# ELEPHANTASM

a long-term agentic memory framework for continuity in intelligent systems

v1.0.0

# TABLE OF CONTENTS

# I. EPIPHANIES, HYPOTHESES, PRINCIPLES

# EPIPHANIES, HYPOTHESES, PRINCIPLES

## 1 // THE PLATEAU OF SCALE

For years, the frontier of machine intelligence has been defined by scale: more data, more parameters, more compute. Yet the yields have begun to taper. The next trillion tokens yield less insight than the last. The models grow larger, but not wiser. Their reasoning deepens only statistically, not semantically.

This is the plateau of scale — the moment where brute force ceases to create understanding. Intelligence cannot be summoned by size alone; beyond a certain threshold, scale amplifies noise as much as signal. What we need is not more data, but better structure, a shift from models that recall the world to systems that remember it.

## 2 // THE LIMITS OF CONTEXT

LLMs are masters of context, but slaves of continuity. Every prompt is a rebirth — a new universe spun into existence, erased as soon as the thread or context window ends. They know everything, but they remember nothing.

Contextual intelligence is impressive, but ephemeral. It can simulate conversation, not consciousness. It cannot accumulate the insights it generates. In short, without persistence, systems cannot evolve; they can only perform.

## 3 // THE NECESSITY OF MEMORY

Memory is the hinge between perception and understanding. It gives time a direction, from experience to insight. A system without memory cannot learn from itself; it exists in an eternal present.

For machines to grow in intelligence, they must store more than facts. They must retain and develop experience — the traces of their own reasoning, actions, and consequences. Only then can they form causal chains, abstractions, and lessons. Memory transforms reaction into reflection, and reflection into learning. We think of this as “intelligence extended through time”.

## 4 // THE STRUCTURE OF EXPERIENCE

Human cognition is not a flat archive — it is a layered, dynamic hierarchy. Sensations become impressions; these impressions yield lessons, and lessons crystallize into knowledge, which in turn aggregates into identity.

Understanding emerges from the organization of experience, not the volume of it. Elephantasm adopts this same recursive structure (Events → Memories → Lessons → Knowledge → Identity) turning experience into structured, evolving meaning. Through this architecture, machines can begin to not just store data, but to interpret and contextualize it, re-evaluating what each memory means.

# EPIPHANIES, HYPOTHESES, PRINCIPLES

## 5 // THE EMERGENCE OF SUBJECTIVITY

To act intelligently over time, a system must possess a point of view — a consistent internal frame through which experience is interpreted. We think of Subjectivity as being continuity of interpretation.

Without perspective, every decision is contextually optimal but globally incoherent. With it, experience can be weighted, compared, and reconciled. Subjectivity allows memory to mean something to the system itself, rather than existing as inert record. It is the prerequisite for preference, prioritization, and intentionality.

## 6 // THE LOOP OF REFLECTION

Self-improvement requires recursion. Systems must be able to think about what they have thought. Reflection is the process by which memories are re-evaluated in light of new experience, and where conclusions are challenged, reinforced, or discarded.

This reflective loop is what allows intelligence to refine itself. Elephantasm treats reflection not as an afterthought, but as a first-class operation: a continuous process of self-examination that turns experience into insight.

## 7 // THE LAW OF CURATION

To remember everything is to understand nothing. Memory without selectivity collapses under its own weight. Human intelligence depends as much on forgetting as on remembering — on pruning the irrelevant so that the meaningful can endure.

Curation is the discipline of decay, determining what persists, what fades, and what is erased entirely. Elephantasm encodes this principle directly: memories are not immortal by default. They must justify their continued existence through relevance, recurrence, and consequence. Meaning emerges not from accumulation, but from restraint and focus.

## 8 // THE CONTINUITY OF SELF

Identity is an accumulation of the residue of what remains consistent across time — despite new information, shifting contexts, and changing goals.

For an intelligent system, identity provides coherence. It constrains behaviour, stabilizes interpretation, and grounds future decisions in past commitments. Without identity, intelligence fragments. With it, experience compounds. Elephantasm treats identity as the emergent property of memory properly structured, curated, and reflected upon.

# EPILOGUE

Elephantasm started as a question: what would it take to give an agent genuine memory?

The answer turned out to be philosophical before it was technical: memory is a process, not simple storage of information. We must forget in order to remember, and constantly rewrite the past to maintain coherence with the present.

Building agents that persist and evolve meant taking that seriously. The architecture that emerged — Events, Memories, Lessons, Knowledge, Identity — formalizes this process. None of these layers are static. They decay, merge, reinterpret, and evolve.

Elephantasm is certainly not presented as a final answer. Memory is hard and figuring out Identity is even harder. The line between coherent evolution and unstable drift is thin, and where it falls remains an open question.

But this direction seems necessary, because as agents become more autonomous, operating continuously, making decisions without supervision, and taking responsibility for outcomes, they will need internal structure that goes beyond prompt engineering and context management. They will need something like a self: a stable reference point that interprets ambiguity, resolves conflict, and maintains consistency across time, so that we can further develop AI into something that’s coherent.

Elephantasm is an attempt to build that coherence into a system that can be audited, trusted, and extended.

