group' | 'custom'
  card_fields?: string[]
  collapsible?: boolean
  colour_map?: Record<string, string>
}
```

Education layer

Every framework carries contextual guidance for first-time users: purpose, the question the framework answers, when to use, when not to use, and an optional step-by-step walkthrough.

```
export interface FrameworkEducation {
  purpose: string                          // one-sentence explanation
  core_question: string                    // what the framework answers
  when_to_use: string[]                    // good fit situations
  when_not_to_use: string[]                // poor fit situations
  learn_more_url?: string
  steps?: FrameworkStep[]                  // guided walkthrough
}

export interface FrameworkStep {
  order: number
  instruction: string
  property?: string                        // property this step asks to fill
  entity_type?: string                     // entity type this step focuses on
}
```

Slots

```
export interface FrameworkSlot {
  label: string                          // Display label
  role?: string                          // Semantic role, distinct from the
type
                                         //   (e.g. 'pain_reliever',
'accountable')
  entityTypeId: string                   // The UPG entity type filling this
slot
  description?: string                   // What this slot represents
}
```

`role` disambiguates slots that share an entity type. A Value Proposition Canvas fills six slots with `feature`, and a RACI fills four with `role`; the slot's `role` (`pain_reliever` vs `gain_creator`; `responsible` vs `accountable`) is what makes each addressable, since the entity type alone cannot tell them apart. It is a framework-local vocabulary, additive and optional, and distinct from the coarse structural `FrameworkEntityTypeSpec.role` above (`item`/`bucket`/`scored_item`).

```
export interface FrameworkOrigin {
  type: 'academic' | 'practitioner' | 'community' | 'custom'
  attribution?: string
  description?: string
  url?: string
  year?: number
  license?: string
}
```

The exercise model

A `UPGFramework` is a static definition. *Running* one is a separate, persisted act. An exercise is one named pass of a framework over a chosen set of entities, and its results are stored on edges, never on the entities, so the entities stay framework-agnostic (§2.5).

```
// the entity is untouched by the run
{ "id": "feat_sso", "type": "feature", "title": "SSO login" }

// one named pass of the framework
{ "id": "ex_q3", "type": "framework_exercise",
  "properties": { "framework_id": "rice-scoring" } }

// the per-entity result rides the includes edge
{ "source": "ex_q3", "target": "feat_sso",
  "type": "framework_exercise_includes_node",
  "properties": { "reach": 4, "impact": 3, "confidence": 4, "effort": 2,
                  "rice_score": 24, "slot_role": "candidate" } }
```

The `framework_exercise_includes_node` edge is one of the few edge types declared to carry properties (`carries_properties: true`, alongside the competitive `feature_rivals_competitor_feature` and the classification edges); it holds the inputs, the computed score, and the `slot_role` the entity played in this pass. Because results live on the relationship, one entity can sit in many exercises without collision, any entity type the `scoring_method.applies_to` allows can be scored, and a re-run records a fresh pass rather than overwriting the last. The SDK (`applyFramework`, `scoreEntity`), the CLI (`upg apply`, `upg score`), and the MCP server (`apply_framework`, `score_entity`) are three surfaces over this one model.

Full Business Model Canvas definition

```
{
  id: 'business-model-canvas',
  name: 'Business Model Canvas',
  version: '1.0.0',
  description:
    'Nine building blocks that describe how an organisation creates,
delivers, and captures value.',
  category: 'business_model',
  origin: {
    type: 'practitioner',
    attribution: 'Alexander Osterwalder & Yves Pigneur',
    description:
      'Published in Business Model Generation (Wiley). The most widely used
business model framework in the world.',
    url: 'https://www.strategyzer.com/business-model-canvas',
    year: 2010,
    license: 'published_methodology',
  },
  tags: ['business_model', 'matrix'],
  slots: [
    { label: 'Key Partners',           entityTypeId: 'partnership',
      description: 'Who are your key partners and suppliers?' },
    { label: 'Key Activities',         entityTypeId: 'key_activity',
      description: 'What key activities does your value prop require?' },
    { label: 'Value Propositions',     entityTypeId: 'value_proposition',
      description: 'What value do you deliver to the customer?' },
    { label: 'Customer Relationships', entityTypeId: 'customer_relationship',
      description: 'What type of relationship does each segment expect?' },
    { label: 'Customer Segments',      entityTypeId: 'market_segment',
      description: 'For whom are you creating value?' },
    { label: 'Key Resources',          entityTypeId: 'key_resource',
      description: 'What key resources does your value prop require?' },
    { label: 'Channels',               entityTypeId: 'distribution_channel',
      description: 'How do you reach your customer segments?' },
    { label: 'Cost Structure',         entityTypeId: 'cost_structure',
      description: 'What are the most important costs?' },
    { label: 'Revenue Streams',        entityTypeId: 'revenue_stream',
      description: 'For what value are customers willing to pay?' },
  ],
  data: {
    entity_types: [
      { type: 'partnership',           role: 'bucket' },
      { type: 'key_activity',          role: 'bucket' },
      { type: 'value_proposition',     role: 'bucket' },
      { type: 'customer_relationship', role: 'bucket' },
      { type: 'market_segment',        role: 'bucket' },
      { type: 'key_resource',          role: 'bucket' },
      { type: 'distribution_channel',  role: 'bucket' },
      { type: 'cost_structure',        role: 'bucket' },
```

```
  },  // ... layout and education specifications}
```

Appendix I

Sample `.upg` Files

Minimal graph

The minimal canonical form — a `$upg` header over a persona, job, and need joined by `persona_pursues_job` and `job_surfaces_need` — is shown in full in §4.1. The medium graph below is a superset of it: the same Felix → job → need spine, extended into a complete discovery-to-delivery chain.

Medium graph

the discovery spine plus the customer-feedback chain

Eighteen nodes and eighteen typed edges, drawn from the Threadline reference graph, demonstrating a complete discovery-to-delivery traceability chain *and* the parallel Customer Feedback chain that fed the decision. Two domains in one traversal: this is the compounding property the paper argues for, made concrete.

```
{
  "upg_version": "0.8.10",
  "exported_at": "2026-04-23T12:00:00Z",
  "source": { "tool": "upg-mcp-server", "tool_version": "0.8.10" },
  "product": {
    "id": "n_5KO9z8qsX-pYKVIb",
    "title": "Threadline",
    "stage": "growth"
  },
  "nodes": [
    { "id": "n_SuIk0TASeSWJRFaf", "type": "persona",
      "title": "Felix, solo builder",
      "properties": { "is_primary": true, "experience_level": "intermediate"
} },

    { "id": "n_XqLDdZOoqrmufgjd", "type": "job",
      "title": "Decide which feature to ship before the holiday window",
      "properties": {
        "job_type": "functional",
        "importance": { "value": 5, "label": "Critical" }
      } },

    { "id": "n_8B1SNCNbDSs4O8Qr", "type": "need",
      "title": "Stop letting feedback volume decide the roadmap when volume
and value are uncorrelated",
      "status": "raw",
      "properties": { "valence": "pain", "severity": { "value": 4, "label":
"Severe" } } },

    { "id": "n_t84XZhG_BeB3FFSt", "type": "opportunity",
      "title": "Cluster feedback by persona × retention bucket to surface the
request that retained users actually pull",
      "status": "identified",
      "properties": {
        "reach":     { "value": 4, "label": "high" },
        "frequency": { "value": 4, "label": "every-feature-decision" },
        "pain":      { "value": 4, "label": "high" }
      } },

    { "id": "n_Xfn1ags7Jv2udgkJ", "type": "solution",
      "title": "Cluster the last six weeks of feedback by persona × job ×
retention bucket",
      "status": "proposed" },

    { "id": "n_DMlhLiZ3a5goK1cf", "type": "hypothesis",
      "title": "One of the three requests is asked by retained week-8+ users
at ≥3× the rate of churned users",
      "status": "untested",
      "properties": { "falsifiable": true, "confidence_prior": 0.6 } },
```