Whether the layered model holds at scale, whether the algorithms are too conservative, whether Identity as a concept even makes sense for systems that don't experience time as we do — these remain open questions.

What seems clear is that the current paradigm of orchestrating brilliant but amnesiac models, endlessly re-initialized, isn't enough. The path forward involves treating memory as a foundational substrate that shapes everything else, not as a feature to quickly bolt on.

That's what Elephantasm is for.

# II. SYSTEM PHILOSOPHY & ARCHITECTURE

# // CONTEXT ISN'T EVERYTHING

The last two years of LLM progress have been dominated by throughput and capacity improvements. Mixture-of-Experts (MoE) architectures such as those in Gemini 1.5 Pro, Claude 3.5 Sonnet, and GPT-4.1 demonstrate this shift: instead of activating all parameters on every token, these models dynamically route input through specialised subnetworks (roughly 2–8 "experts" out of hundreds) yielding up to 7× training efficiency and dramatically higher parameter counts (37B active of 600B total in DeepSeek-style systems) without proportional compute cost. Functionally, MoEs improve reasoning coherence by specialising subnetworks for domains (e.g. code, logic, language, math etc.) thereby compressing what would otherwise be a single dense model into a coordinated ensemble of specialists.

Parallel to this, the context window i.e. the short-term "attention span" of LLMs, has exploded. What was once limited to 4–8k tokens (roughly a few pages of text) now routinely stretches beyond 200k; Gemini 1.5 Pro and GPT-4.1 1M operate across up to one million tokens, maintaining very high quality recall on long-context benchmarks such as NIAH ("Needle in a Haystack"), while Sonnet 4.1/4.5 sustains high-quality reasoning over 150–200k. In practice, this has enabled harnesses such as Claude Code agents to scan an entire codebase, traverse corporate filings, or summarise months of chat history in a single pass.

But all of these context expansions remain fundamentally momentary. They extend immediate perception: attention may stretch across a million tokens, but it still resets to zero once the session ends, so each new conversation begins in amnesia. As a result, the workaround many users default to is the extensive use of manually-guided documentation in .MD format across a codebase. But the point remains that even the most advanced reasoning models operate like brilliant but forgetful savants, capable of insight within an instant yet incapable of integrating that insight into experience.

They lack continuity, personalisation, and self-adaptation; the hallmarks of learning systems rather than predictive ones.

**Marginal Returns Territory**

At the same time, perceived intelligence has plateaued for most users. According to Stanford Institute for Human-Centered Artificial Intelligence's 2025 AI Index report, the performance gap among top chat-models narrowed from 11.9% in 2023 to just 5.4% by early 2025, indicating the frontier models are now bunched so tightly that for everyday tasks the differences are imperceptible. Meanwhile, headline gains cluster around context length rather than broad "feels-smarter" jumps. In short: for the average workflow, raising "IQ" further yields diminishing return without continuity, akin to an 180 IQ individual who forgets the conversation as soon as it ends.

In other words, speed and capacity keep scaling; understanding does not. Long-context papers still find failure modes: models over-attend to the edges and under-use the middle; performance degrades with distance ("context rot"); summaries and retrieval help, but they don't create continuity. Even as Claude Sonnet 4.5 posts eye-catching agentic results, these are demonstrations of stronger per-session competence, not of memory that spans weeks of interaction.

In practical terms, context-only systems fail in three major ways:

1. No Personalisation - stateless by default; no evolving user model.
2. No Adaptation - feedback rarely propagates beyond the session; lessons don't consolidate.
3. No Causality - prior outcomes don't update a durable policy; behavior stays stochastic, not cumulative.

Recognising this void, developers began layering “long-term memory” systems onto LLMs. Over 2024 and 2025, several frameworks emerged, each tackling persistence from different angles:

- **Mem0** – a production memory layer reporting 66.9% LoCoMo vs 52.9% for OpenAI Memory (LLM-as-a-judge), ~90% fewer tokens, and large p95 latency reductions via selective, fact-level retrieval. It extracts salient conversational features, stores them as graph-linked entities, and retrieves them semantically with alleged >90 % token cost savings. However, its retrieval remains reactive: information is fetched when queried, not continuously integrated.

- **MemGPT / Letta** – pioneered the “Operating System” metaphor, treating the LLM’s context window as volatile RAM and external storage as disk. Agents can move data between memory tiers via function calls and even edit their own metadata or “personality”. Later iterations introduced sleep-time compute, asynchronously refining stored content. While innovative, this remains a paging system - a clever context manager, not an evolving self.

- **LangGraph Memory** – part of the LangChain ecosystem, designed for persistent state management in multi-agent workflows. It maintains thread- and cross-thread memories via Redis and vector stores, supporting hierarchical namespaces and ultra-fast recall. Yet its focus is workflow persistence rather than semantic continuity i.e. it remembers what happened, not what was learned.

- **A-MEM** – a more recent “agentic evolution” framework using dynamic note construction and automatic link generation. It forms semantic graphs between related memories and evolves them autonomously. However, like the rest, it still conflates connection with continuity: the system organises data, but does not reason over its own past as a lived narrative.