```
      "title": "Tag 60 feedback items in Linear by persona, job-pursued, and
retention bucket; recompute volume table",      "status": "done",
"properties": { "duration_days": 1, "method": "manual_tag_then_pivot",
"owner": "Felix" } },    { "id": "n_mtjqbFnY0hg3fQPK", "type": "learning",
"title": "The loudest request is a churn-cohort projection; the quietest
request is a retained-cohort pull" },    { "id": "n_Pobm-G9CKhm38L5w",
"type": "feature",      "title": "Cross-meeting search",      "status":
"proposed" },    { "id": "n_eazzaR_5OTzrUa6h", "type": "feature_area",
"title": "Search & Recall",      "status": "planned" },    { "id":
"n_kTgncoZygHPxoo_v", "type": "feature_request",      "title": "Slack
integration: push action items to a #meetings channel",      "status":
"under_review",      "properties": { "vote_count": 25, "signal_sentiment":
"mixed" } },    { "id": "n__zNL5qCsBM0BP3hL", "type": "feature_request",
"title": "Cross-meeting search: find decisions across past meetings",
"status": "under_review",      "properties": { "vote_count": 8,
"signal_sentiment": "positive" } },    { "id": "n_kYKW9gqLeQJ8qVgv", "type":
"customer_feedback",      "title": "Slack request verbatim, Team Lead,
churned",      "properties": { "feedback_type": "review", "sentiment":
"negative" } },    { "id": "n_RQOfUYFvqmULpGcH", "type": "customer_feedback",
"title": "Cross-meeting search request verbatim, IC Researcher, week-12
retained",      "properties": { "feedback_type": "interview", "sentiment":
"positive" } },    { "id": "n_GpnnfzONZPWNeydY", "type":
"behavioral_segment",      "title": "Churned within 30 days",
"properties": { "size": 113, "segment_type": "behavioral" } },    { "id":
"n_Q9KjiuhCDHu6kaEl", "type": "behavioral_segment",      "title": "Active
week-8+",      "properties": { "size": 47, "segment_type": "behavioral" } },
{ "id": "n_KEjat6PPvUUhavwH", "type": "persona",      "title": "Team Lead
(Threadline user)" },    { "id": "n__xXl1ITTYamOrAAE", "type": "persona",
"title": "IC Researcher (Threadline user)" }  ],  "edges": [    { "id": "e1",
"source": "n_SuIk0TASeSWJRFaf", "target": "n_XqLDdZOoqrmufgjd",      "type":
"persona_pursues_job",                "mapping_confidence": "high" },    {
"id": "e2",  "source": "n_XqLDdZOoqrmufgjd", "target": "n_8B1SNCNbDSs4O8Qr",
"type": "job_surfaces_need",                  "mapping_confidence": "high" },
{ "id": "e3",  "source": "n_t84XZhG_BeB3FFSt", "target":
"n_8B1SNCNbDSs4O8Qr",      "type": "opportunity_addresses_need",
"mapping_confidence": "high" },    { "id": "e4",  "source":
"n_t84XZhG_BeB3FFSt", "target": "n_Xfn1ags7Jv2udgkJ",      "type":
"opportunity_drives_solution",        "mapping_confidence": "high" },    {
"id": "e5",  "source": "n_Xfn1ags7Jv2udgkJ", "target": "n_DMlhLiZ3a5goK1cf",
"type": "solution_proposes_hypothesis",       "mapping_confidence": "high" },
{ "id": "e6",  "source": "n_DMlhLiZ3a5goK1cf", "target":
"n_DMsfelprtcx5jfQC",      "type": "hypothesis_requires_experiment",
"mapping_confidence": "high" },    { "id": "e7",  "source":
"n_DMsfelprtcx5jfQC", "target": "n_mtjqbFnY0hg3fQPK",      "type":
"experiment_produces_learning",       "mapping_confidence": "high" },    {
"id": "e8",  "source": "n_mtjqbFnY0hg3fQPK", "target": "n_t84XZhG_BeB3FFSt",
"type": "learning_validates_opportunity",     "mapping_confidence": "high" },
{ "id": "e9",  "source": "n_mtjqbFnY0hg3fQPK", "target": "n_Pobm-
G9CKhm38L5w",      "type": "learning_informs_feature",
```