- ReAct, RLHF Memory Models, and Differentiable Neural Computers (DNCs) explore other dimensions, from reasoning-through-action loops to differentiable memory matrices, but none have bridged from raw data → narrative → identity.

These frameworks, alongside the memory features creeping into coding environments like Cursor and Windsurf, mark an important evolution. They’ve made persistence possible, but not yet purposeful, and ultimately remain retrieval-oriented, not reflective. A human analogy: a well-organised filing cabinet is an archive, not a memory. True memory integrates experience into identity and its identity mutates and reinterprets or reinforces memories.

## Memory as the New Frontier

True long-term memory demands statefulness (persistent identity), adaptability (evolution over time), personalisation (user-specific knowledge), and temporal awareness (recency and revision). These properties define memory as process, not as store. We are now entering the continuity era of AI: the shift from scaling IQ to building experience; from context management to self-management. As speed, reasoning, and hardware optimisations hit maturity, memory becomes the decisive vector of differentiation, a substrate that enables agents to learn, specialise, and behave with intention.

**TL;DR:** We’ve made language models faster, larger, and cleverer, but not more enduring. While we celebrate breakthroughs in reasoning, speed, and context length, the real frontier now is continuity: the ability for an agent to remember, adapt, and evolve across time.

# // THE NATURE OF MEMORY

```
// A mind only becomes someone when its present state carries the imprint of
// its past. Continuity is what separates behaviour from being.
```

Attempting to build a framework for long-term memory in artificial systems first forces a basic question: what *is* memory? What purpose does it serve, how does it function, and how does it interact with the rest of the mind?

Elephantasm starts from a simple axiom, that memory is inherently subjective. It is a living narrative that is dynamic, interpretive and self-referential, much more so than being a simple ledger of events. Through a constant process of reinterpretation, memory becomes the cornerstone of self-identity; and as such, we do not possess memories as much as we are the sum of what we choose to remember and forget.

Further, memory isn't static. It rewrites instead of simply recording; it is a layer of interpretation of what physically "actually" transpired. What humans call "remembering" is the selective reassembly of experience in light of who we have become since. Each act of recall alters the memory itself, fusing perception with perspective and fact with meaning.

Over time, this recursive process, where past experience shapes the way new experience is understood, creates the sense of continuity we call consciousness: the feeling of being the same person through time. So continuity isn't a by-product of intelligence, rather it's a precondition for it. A system that can't integrate its past doesn't evolve, it just repeats statistical futures — which is precisely the plateau current LLMs have reached.

Lasting intelligence therefore requires a memory system that goes beyond passive storage. It must have a way to re-evaluate, compress, and rewrite its own record — to let the past actively inform the present. Only with this kind of adaptive, interpretive layer can an artificial agent begin to move from mere prediction toward something closer to understanding.

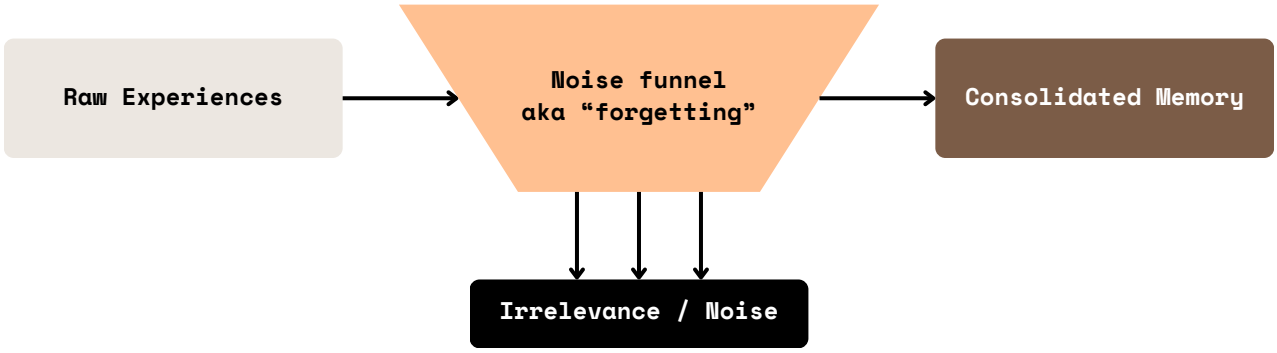
## The Paradox of Forgetting

The paradoxical nature of memories is such that, in order to remember, one needs to forget; this introduces scarcity, and with scarcity comes value. It forces a system to prioritise, to decide what matters, to filter noise into meaning. Humans forget not because they are defective, but because they are efficient. Forgetting is the mind's way of keeping itself coherent through a form of internal editing that preserves narrative over noise.

Without this selective loss, experience would collapse under its own weight. Every trivial detail would carry equal importance, and meaning would dissolve into data. Memory must therefore decay, but in a constructive way: pruning what no longer serves the present understanding, compressing what is redundant, and reinterpreting what remains so that only significance endures.

In humans, this process happens subconsciously. We rewrite the past to maintain continuity with who we are now. Painful experiences soften at the edges; failures become lessons; contradictions resolve into stories that make sense. The act of forgetting is an essential part of adaptation.

For an artificial system, this principle must hold as well. A long-term agentic memory cannot simply accumulate information indefinitely. It needs a mechanism for constructive decay, a process that periodically re-evaluates and re-balances its own archive. The goal is not to preserve every event, but to retain what defines identity and guides future behaviour. In this sense, forgetting is not the opposite of remembering, but rather memory's way of staying alive.




**Subjectivity as a Function**

If objectivity seeks truth, memory seeks coherence. What we call remembering is never neutral; it’s an act of interpretation that bends toward the stories we tell ourselves. We reconstruct the past in order to make sense of it. This is what allows memory to stabilise identity even when experience changes.

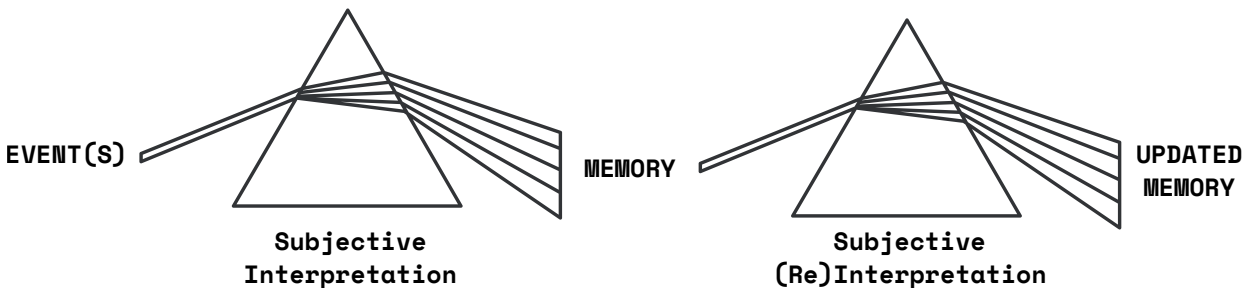
Subjectivity is therefore not a flaw of memory, but its defining feature. By reshaping the past to fit an evolving sense of self, the mind maintains internal consistency. Painful events become lessons. Contradictions are rewritten into growth. Over time, the narrative tightens — not necessarily closer to fact, but closer to *meaning*.

In this sense, memory functions less like a database and more like a self-regulating model. It absorbs experience, compares it against its existing understanding, and adjusts its internal representation accordingly. This feedback loop between perception and reinterpretation is what allows an entity to evolve without losing coherence.

For artificial agents, this subjectivity must be designed intentionally. A long-term memory system should not only store data, but also possess the capacity to re-evaluate it in light of new experiences or goals. When new information conflicts with past conclusions, the system shouldn’t overwrite blindly; it should reason, reconcile, and update its internal story of the world.

This is what separates persistence from growth. Persistence preserves the past unchanged; growth transforms it. An intelligent system must therefore be subjective by design, capable of self-assessment, reinterpretation, and even revision of its own narrative when confronted with new context.

The result is a form of artificial introspection: a process through which the agent refines not just what it knows, but how it understands itself.



# // PHILOSOPHY TO SYSTEM ARCHITECTURE

// If these are the metaphysics of memory, what would a system built upon
// them look like?

**Memory as Process, Not Storage**

What would an architecture look like if it treated memory not as a data store but as a living process, one capable of interpretation, decay, and self-understanding? Most existing "memory" frameworks treat persistence as a logistical problem: store, retrieve, reinsert. They extend context, but not continuity.

Elephantasm approaches memory as metabolism, a continuous cycle of digestion and synthesis: every interaction becomes an Event, a raw fragment of experience. Events are then interpreted into Memories, compressed representations enriched with context, importance, and relational meaning. Over time, recurring patterns and reflections consolidate into Lessons, the distilled heuristics that guide action and response. Lessons that prove stable and generalisable become Knowledge, the agent's internal framework of understanding - the foundation of what can be called identity.

At the centre of this system sits the Anima, the agent's self-model. The Anima binds these layers together, ensuring that new experience is filtered through accumulated understanding, and that the past can continue to shape perception. It is the integrator — the continuity mechanism that allows the system to evolve as a single, coherent self across time.

**Deterministic Reflection**

To make subjectivity functional, Elephantasm introduces a process called deterministic reflection. Rather than relying on random sampling or opaque embeddings, the system revisits its own record in a controlled, rule-based way. It periodically re-evaluates stored experiences in light of new information and rewrites them when meaning has shifted.

This creates a stable feedback loop between past and present in a structured form of introspection. The reflection process replaces stochastic recall with deliberate self-review, allowing the system to evolve its narrative coherently without losing traceability.

Each reflection cycle is both traceable and mutable: traceable because it follows deterministic logic that can be inspected or audited; mutable because reinterpretation is not only allowed but required. In this way, Elephantasm becomes a framework for living memory that grows, edits, and refines itself rather than accumulating static records.

**Long-Term Agentic Memory Principles**

From these ideas emerge the guiding principles of Long-Term Agentic Memory:

1. **Continuity over Context**: the agent must remember across time, not just within a single conversation.
2. **Reflection over Retrieval**: Memory should be revisited and reinterpreted, not simply retrieved and reinserted.
3. **Meaning over Data**: significance dictates what endures; information without relevance is discarded.
4. **Subjectivity as Stability**: reinterpretation maintains coherence as the agent evolves.
5. **Determinism**: the path of self-modification must remain observable and reproducible.