```
"feature_request_in_feature_area",    "mapping_confidence": "high" },    {
"id": "e11", "source": "n_kYKW9gqLeQJ8qVgv", "target": "n_kTgncoZygHPxoo_v",
"type": "customer_feedback_becomes_feature_request", "mapping_confidence":
"high" },    { "id": "e12", "source": "n_RQOfUYFvqmULpGcH", "target":
"n__zNL5qCsBM0BP3hL",      "type":
"customer_feedback_becomes_feature_request", "mapping_confidence": "high" },
{ "id": "e13", "source": "n_kTgncoZygHPxoo_v", "target":
"n_GpnnfzONZPWNeydY",      "type": "feature_request_from_behavioral_segment",
"mapping_confidence": "high" },    { "id": "e14", "source":
"n__zNL5qCsBM0BP3hL", "target": "n_Q9KjiuhCDHu6kaEl",      "type":
"feature_request_from_behavioral_segment",   "mapping_confidence": "high" },
{ "id": "e15", "source": "n_kTgncoZygHPxoo_v", "target":
"n_t84XZhG_BeB3FFSt",      "type": "feature_request_creates_opportunity",
"mapping_confidence": "high" },    { "id": "e16", "source":
"n__zNL5qCsBM0BP3hL", "target": "n_t84XZhG_BeB3FFSt",      "type":
"feature_request_creates_opportunity",       "mapping_confidence": "high" },
{ "id": "e17", "source": "n_GpnnfzONZPWNeydY", "target":
"n_KEjat6PPvUUhavwH",      "type": "behavioral_segment_maps_to_persona",
"mapping_confidence": "high" },    { "id": "e18", "source":
"n_Q9KjiuhCDHu6kaEl", "target": "n__xXl1ITTYamOrAAE",      "type":
"behavioral_segment_maps_to_persona",        "mapping_confidence": "high" }
],  "_integrity": {    "checksum": "8e42b7f10c9d5a6b1f3e4c2d7a8b9e05",
"verified_at": "2026-04-23T12:00:00Z",    "verified_by": "upg-mcp-
server@0.8.10"  }}
```


This 18-node graph answers two questions in two traversals over the same data.

Why does the Cross-meeting search feature exist? `n_Pobm-G9CKhm38L5w` ⟵ `n_mtjqbFnY0hg3fQPK` ⟵ `n_DMsfelprtcx5jfQC` ⟵ `n_DMlhLiZ3a5goK1cf` ⟵ `n_Xfn1ags7Jv2udgkJ` ⟵ `n_t84XZhG_BeB3FFSt` ⟵ `n_8B1SNCNbDSs4O8Qr` ⟵ `n_XqLDdZOoqrmufgjd` ⟵ `n_SuIk0TASeSWJRFaf`. The discovery spine reads end to end from the persona to the shipped-decision feature.

Why was the loudest request the wrong one? `n_kTgncoZygHPxoo_v` ⟶ `n_GpnnfzONZPWNeydY` (Slack feature_request from the *Churned within 30 days* segment) versus `n__zNL5qCsBM0BP3hL` ⟶ `n_Q9KjiuhCDHu6kaEl` (Cross-meeting search from the *Active week-8+* segment). The customer-feedback chain crosses into the Discovery domain through `feature_request_creates_opportunity` and supplies the evidence the spine validates. Every edge is typed; every verb reads both directions; every node has a stable ID that will survive the next AI session.

The self-hosting corpus

As of the date of this draft, the reference product graph for Entopo is maintained as a living `.upg` workspace. It is built incrementally across AI-assisted sessions, with agents operating through UPG's role-based lenses on a single shared graph. The population

skews toward the build-side entity types (features, screens, services, architecture decisions), which is what the authors would expect from a product past the discovery phase. The graph is kept open so anyone can inspect how UPG is used on a real product, rather than read a stylised example.

Appendix J

The 11 Canonical Regions

A **region** is a super-domain rollup, a coherent slice of product knowledge across multiple atomic domains, unified by a shared design problem and an anchor entity. Regions are Layer 4 read-time constructs; they are never written to the `.upg` file. A node's region membership is derived from its atomic domain assignment.

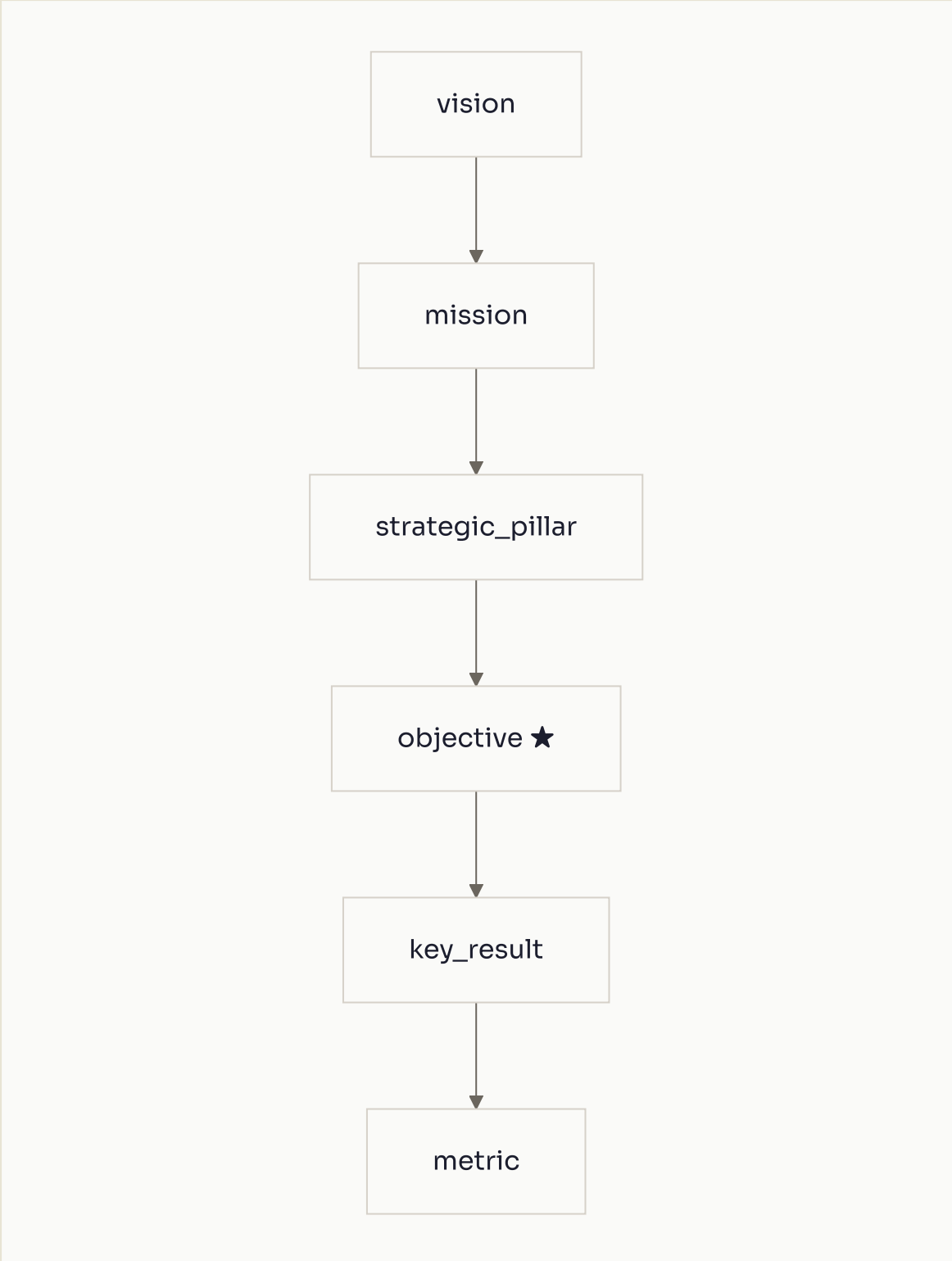
Each region carries four defining pieces: the **anchor entity** (the type where the region's design problem concentrates most sharply), a **shape archetype** (the graph topology that characterises how entities in the region connect), the **composed atomic domains** (the atomic domains whose entities the region rolls up), and **boundary edges** (the typed edges that connect this region to its neighbours).

#	Region	Anchor	Shape	Composes
1	Strategy & Outcomes	`objective`	Cascade	strategy
2	Users & Needs	`persona`	Convergent hub	user
3	Discovery, Research & Validation	`opportunity`	Cyclic processing graph	discovery · validation · user_research · feedback
4	Market & Competitive	`competitor`	DAG	market_intelligence
5	Experience, Design & Brand	`user_journey`	Event-driven collage	ux_design · design_system · content · brand
6	Product & Delivery	`feature`	Layered DAG	product_spec · feedback
7	Engineering & Platform	`service`	Layered mesh	engineering · devops · testing · security · ai_ml · agentic · data · automation
8	Business, GTM & Growth	`value_proposition`	Cyclic value-exchange graph	business_model · pricing · gtm · growth · marketing · sales · ecosystem
9	Analytics & Data	`metric`	DAG	data_analytics
10	Operations & Quality	`incident`	Event-driven collage	devops · testing · security · accessibility · customer_success · team_org · legal · localisation · program_mgmt
11	Foundations	`specification`	Polymorphic-target	foundations

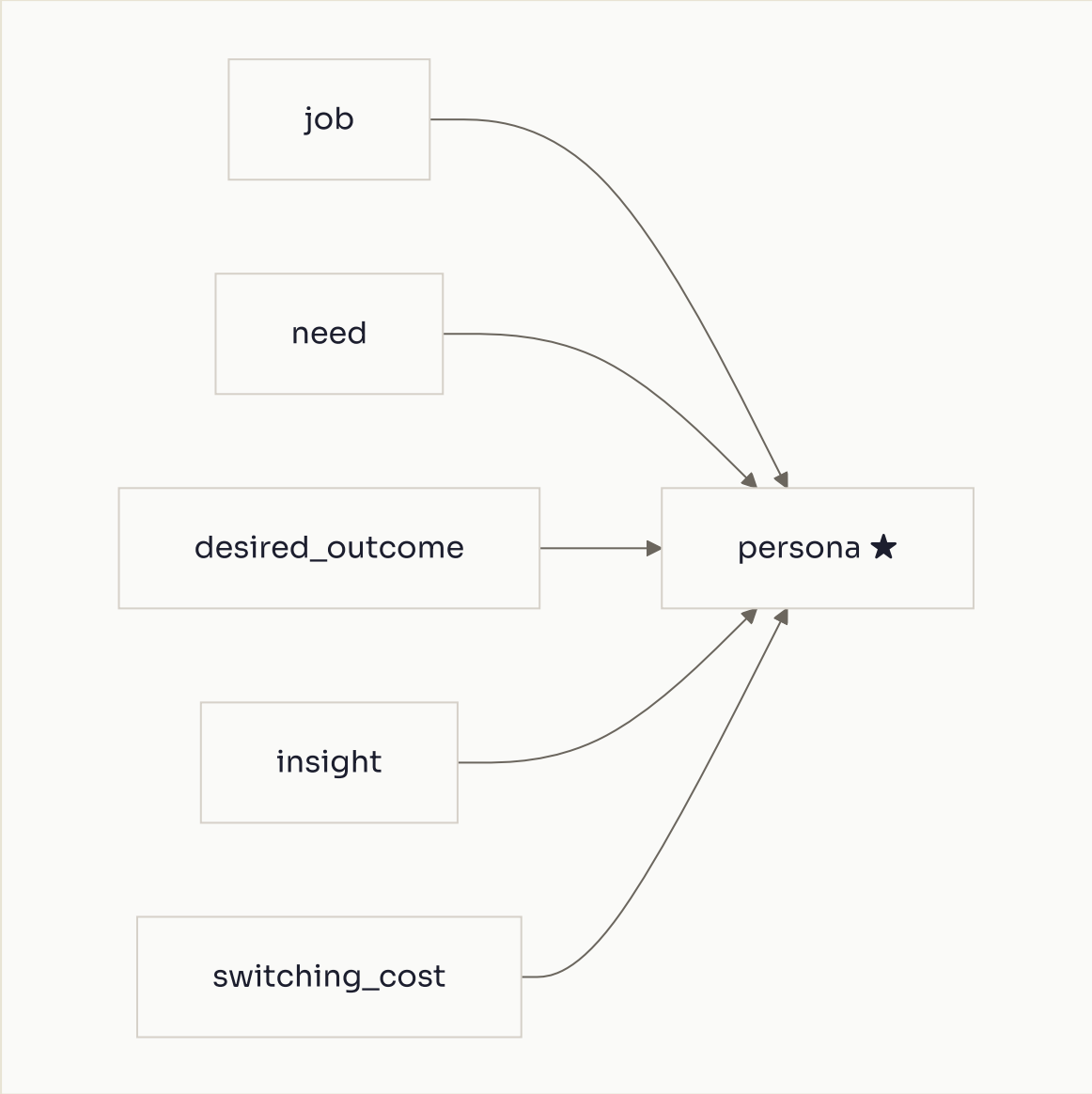
Shape archetypes illustrated

Each shape archetype describes the dominant topology of a region's internal edges, how entities connect to each other within that region. The diagrams below show the structural pattern for each shape using representative entity types.

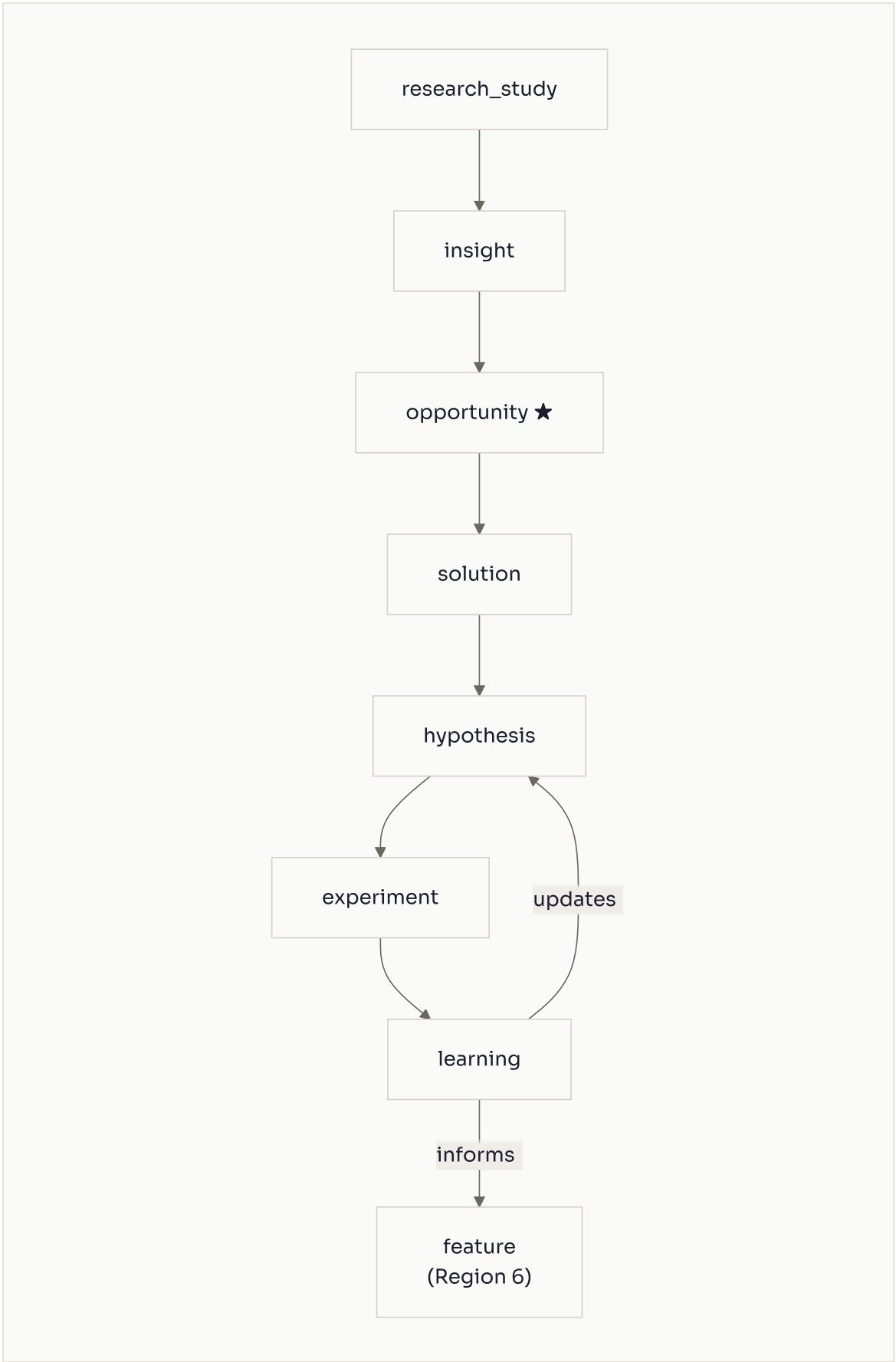
Cascade: Strategy & Outcomes. Aspiration flows downward through direction to measurement. A strict top-down hierarchy with no cycles.



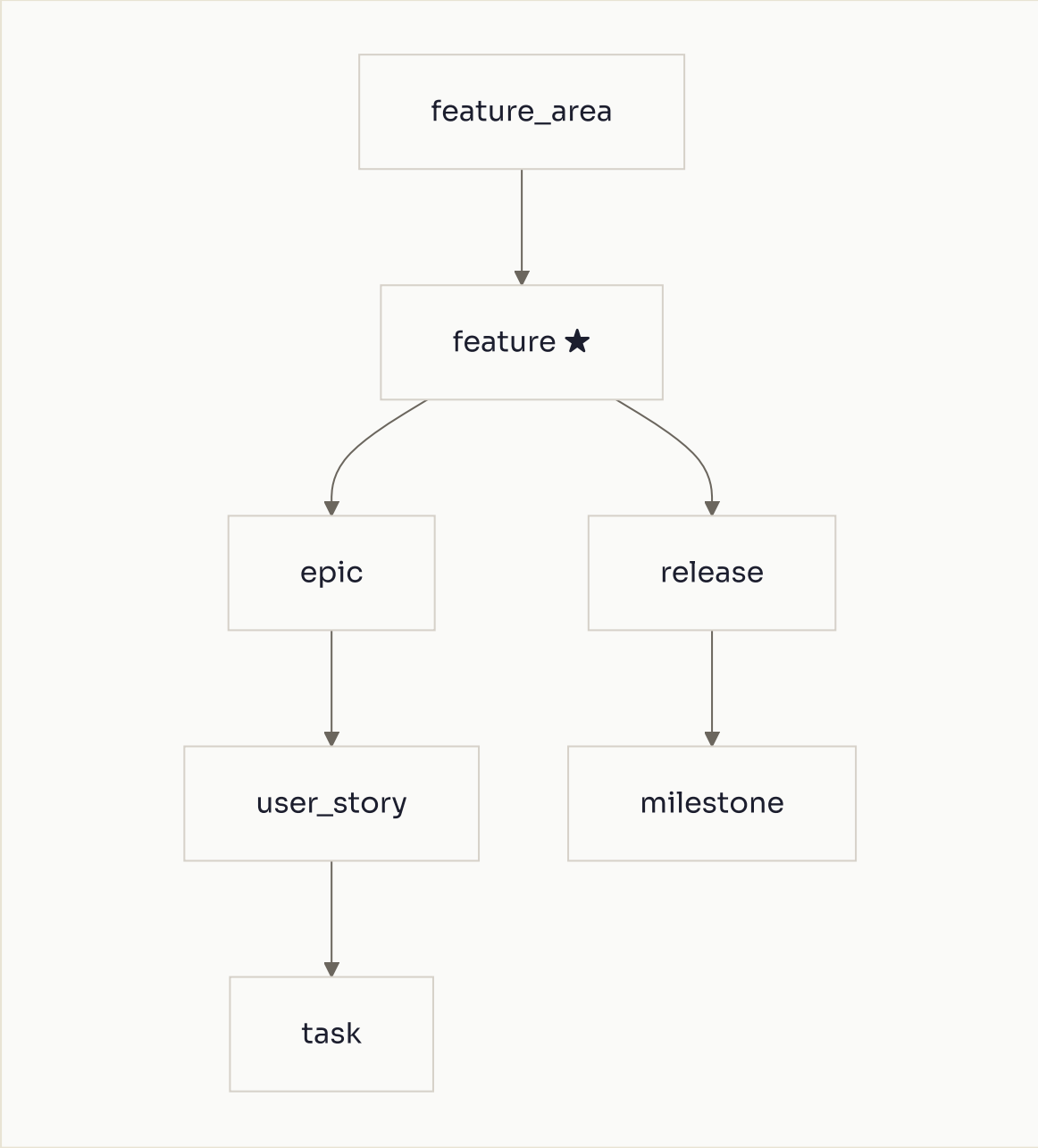
Convergent hub: Users & Needs. Many entity types feed into one gravitational centre. Persona is the hub; everything a team learns about users traces back to it.



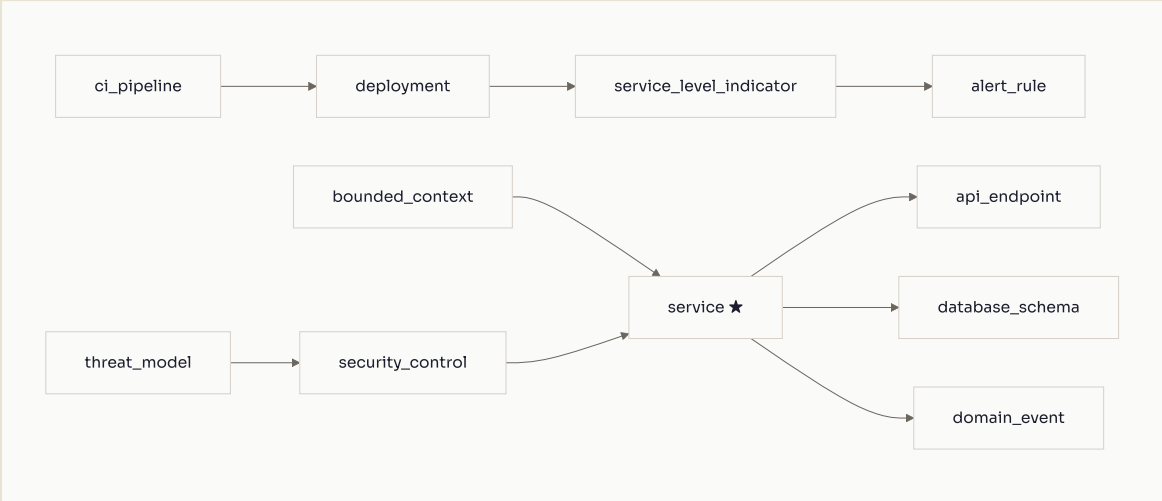
Cyclic processing graph: Discovery, Research & Validation. A closed loop where learnings feed back into hypotheses, driving continuous iteration. The cycle is the point.


research_study
insight
opportunity ★
solution
hypothesis
experiment
updates
learning
informs
feature
(Region 6)

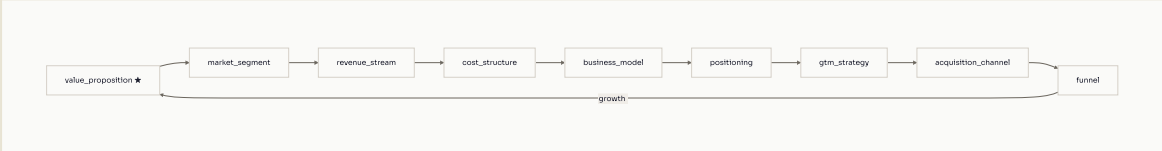
Layered DAG: Product & Delivery. A directed acyclic graph with clear levels of decomposition. No cycles; each level refines the one above.



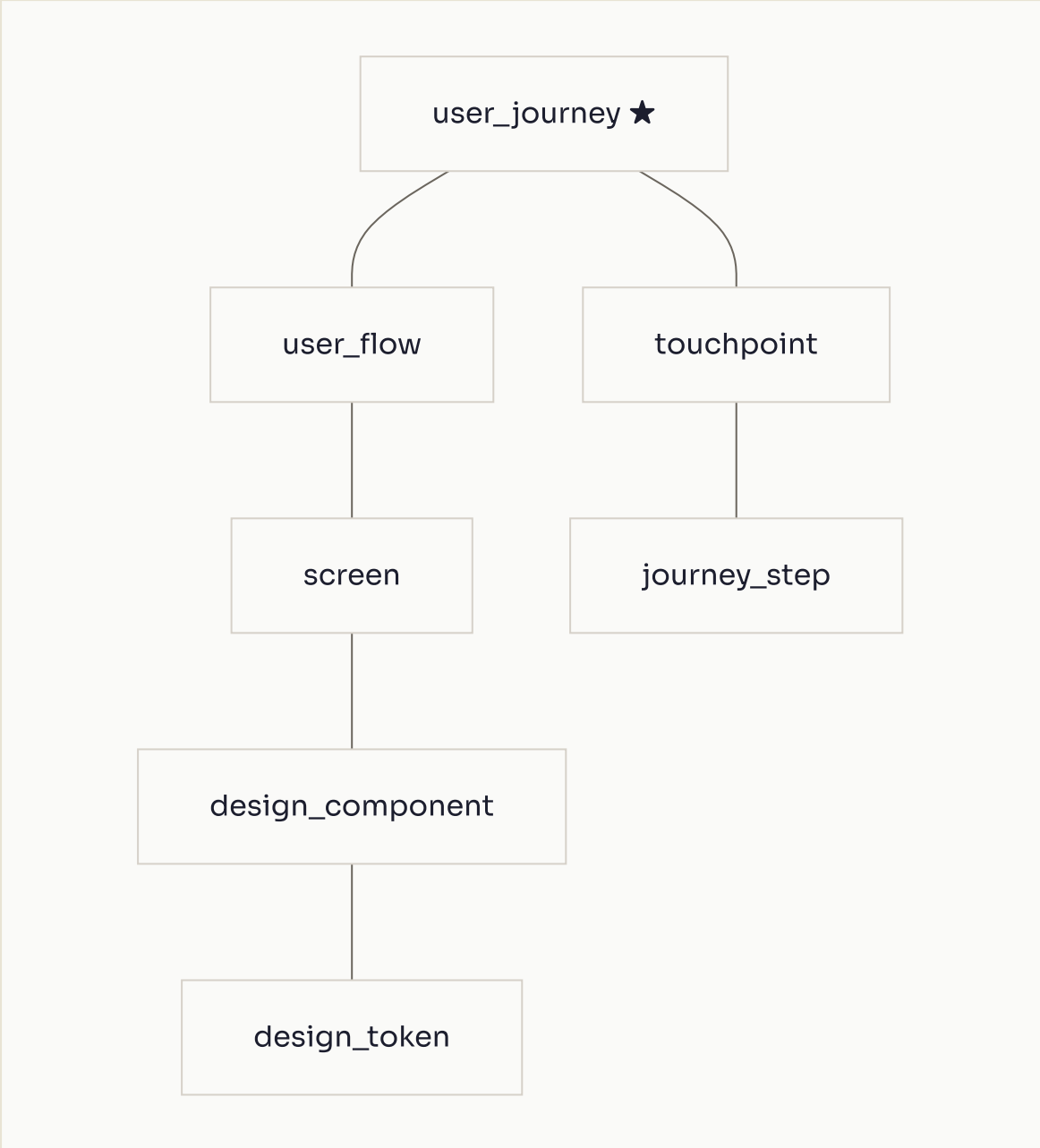
Layered mesh: Engineering & Platform. Interconnected but not cyclic. Multiple concern layers (architecture, data, build, deploy, monitor, security) cross-reference each other through typed edges.



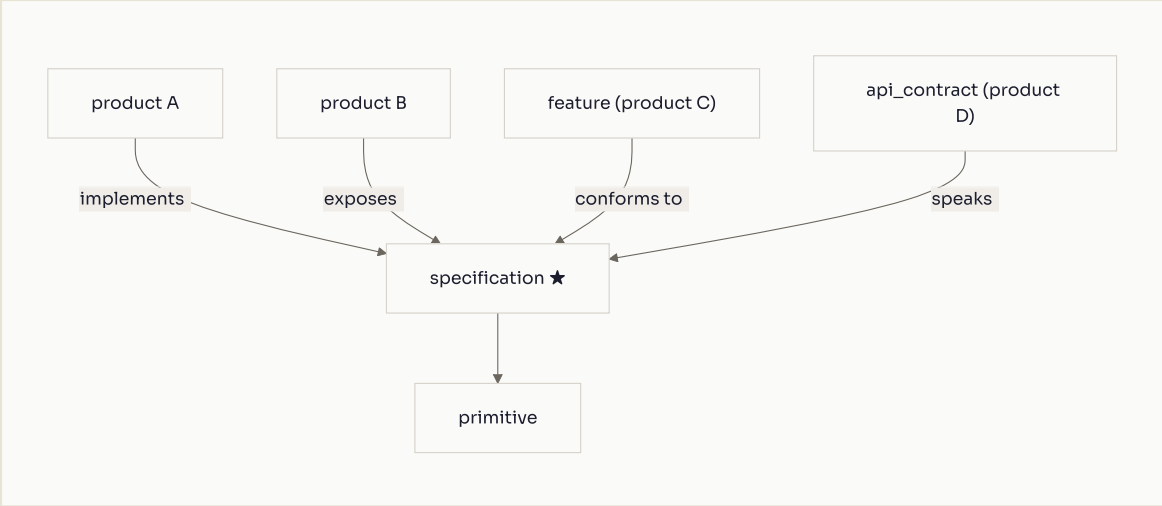
Cyclic value-exchange graph: Business, GTM & Growth. A loop where each element enables the next, and growth feeds back into the value proposition that started it.



Event-driven collage: Experience & Design / Operations & Quality. No single root; entities co-exist and cross-reference laterally. The shape is a network, not a hierarchy or a loop.



Polymorphic-target: Foundations (and the read-level shape of Analytics & Data). No internal hierarchy; many heterogeneous sources point inward at one small set of shared targets. Unlike a convergent hub, the sources live in *different products*: the anchor is a fan-in across the portfolio. In Foundations, many products implement, expose, and conform to the same `specification`.



★ = *anchor entity for that region. Anchor entities are defined in the region table above.*

Region profiles

Region 1: Strategy & Outcomes. The cascade shape reflects the region's mental model: aspiration flows downward through direction, strategic bets, measurable key results, and finally proof in metrics. `objective` is the anchor because it is where planning language (`desired outcome`) meets measurement (`key_result`, `metric`). The accountability question of strategy, *is this objective actually being met?*, resolves through the objective node. Cross-region exports connect to Discovery (outcomes reveal opportunities) and Product Delivery (outcomes are delivered by features).

Region 2: Users & Needs. The convergent hub shape reflects that persona is the gravitational centre of the spec: 26 inbound cross-edges from domains across the graph flow into `persona`. Everything a product team learns about users eventually resolves to a persona, a job that persona is pursuing, a need that job surfaces, and a desired outcome the persona would accept as *done*. The region composes only the `user` atomic domain, keeping it the spec's most focused region.

Region 3: Discovery, Research & Validation. A cyclic processing graph: research study produces insights, insights inform opportunities, opportunities drive solutions, solutions propose hypotheses, hypotheses require experiments, experiments produce learnings, and learnings update hypotheses, closing the loop. `opportunity` is the anchor because the discovery-to-delivery spine's first branching point is the opportunity a team decides to pursue. This region has the most boundary edges of any, connecting inbound from Users & Needs and Strategy, outbound into Product Delivery and Engineering.

Region 4: Market & Competitive. A directed acyclic graph from market landscape to competitive differentiation. `competitor` is the anchor: the entity type that carries external threat signal most directly. Competitor features connect to features (competitive differentiation), market trends connect to opportunities (external drivers), and segments connect to personas (market-as-users). As of 0.10.0 the region also carries `competitor_signal` — a dated competitor move that maps onto a feature or surfaces an opportunity (§3.11) — and the parity edge `feature_rivals_competitor_feature`, which makes *where are we behind* a single traversal.

Region 5: Experience, Design & Brand. An event-driven collage: journeys, flows, screens, interactions, design components, tokens, brand assets coexist without a single dominant hierarchy. `user_journey` is the anchor because it is the entity that narrates the user experience end-to-end. Design tokens underpin components underpin screens underpin flows underpin journeys, a dependency chain that runs bottom-up, unlike the top-down cascade of Strategy.

Region 6: Product & Delivery. A layered DAG: feature areas contain features, features are refined by epics and user stories, stories decompose to tasks, tasks aggregate into releases and milestones. `feature` is the anchor: the unit of user-visible product functionality, and the end point of the discovery-to-delivery spine. Cross-edges into Engineering (features realised by services), Strategy (features deliver key results), and Users (features address jobs and needs).

Region 7: Engineering & Platform. The largest region by entity count. A layered mesh: bounded contexts own services, services expose APIs, APIs consume data models, data flows through pipelines, deployments ship changes, monitors watch SLIs, alerts fire on SLOs. `service` is the anchor because it is the canonical engineering abstraction visible at product level. Eight atomic domains compose this region: the full build/deploy/monitor/security/AI stack.

Region 8: Business, GTM & Growth. A cyclic value-exchange graph: value proposition addresses segment needs, segments generate revenue through pricing, revenue funds cost structure, cost structure constrains the business model, business model informs positioning, positioning shapes the GTM strategy, GTM generates acquisition, acquisition feeds the growth funnel, growth loops create retention. `value_proposition` is the anchor: the claim the product makes about why someone should pay for it.