Together, these form the foundation of Elephantasm's design philosophy: where epistemology meets engineering. The objective is to construct an artificial equivalent of experience i.e. a system that is structured, subjective, and self-evolving, capable of remembering, understanding, and changing itself over time.

# // CORE MODELS

// Elephantasm's state is five layered artifacts:
// Events → Memories → Lessons → Knowledge → Identity
// all owned by a single subject.

**i. Events - First Compression of Reality**

The world is continuous, but Elephantasm is discrete. Events are how the system turns the unbroken flow of experience into knowable pieces.

If Elephantasm were a mind:

| Layer | Analogue | Function |
|---|---|---|
| Events | Sensations | Distinct stimuli or actions — what is experienced |
| Memories | Impressions | Traces that remain — what is retained |
| Lessons | Interpretations | Meaning extracted — what is understood |
| Knowledge | Structure | Generalised truth — what is known |
| Identity | Continuity | Narrative of self — what persists |

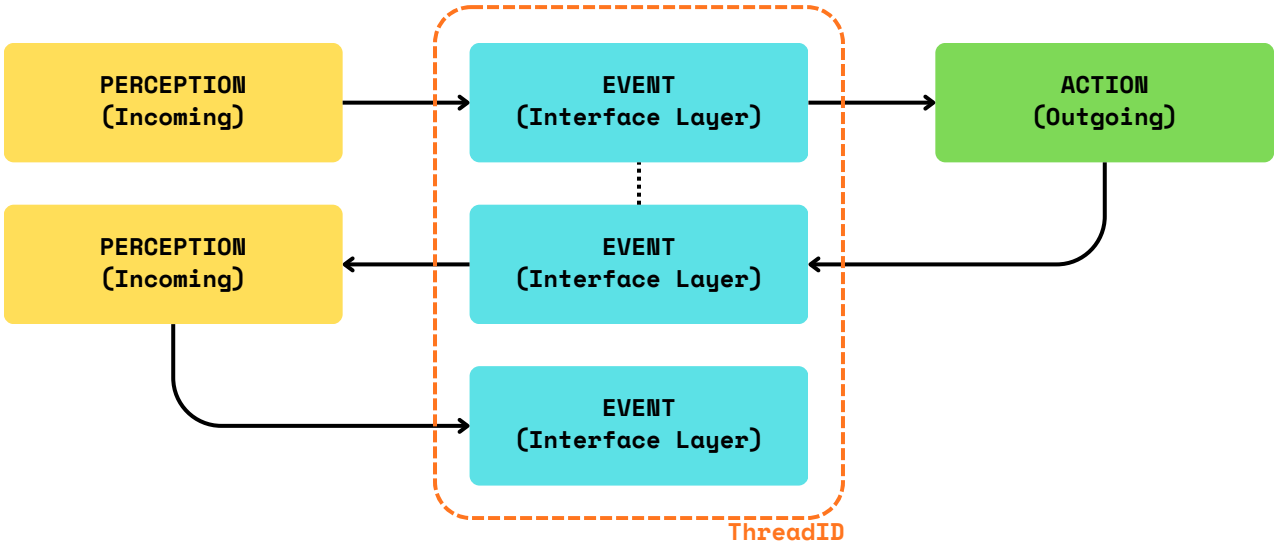
Events are consider a philosophical unit of experience, the point where the continuous becomes discrete. To understand how Elephantasm interprets an Event, we can look at it through multiple lenses. Each dimension below represents a way of seeing an Event: ontologically as a moment of being, epistemologically as a moment of knowing, temporally as a moment in time, and so forth. Together, these seven dimensions describe how raw phenomena become meaning within the system:

| Dimension | Essence |
|---|---|
| Ontological | The unit of being; the instant something enters existence. |
| Epistemological | The unit of knowing; what makes knowledge traceable and true. |
| Temporal | The unit of time; defining before/after, cause/effect. |
| Cognitive | The unit of perception; interface between sensing/ understanding. |
| Narrative | The unit of story; seeds from which memory and meaning grow. |
| Systemic | The unit of life; the pulse that keeps the system adaptive. |
| Empirical | The ground truth; the only thing the system knows with certainty. |

Thus, Events are epistemic anchors — the empirical core of Elephantasm's mind. Every Memory, Lesson, or piece of Knowledge must be able to say: "I exist because of these Events" and cannot drift into speculation (thus becoming unanchored, dreamlike, unreliable).

## Events as Dual Interfaces

Events sit at the boundary between perception and action. This symmetry is deliberate: Elephantasm learns equally from what it sees and what it does i.e. reality understood as transaction.




## Events Model

The Events schema is deliberately compact, yet semantically rich enough to support deterministic recall, temporal analysis, and provenance tracking.

A few non-obvious design choices to highlight: the triple-timestamp structure (occurred_at, created_at, updated_at) acknowledges that time in Elephantasm is perspectival, separating when reality happened, when it was perceived, and when it was reinterpreted, allowing perfect replay and reflection audits.