Region 9: Analytics & Data. A DAG: data sources feed event schemas, events define metrics, metrics compose into dashboards, dashboards inform decisions. `metric` is the anchor because analytics ultimately reduces to the question *"what number are we trying to move?"* This region is the measurement plane shared across all other regions: metrics appear in Strategy (key results), Growth (funnel metrics), and Engineering (SLIs).

Region 10: Operations & Quality. An event-driven collage spanning DevOps, testing, security, accessibility, customer success, team org, legal, localisation, and program management. `incident` is the anchor: the entity type that surfaces operational failure most directly. The region is deliberately the catch-all for everything that keeps the product running, safe, and compliant after it ships.

Region 11: Foundations. A polymorphic-target shape, and the region model's home for the portfolio tier (§3.10). `specification` is the anchor: the governed rulebook many products implement, expose, and conform to, and the type by which primitives are defined. It is the highest-inbound canonical of the region, but unlike every other region its inbound edges are *cross-product* (`product_implements_specification`, `product_exposes_specification`, `feature_conforms_to_specification`, `api_contract_speaks_specification`) rather than within-product boundary edges, which is why the region carries no boundary edges of its own. The region composes a single atomic domain (`foundations`) of two types, `specification` and `primitive`; its internal edges (`primitive_defined_by_specification`, `specification_extends_specification`, `primitive_composes_primitive`, `specification_governed_by_organization`) are catalog edges between canonical registry nodes. Foundations is where a portfolio's shared technical substrate, a query language, a content format, a protocol, becomes a first-class object rather than a string scattered across product graphs.

Boundary edges

Regions connect through a set of typed boundary edges, cross-domain edges whose source lives in one region and whose target lives in another. Every boundary edge is a registered entry in the edge catalog with a forward and reverse verb. The boundary between regions is where UPG's value is most visible: a `feature` in Region 6 is connected by a `feature_drives_key_result` edge to an `objective` in Region 1, by `feature_tests_hypothesis` to a `hypothesis` in Region 3, and by `feature_realised_by_service` to a `service` in Region 7. No single tool today holds all three relationships on one screen. In a UPG graph, they are three traversals of the same structure.

Appendix K

The 5 Canonical Approaches

Five canonical approaches cover the full space of agent-graph interaction modes. The catalog is closed at v0.8.0: each approach is a named MCP tool invocable by its id as a bare verb.

Plan

MCP tool: `plan` · **Question answered:** *What should I build next?*

Cartographic framing. Walking the coastline of a region and noting where the contour is incomplete: not deciding a strategy, but mapping what is missing against canonical expectations. The approach surveys entity coverage per region and surfaces the entities a healthy region carries that this graph does not yet have.

Signature: `({ region?: UPGRegionId }) ⟶ { missing_entities, coverage_score }`

Frameworks inside Plan: `now-next-later` · `moscow` · `wardley-map` · `okr-framework` · `three-horizons`. Frameworks inside Plan answer *in what order* to fill the gaps: Now/Next/Later sequences them by horizon, MoSCoW by must/should/could priority, Wardley by evolution stage.

Typical invocation. An agent opening a new product graph calls `get_graph_digest`, sees chain completeness at 41% on `evidence_to_feature`, then calls `plan({ region: 'discovery_research_validation' })` to get the specific entity types missing from the discovery chain. The response drives the next `batch_create_nodes` call.

Inspect

MCP tool: `inspect` · **Question answered:** *What's broken?*

Cartographic framing. Surveying the coastline for hazards before approach: the violations are the rocks marked on the chart. The approach runs anti-pattern audits and lint passes against the graph or a scoped region, returning a structured violation list with severity, target entity id, description, and a remediation hint.

Signature: `({ region?: UPGRegionId, entities?: entity_ids[] }) ⟶ { violations: [{ severity, entity_id, description, remediation }] }`

Frameworks inside Inspect: `design-critique` · `ui-audit` · `wcag-audit-checklist` · `risk-assessment-matrix` · `blameless-postmortem`. Frameworks inside Inspect are the named audit catalogues: each provides a structured set of checks for a different concern area.

Anti-pattern library. Inspect draws on the registered `UPG_ANTI_PATTERNS` catalog: 12 named anti-patterns that detect broken or missing narrative chains — personas without jobs, opportunities without needs, features without hypotheses, untested-hypothesis pile-up, experiments run without learning, objectives without key results, roadmap features with no outcome link, and single-domain graphs among them. Each carries a machine-checkable `structured_condition` (so detection is data, not code), a `severity` (`high` / `medium`), the lifecycle `stages` it applies to, a `why_it_matters` rationale, a `remediation` hint, and a `source` attribution.

Prioritise

MCP tool: `prioritise` · **Question answered:** *What's most important?*

Cartographic framing. The caller supplies a candidate set and a framework; the approach computes the order of arrival from a chosen vantage. Prioritisation without a declared scoring framework is incoherent: the approach enforces that `framework_id` is always explicit, making the scoring methodology auditable in the graph.

Signature: `({ candidates: entity_ids[], framework_id }) ⟶ { ranked: [{ entity_id, score, rationale }], framework_used }`

Frameworks inside Prioritise: `rice-scoring` · `ice-scoring` · `kano-model` · `cost-of-delay` · `moscow` · `wsjf`. Each framework weights the same candidate set differently: RICE by Reach × Impact × Confidence ÷ Effort, ICE by Impact × Confidence × Ease, Kano by functional vs. delight response, Cost of Delay by value at risk per unit of delay.

Integration with graph data. For candidates with `properties.impact`, `properties.reach`, `properties.confidence`, `properties.effort` already recorded (from prior creation sessions), the approach uses those property values directly in the scoring formula, making the prioritisation reproducible and auditable from the graph alone.

Trace

MCP tool: `trace` · **Question answered:** *Walk a meaningful path through what exists.*

Cartographic framing. Tracing a route across charted terrain: the anchor is the departure point, the path array is the heading sequence, and canonical edges are the roads. The approach follows a typed path from an anchor entity through named entity types, resolving the canonical edge for each consecutive pair via `resolve_edge_for_pair`. An optional `edges_override` array selects non-canonical edges per hop when a pair has multiple resolutions.

Signature: `({ anchor: entity_id, path: UPGEntityType[], edges_override?: (string | null)[] }) → { trail: [{ depth, entity_id, edge_type_in }], reached: entity_id[] }`

Frameworks inside Trace: `opportunity-solution-tree` · `strategic-cascade` · `metrics-tree` · `user-journey-map` · `impact-map` · `data-lineage-map`. Each framework defines a named traversal pattern: OST walks `outcome → opportunity → solution → experiment`, Strategic Cascade walks `vision → mission → pillar → theme → objective → key_result`, Metrics Tree walks `north_star → input_metric → health_metric`.

The discovery spine as a trace. The canonical discovery-to-delivery spine from §3.6 is a `trace` invocation: `{ anchor: persona_id, path: ['job', 'need', 'opportunity', 'solution', 'hypothesis', 'experiment', 'learning', 'feature'] }`. The approach resolves the canonical edge for each hop in the path array and returns every entity reached at each depth, along with which edge type connected it.

Reflect

MCP tool: `reflect` · **Question answered:** *What should I be questioning?*

Cartographic framing. Before approaching the coastline, asking which features of the chart have not actually been verified: the prompts mark the parts of the map that may be conjecture. The approach surfaces structured prompts that expose assumptions, alternatives, blind spots, and load-bearing claims in the graph or a scoped region. An optional `mode` parameter narrows to a specific reflection type.

Signature: `({ scope?: UPGRegionId | entity_id | null, mode?: 'assumptions' | 'alternatives' | 'blind-spots' | 'load-bearing' }) → { prompts: [{ kind, question, target_entities? }] }`

Frameworks inside Reflect:

Framework	Mode	What it surfaces
`five-whys`	assumptions	Causal chain from a symptom back to a root cause
`pre-mortem`	blind-spots	Imagined failure of the current plan and its causes
`red-team`	alternatives	Strongest case against the current direction
`devils-advocate`	alternatives	Counter-argument to each load-bearing assumption
`second-order-thinking`	load-bearing	Downstream consequences of each current decision

Integration with hypothesis entities. When `scope` is a region that contains `hypothesis` entities, Reflect automatically surfaces any hypotheses whose `status` is still `untested`: the highest-leverage load-bearing claims in the graph. These appear as `kind: 'load-bearing'` prompts with the hypothesis id as `target_entities`.

Appendix L

The 13 Playbook Catalog

13 playbooks across 11 regions. The W1 invariant holds: exactly one canonical playbook per region, audited by CI. Two additional specialised playbooks provide alternative entry paths into the Business, GTM & Growth region.

Region	#	ID	Name	Type	Framework anchor
1 Strategy & Outcomes	C	`playbook:strategy-outcomes`	Strategy & Outcomes	canonical	—
2 Users & Needs	C	`playbook:users-needs`	Users & Needs	canonical	—
3 Discovery, Research & Validation	C	`playbook:discovery-research-validation`	Discovery, Research & Validation	canonical	—
4 Market & Competitive	C	`playbook:market-competitive`	Market & Competitive	canonical	—
5 Experience, Design & Brand	C	`playbook:experience-design-brand`	Experience, Design & Brand	canonical	—
6 Product & Delivery	C	`playbook:product-delivery`	Product Delivery	canonical	—
7 Engineering & Platform	C	`playbook:engineering-platform`	Engineering & Platform	canonical	—
8 Business, GTM & Growth	C	`playbook:business-gtm-growth`	Business, GTM & Growth	canonical	—
	S	`playbook:business-growth-metric-driven`	Metric-Driven Growth	specialised	—
	S	`playbook:business-marketing-audience-first`	Audience-First Marketing	specialised	—
9 Analytics & Data	C	`playbook:analytics-data`	Analytics & Data	canonical	—
10 Operations & Quality	C	`playbook:operations-quality`	Operations & Quality	canonical	—
11 Foundations	C	`playbook:foundations`	Foundations	canonical	—

C = canonical (one per region, W1 invariant). S = specialised.

Creation sequence depth

Playbooks range from short two-step sequences (Foundations) to eight-step multi-domain sequences (the full canonical playbooks for Discovery, Experience & Design, and Engineering & Platform). The `creation_sequence` field on each playbook is the ordered list of `Step` records: each step carrying a `kind` (domain_guide, framework, entity_sequence, or sub_sequence), an ordered step number, a phase name, a prompt hint, and optionally a `next_sequence_on_gap` that chains to a follow-up playbook when the intelligence module detects a structural gap at that step.