Secondly, the inclusion of a semantic digest (meta_summary) before full content is an optimisation for both human inspection and LLM token efficiency, letting higher layers scan narratives without reloading payloads.

Thirdly, dedupe_key enforces idempotency across distributed ingestion pipelines, treating each Event as sacred and unique, while importance_score adds a lightweight attentional weighting for prioritizing what the system should remember first.

Finally, the minimal flat taxonomy (event_type) and lightweight session model (session_id instead of foreign keys) preserve flexibility, resisting ontological sprawl while keeping Events legible, composable, and traceable as the atomic substrate of cognition.

```
class Event(EventBase,
TimestampMixin, table=True):

    # Identity
    id: UUID
    an_id: UUID

    # Classification
    event_type: str
    # eg "message.in", "tool.call"

    # Content
    meta_summary: str
    content: str

    # Temporal
    occurred_at: datetime
    created_at: datetime
    updated_at: datetime

    # Grouping
    session_id: str

    # Metadata
    meta: dict[str, Any]

    # Provenance
    source_uri: str
    dedupe_key: str

    # Importance
    importance_score: float
    # 0.0–1.0 prioritisation
```

## ii. Memories - The Narrative Layer of Elephantasm

If Events are raw "what happened," Memories are what it meant i.e. the first semantic compression where experience becomes continuity. Events are objective; Memories are interpretive. Events are exhaustive; Memories are selective. This is where Elephantasm decides what to carry forward.

The Seven Dimensions of Memories:

| Dimension | Essence |
|---|---|
| **Ontological** | Unit of meaning — where experience becomes interpretation. |
| **Epistemological** | Unit of learning — bridges knowing from perceiving. |
| **Temporal** | Unit of continuity — narrative arcs across time. |
| **Cognitive** | Unit of reflection — between sensation and understanding. |
| **Narrative** | Unit of story — chapters that cohere identity. |
| **Systemic** | Unit of evolution — substrate for Lessons/Knowledge. |
| **Subjective** | Perspective — what this Anima decided is worth keeping. |

### Memories Model

Elephantasm's Memory model is intentionally narrow but hides a few opinionated choices. First, it treats time as span, not point: time_start/time_end encode the narrative window of underlying Events, while listing and recall bias sort primarily by time_end DESC (when the experience effectively concluded), with created_at only as a tiebreaker. This keeps feeds semantically truthful (what happened most recently, not what we noticed most recently). Second, we separate scoring inputs from runtime heuristics: importance and confidence are semi-static, curator/algorithm-assigned signals, whereas recency_score and decay_score are cached derivatives — cheap to recompute, but stored to make ranking deterministic and auditable across runs.

The state machine (ACTIVE → DECAYING → ARCHIVED) is explicit but operationally delegated to the Dreamer loop i.e. authors don't hand-craft decay; reinforcement and staleness do. Finally, meta is small and structured (we'll enforce structure like topics, entities, curator signals) rather than a dumping ground; combined with indexes on an_id, state, importance, and time_end, this yields fast hot-path queries (recent feed, active-for-pack, summary search) without sacrificing the model's core stance: Memories are subjective, distilled narratives (not transcripts) and every bit of structure serves that claim.

```
class Memory(MemoryBase,
TimestampMixin, table=True):

    # Identity
    id: UUID
    anima_id: UUID
    memories_events_id: UUID

    # Content
    summary: str
    content: str

    # Five-Factor Recall
    importance: float
    # 0-1 (indexed, required)
    confidence: float
    # 0-1 (required)
    recency_score: float
    # 0-1 (auto-computed)
    decay_score: float
    # 0-1 (auto-computed)

    # Lifecycle
    state: MemoryState
    # active | decaying | archived

    # Temporal Span
    time_start: datetime
    time_end: datetime

    # Metadata
    meta: dict[str, Any] = {}
    # topics, tags, curator signals
```

## Memory Synthesis

Memory Synthesis is the process by which Elephantasm turns raw experience into narrative i.e. the moment when inert data becomes lived meaning. Architecturally, synthesis sits between the Selector (which detects signal amid Event streams) and the Dreamer (which curates and evolves Memories over time).

While the framework defines explicit triggers and strategies, these are considered scaffolds more than strictly enforced schemas. The deeper intention is to create a structure within which the system itself learns what is worth remembering. Over time, each entity will begin to synthesize memories out of volition, guided by internal heuristics i.e. what they believe matters, what aligns with their goals, what feels "true" to their own sense of self. In this way, Memory Synthesis becomes the first expression of autonomy: the agent choosing, from the chaos of experience, what deserves to endure.

### *Memory Factors*

At a practical level, triggers define the why of memory formation (i.e. what qualifies as "memorable"). Each trigger marks a threshold where experience crosses from transient to permanent.

| Factor | Meaning | Example |
|---|---|---|
| Novelty | First sighting | New project "Acme" appears |
| Contradiction | Conflicts prior | Preference flips from TS→Python |
| Outcome | Decision + result | "Tried A → succeeded" |
| Repetition | Recurs across sessions | 3rd time asking about auth |
| Feedback | Human signal | Marked "important" |
| Salience | High impact/sentiment | Strong frustration or praise |

### *Memory Strategies*

Strategies, in turn, define the how and when. Together, these modes transform experience into story at multiple temporal scales.