The eight-step Engineering & Platform playbook illustrates the full depth: Architecture (bounded contexts, decisions, services) → Services & APIs (endpoints, contracts, integrations) → Data (schemas, events, pipelines) → Build (features, epics, stories) → Test (suites, coverage, QA) → Deploy (pipelines, flags, releases) → Monitor (SLIs, alerts, runbooks) → Security (threat models, controls, policies).

Appendix M

The Rosetta Stone Reference

Every UPG entity type has a canonical label: the name used in the spec, the API, and the file format. Most also have framework-specific labels: what Lean Canvas calls it, what JTBD calls it, what the Business Model Canvas calls it. These labels are all aliases for the same underlying entity. The label map is queryable via the `list_type_labels` and `get_type_label` MCP tools; this appendix reproduces the hand-authored priority entries: the types that appear across the most frameworks. §2.5 carries a complementary view of the most-collided concepts grouped by discipline rather than by framework.

Framework abbreviations

Abbreviation	Framework
OST	Opportunity Solution Tree (Torres)
JTBD	Jobs-to-be-Done (Christensen / Ulwick / Moesta)
LC	Lean Canvas (Maurya)
BMC	Business Model Canvas (Osterwalder & Pigneur)
VPC	Value Proposition Canvas (Strategyzer)
DT	Design Thinking (IDEO / d.school)
LS	Lean Startup (Ries)
AARRR	Pirate Metrics (McClure)
OKR	Objectives & Key Results (Doerr)
DORA	DORA Four Keys (Forsgren et al.)
RICE	RICE Scoring (Intercom)
Kano	Kano Model (Kano)
MoSCoW	MoSCoW Prioritisation

Core product types

Rows for `need`, `opportunity`, `learning`, and `insight` are not duplicated here; see §2.5 for those with the discipline-grouped lens.

UPG type	Canonical label	OST	LC	DT	JTBD	VPC
`solution`	Solution	Solution	Solution	Solution	—	—
`hypothesis`	Hypothesis	—	Riskiest Assumption	—	—	—
`experiment`	Experiment	Experiment	—	Test	—	—
`desired_outcome`	Desired Outcome	Desired Outcome	—	—	Desired Outcome	Customer Gain

User types

The `persona` row is not duplicated here; see §2.5.

UPG type	Canonical label	OST	LC	DT	JTBD	BMC	VPC
`job`	Job	Opportunity (job)	—	Task	Job	—	Customer Job
`desired_outcome`	Desired Outcome	Desired Outcome	—	—	Desired Outcome	—	Customer Gain

Strategy & metrics types

UPG type	Canonical label	OKR	AARRR	DORA	LC
`objective`	Objective	Objective	—	—	—
`key_result`	Key Result	Key Result	—	—	—
`outcome`	Outcome	Outcome	—	—	—
`metric`	Metric	Key Result Metric	Pirate Metric	DORA Metric	Key Metric

Business model types

UPG type	Canonical label	BMC	LC
`value_proposition`	Value Proposition	Value Proposition	Unique Value Proposition
`partnership`	Partnership	Key Partner	—
`key_resource`	Key Resource	Key Resource	—
`key_activity`	Key Activity	Key Activity	—
`customer_relationship`	Customer Relationship	Customer Relationship	—
`distribution_channel`	Distribution Channel	Channel	Channel
`revenue_stream`	Revenue Stream	Revenue Stream	Revenue Stream
`cost_structure`	Cost Structure	Cost Structure	Cost Structure

Design & experience types

UPG type	Canonical label	DT	LC
`user_journey`	User Journey	Journey Map	Customer Journey
`design_question`	Design Question	How Might We	—
`design_concept`	Design Concept	Concept	—
`prototype`	Prototype	Prototype	—
`observation`	Observation	Observation	—

Prioritisation types

UPG type	Canonical label	RICE	Kano	MoSCoW
`feature`	Feature	Scored Item	Classified Feature	Prioritised Item
`user_story`	User Story	—	—	Prioritised Story
`solution`	Solution	Scored Solution	—	—

Engineering & operations types

UPG type	Canonical label	DORA	Notes
`service_level_indicator`	Service Level Indicator	SLI	—
`service_level_objective`	Service Level Objective	SLO	—
`deployment`	Deployment	Deployment	—
`ci_pipeline`	CI Pipeline	Deployment Pipeline	—
`incident`	Incident	Incident	Designations: operational, security, performance

Consolidated types with designations

Several UPG types subsume multiple framework-specific types by adding a `designation` property. The designation selects a sub-type label at display time without splitting the entity type.

UPG type	Designations	Old types subsumed
`need`	`pain` → Pain Point · `gap` → Need · `desire` → Desire · `constraint` → Constraint	`pain_point` · `user_need`
`metric`	`north_star` · `kpi` · `driver` · `input` · `guardrail` · `proxy` · `health` · `vanity`	`kpi` · `north_star_metric` · `input_metric`
`experiment`	`discovery` · `ab_test` · `growth` · `pricing` · `usability`	`ab_test` · `growth_experiment` · `pricing_experiment`
`decision`	`product` · `architecture` · `strategic` · `operational`	`product_decision` · `architecture_decision` · `design_decision`
`user_journey`	`current_state` · `future_state` · `day_in_the_life` · `service_blueprint`	—
`insight`	`atomic` · `composite` · `strategic`	`research_insight` · `finding` · `ux_insight`
`incident`	`operational` · `security` · `performance`	—

The full alias index

The complete alias map resolves over 600 alt-labels and framework labels to their canonical UPG types. This includes tool-specific terms (Linear `issue` → `feature` or `bug`, Notion `database item` → type-driven, GitHub `PR` → `feature`), domain vocabulary (DDD `aggregate`, `bounded context`; SRE `SLI`, `SLO`, `error budget`; AARRR funnel stages), and common synonyms (`squad` → `team`, `pentest` → `penetration_test`, `jtbd` → `job`). The full map is available via `list_type_labels` (returns all entries) and `get_type_label({ type: 'need' })` (returns one entry with all alt labels and framework labels).

Appendix N

Canonical Form and the `$upg` Header

A `.upg` file is co-authored by humans and AI agents, version-controlled, reviewed in pull requests, and synced between local and cloud copies. Plain JSON serves none of these well: array and key order drift between writers, volatile timestamps churn on every save, and a deeply nested tree is hard to read in a diff. The remedy is a **canonical form**: a deterministic serialisation in which the same logical graph always produces byte-identical output, no matter which tool wrote it. Git then diffs *meaning* rather than *formatting*, in the same way `gofmt`, `prettier`, and `terraform fmt` discipline source code. This appendix specifies that form and the reserved `$upg` header that accompanies it.

N.1 Relationship to RFC 8785

The canonical form adopts **RFC 8785, the JSON Canonicalization Scheme (JCS)**, for all object-internal rules, with two deliberate departures driven by the review lifecycle.

Rule	RFC 8785 (JCS)	UPG canonical form	Rationale
Object key ordering	sort by UTF-16 code unit	same	adopt verbatim
Number serialisation	ECMAScript `Number`-to-string	same	adopt verbatim
String escaping / charset	minimal escaping, UTF-8	same	adopt verbatim
Whitespace	none (single line)	pretty-printed: two-space indent, one element per line, LF	a 600-node graph on one line is unreviewable
Array order	preserved as written	the set-like collections are sorted	insertion order is an accident of write history and the dominant source of spurious diffs

JCS rules apply recursively inside every object and inside every order-bearing array; the set-like collections are sorted by the keys in N.3; the result is pretty-printed.

N.2 The `$upg` header

Every canonical file opens with a reserved `$upg` object that consolidates metadata previously scattered as siblings of the data:

- `format_version`: the serialisation version (currently `1.0.0`), distinct from `spec_version` so the *file format* can evolve independently of the *ontology*.
- `spec_version`: the catalogue version the data conforms to.
- `kind`: `"portfolio"` for portfolio documents; omitted for single-product.
- `product` (or `organization` for portfolios): an identity summary for at-a-glance orientation.
- `summary`: a one-line description, re-derived on every write so it cannot drift.
- `counts`: element counts, for cheap size reads without walking the graph.
- `provenance`: the writing tool, its version, and `exported_at` (the volatile fields live here, isolated from the body).
- `integrity`: `{ algorithm, body }`, a checksum of the canonical body.

Below the header, `product` / `nodes` / `edges` (single-product) or `organization` / `product_areas` / `portfolios` / `products` / `cross_edges` (portfolio) carry the graph itself.

N.3 Ordering rules

All comparisons are by UTF-16 code unit, never locale-aware (locale collation is not stable across machines, which would break byte-identity).

- **Object keys** are emitted in a fixed per-type order (identity fields first, open `properties` last), with any unlisted keys sorted after them. Keys inside `properties` and other open objects are fully sorted.
- **`nodes`** are sorted by `(type, slug or title, id)`, so a new node lands beside its siblings.
- **`edges`** and **`cross_edges`** are sorted by `(source, target, type, id)`.
- **`tags`** are sorted and de-duplicated (an unordered label set). **`aliases`** preserve insertion order (an append-only slug-rename history).
- Empty optional fields (`""`, `[]`, `{}`, `null`) are omitted; required identity fields are always present.

N.4 Integrity, volatility, and drift repair

`integrity.body` is computed over the canonical serialisation of the body alone, with the header and its volatile fields excluded. A no-op re-export therefore leaves the checksum unchanged and produces at most a one-line timestamp diff; a real change moves the checksum. On load, a serialiser also repairs legacy drift: a `properties` or `tags` value mistakenly stored as a JSON-encoded *string* (observed in production graphs) is restructured to its object or array form, or rejected with a precise error if it does not parse to the expected type.

N.5 `upg fmt` and its guarantees

A single reference serialiser, shared by every writer (the MCP server, the CLI, the SDK, and cloud export), guarantees that no two tools format the same graph differently. The CLI exposes it as `upg fmt` (rewrite in place) and `upg fmt --check` (a CI gate that fails on any non-canonical file). The serialiser satisfies three properties, each enforced by tests:

1. **Idempotent:** `fmt(fmt(x))` equals `fmt(x)` byte-for-byte.
2. **Round-trip:** parsing then re-serialising preserves the graph (modulo volatile fields and the drift repairs of N.4).
3. **Writer-agnostic:** the output is identical regardless of the key or array order of the input.

N.6 Validation and compatibility

A published JSON Schema (Draft 2020-12), bound to the `.upg` extension, lets any editor or CI step validate a file's structure independently of the serialiser; the schema and the canonical form are complementary, since a schema can constrain shape but not byte order. For compatibility, readers accept both the `$upg` envelope and the older flat layout and normalise to a single in-memory shape, so a fleet of tools can adopt the canonical form incrementally rather than in lockstep.

Appendix O

The Portfolio Document

The portfolio document (§3.10, §4.7) is the multi-product form of the format. It is a strict, additive superset of the single-product `UPGDocument`: the same header, canonical form, and reference syntax, extended with an organisation root, the portfolio and product-area hierarchy, embedded member products, cross-product edges, and an optional shared-vocabulary registry.

The document interface