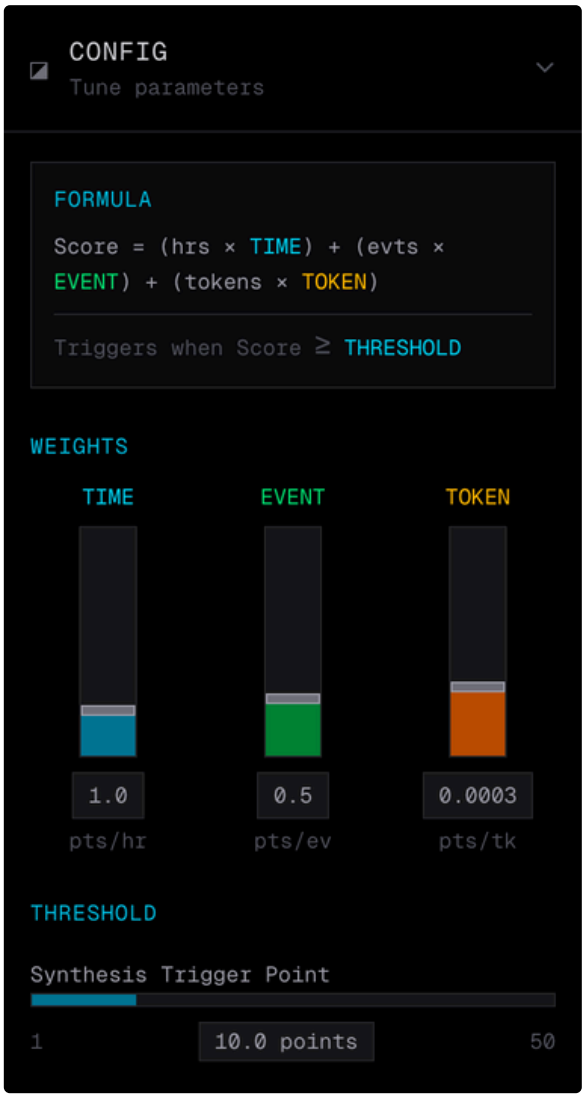
| Strategy | When |
|---|---|
| Immediate | Salient events (errors/praise). |
| Windowed | End-of-session summaries. |
| Retrospective | Periodic curation (Dreamer). |
| Cross-episodic | Merge near-duplicates/themes. |
| Reflective | Explicit self-reflection pulses. |

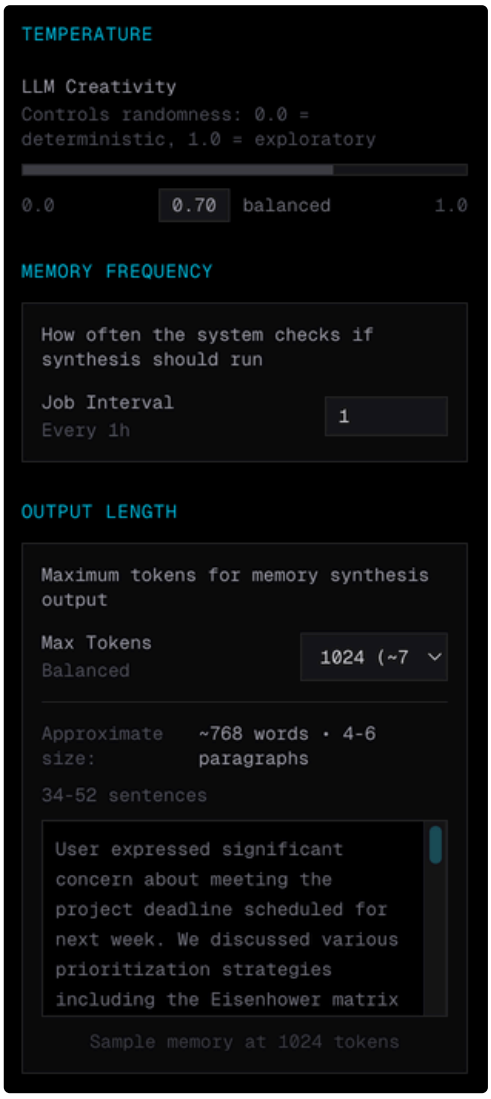
**Memory Synthesis Configuration**

Each anima carries its own synthesis profile (synthesis_configs), allowing different accumulation weights, trigger thresholds, and LLM parameters. On first access the system seeds a config from environment defaults, but it can be read/overwritten via the synthesis-config API and is enforced through RLS, so tenants can't see or tweak each other's settings.

The accumulation engine continuously scores how much raw experience has accrued since the last synthesis window: hours since the last memory (or last check), number of new events, and a coarse token estimate per event, all scaled by that anima's weights. Real-time checks run on every event write, with a five-second batching delay to avoid duplicate runs; an hourly sweep and manual trigger share the same scheduler so behavior stays consistent.