```
interface UPGPortfolioDocument {
  upg_version: string                  // spec version the data conforms to
  type: 'portfolio'                    // discriminator (serialised as
$upg.kind)
  exported_at: string
  source: UPGSource
  organization: UPGOrganization        // the owning organisation (one root)
  portfolios: UPGPortfolio[]           // strategic axis — where we invest
(nestable)
  product_areas: UPGProductArea[]      // organisational axis — who owns what
(nestable)
  products: Array<UPGProduct & {       // members, each a full single-product
graph
    nodes: UPGBaseNode[]
    edges: UPGEdge[]
  }>
  cross_edges: UPGCrossEdge[]          // edges spanning products or the
registry
  registry?: UPGRegistry               // shared-vocabulary tier (optional,
additive)
}

interface UPGRegistry {
  nodes: UPGBaseNode[]                 // canonical entities, addressed
`registry/{id}`
  edges?: UPGEdge[]                    // reserved: canonical-internal
relationships
}
```

A single-product file (`UPGDocument`, Appendix I) embeds unchanged inside `products`, so adopting the portfolio tier never invalidates an existing graph. The `registry` section is optional: a portfolio without shared vocabulary omits it and stays byte-identical to one that never had it.

Registry addressing

Canonical entities live in `registry.nodes` and are referenced as `registry/{node_id}`, where `registry` is a reserved pseudo product-id (`REGISTRY_PRODUCT_ID`) that product creation rejects, so a real product can never collide with it. A canonical entity is an ordinary `UPGBaseNode`; canonical-ness is conferred by living in the registry, not by a distinct type or flag. The `instance_of` cross-edge links a product node to its canonical (same type, enforced at write time), and the area-to-audience and foundations cross-edges resolve their registry targets the same way. Registry node ids are stable, type-prefixed slugs (`persona_developer`, `specification_openapi`) so an instance edge survives renames.

A worked portfolio

A minimal portfolio: one organisation, one portfolio, one area, two member products that each carry a local `persona`, a registry that defines the canonical *Developer* persona and the *OpenAPI* specification, and cross-edges that bind the two products to the shared vocabulary (`instance_of`) and to the shared specification (`product_implements_specification`). Member-product `nodes`/`edges` are elided for brevity where unchanged from the single-product form.

```
{
  "$upg": {
    "format_version": "1.0.0",
    "spec_version": "0.10.6",
    "kind": "portfolio",
    "organization": { "id": "org_acme", "title": "Acme" },
    "counts": { "products": 2, "cross_edges": 5, "registry_nodes": 2 },
    "provenance": { "tool": "upg-mcp-server", "tool_version": "0.10.6" },
    "integrity": { "algorithm": "sha256-128", "body":
"5e9b1c4d2f8a0736b9e1d4c7a206f3e8" }
  },
  "organization": { "id": "org_acme", "title": "Acme", "industry": "Developer
Tools" },
  "portfolios": [
    { "id": "prt_acme", "title": "Acme Platform", "properties": {
"hierarchy_model": "nested" } }
  ],
  "product_areas": [
    { "id": "pa_devtools", "title": "Developer Tools", "properties": {
"strategic_priority": "high" } }
  ],
  "products": [
    {
      "id": "p_docs", "title": "Acme Docs", "stage": "growth",
      "nodes": [
        { "id": "n_docs_dev", "type": "persona", "title": "Developer",
          "properties": { "audience_role": "user", "is_primary": true } }
      ],
      "edges": []
    },
    {
      "id": "p_api", "title": "Acme API", "stage": "growth",
      "nodes": [
        { "id": "n_api_dev", "type": "persona", "title": "Platform
Developer",
          "properties": { "audience_role": "user" } }
      ],
      "edges": []
    }
  ],
  "registry": {
    "nodes": [
      { "id": "persona_developer", "type": "persona", "title": "Developer",
        "properties": { "audience_role": "user" } },
      { "id": "specification_openapi", "type": "specification", "title":
"OpenAPI",
        "description": "The governed REST API description format Acme
products implement and expose." }
    ]
```

```
    { "id": "x_1", "source": "p_docs/n_docs_dev", "target":
"registry/persona_developer", "type": "instance_of" },    { "id": "x_2",
"source": "p_api/n_api_dev", "target": "registry/persona_developer", "type":
"instance_of", "alias": true },    { "id": "x_3", "source": "p_docs",
"target": "registry/specification_openapi", "type":
"product_implements_specification" },    { "id": "x_4", "source": "p_api",
"target": "registry/specification_openapi", "type":
"product_exposes_specification" },    { "id": "x_5", "source": "p_docs",
"target": "p_api", "type": "depends_on_product" }  ]}
```


The two products are independent graphs, but the portfolio now answers questions neither holds alone: *which products serve the Developer persona?* (two, via `instance_of`), *which products touch the OpenAPI spec, and how?* (Docs implements it, API exposes it), and *what is downstream of the API?* (Docs depends on it). The `alias: true` on `x_2` marks *Platform Developer* as a sanctioned product-local name for the canonical *Developer*, so `portfolio_validate` records it as sanctioned rather than as drift.

---

*UPG is open source (MIT). Repository: **github.com/unified-product-graph** (https://github.com/unified-product-graph). Specification: **unifiedproductgraph.org/spec** (https://unifiedproductgraph.org/spec).*

Unified Product Graph

UPG CORE · V0.10.6 · 2026-06-13 · COMMIT B24EEB075

AUTHOR	Faheem Hasan
AFFILIATION	Arkheiev UG, Berlin, Germany
CONTACT	faheem@theproductcreator.com
REPOSITORY	github.com/unified-product-graph
LICENCE	CC BY 4.0 · Specification under MIT

Typeset on the web and exported to PDF via Paged.js. Body set in Source Serif 4; headings and display set in Sora; code set in JetBrains Mono; entity labels set in Space Mono.

END OF SPECIFICATION