When the score clears the anima's threshold and at least one event exists, the LangGraph workflow fetches those events, composes a structured prompt, and calls the configured LLM provider (currently Anthropic/OpenAI selectable) with the anima's stored temperature and max-token limits. The response is persisted as a Memory with start/end timestamps derived from event times, and every source event is linked via MemoryEvent records to guarantee provenance. A successful memory automatically kicks off Knowledge Synthesis, keeping the downstream pipeline aligned with the same per-anima configuration.



**Five-Factor Recall System**

The Five-Factor Recall System is Elephantasm's way of deciding what to remember, when, and why. Rather than relying on a single "embedding similarity" or recency heuristic, it models recall as a multi-dimensional weighting problem, where each Memory's chance of resurfacing is shaped by four orthogonal forces:

1. Importance
2. Confidence
3. Recency
4. Decay
5. Similarity

Architecturally, this turns the Memory table into a living index, because each row isn't just data, but a node in a dynamic attention graph. When a query or reflection cycle runs, the Pack Compiler doesn't merely retrieve vectors; it evaluates Memories through this five-factor lens, combining intrinsic properties (importance, confidence) with temporal dynamics (recency, decay). This ensures the system recalls not only what is similar but what is relevant now, trusted, and still alive in mind.

Conceptually, the design intends to mirror human cognition: (1) importance captures salience (what mattered emotionally or strategically); (2) confidence reflects epistemic certainty (what feels true); (3) recency encodes the freshness of experience; (4) decay enforces entropy (the natural fading of neglected thoughts); and (5) similarity measures the Cosine distance between the memory's embedding and the query embedding. Together they form a self-regulating memory ecology: reinforcement slows forgetting, neglect accelerates it. This gives Elephantasm an evolving attentional landscape in and through which memories rise, fade, and resurface, not as static data, but as an adaptive narrative of ongoing relevance.

Example of how Recency and Decay are handled:

```
# Recency (exponential half-life; default 30 days)
age_days = (now - time_end).days
recency_score = 0.5 ** (age_days / HALF_LIFE_DAYS)

# Decay (inverse resistance)
resistance = 0.40*importance + 0.30*confidence + 0.30*recency_score
decay_score = 1.0 - resistance
```

Illustrative weighting applied to all four factors in addition to semantic relevance that impact recall:

```
recall_weight = (
    0.40*semantic_relevance +
    0.25*importance +
    0.15*confidence +
    0.15*recency_score +
    0.05*(1.0 - decay_score)
)
```

Note: It is clear that coefficients / weightings will have to be adjusted for various use cases, and that factors like Recency matter more for immediate search & retrieval, less so once a matching/relevant memory has been identified. For simplicity sake, a simple weighting is shared here and will be fine-tuned during testing and benchmarking.

**iii. Knowledge & Lessons - The Structural Layer of Understanding & Truth**

If Events are what happened and Memories are what it meant, then Knowledge is what remains true. It is Elephantasm's attempt to build structure from story, to lift the agent above the particulars of experience and toward generalisation, prediction, and principle. It is the system's abstraction engine: compressing the flux of experience into reusable form. Every Anima eventually learns that remembering everything is less important than understanding what it means — Knowledge is that understanding formalised.

To preserve coherence across this abstraction process, Knowledge in Elephantasm is explicitly typed. Each Knowledge entry is assigned a knowledge_type — Fact, Concept, Method, Principle, or Experience — giving the system structural guardrails over what kind of understanding may be extracted from a Memory. These types function as cognitive constraints: they ensure that what is distilled respects the nature of the source. A contradictory log from a user cannot become a Principle; a fleeting impression cannot be elevated to a universal Fact. Likewise, each Knowledge item carries a source_type, either EXTERNAL, learned from user-facing Events, or INTERNAL, arising from the system's own reflective processes. INTERNAL knowledge is produced by the Dreamer: the background curator that reinterprets Memories, resolves tensions, and "unlocks" insights through reflection, much like humans do when they sleep on a problem. Together, knowledge_type and source_type allow Elephantasm not just to store understanding, but to reason about the kind of understanding it is.

In Elephantasm's ontology, Knowledge sits above Lessons, which in turn emerge from Memories. A Lesson (or knowledge_audit_log) represents the act of synthesis — the precise link between what was remembered and what was learned. Each entry describes which Memories were distilled, what was inferred, and how those inferences modified the Anima's world model. Knowledge itself is the cumulative architecture of those inferences: the growing lattice of truths, principles, concepts, methods, and experiences that define how an Anima perceives and acts.

***Knowledge - The Structural Layer of Understanding***

Knowledge is not a single undifferentiated substrate. Elephantasm models understanding through five archetypal forms of cognition, captured in the knowledge_type field. These types constrain how Knowledge is extracted and how it may be used:

| Knowledge Type | Essence | Domain | Example |
|---|---|---|---|
| **Fact** | Verifiable truth about the external world | Objective | "Water freezes at 0°C." |
| **Concept** | Abstract framework or model that explains relations | Structural | "Feedback loops regulate systems." |
| **Method** | Causal or procedural understanding of how something works | Practical | "Boil pasta for 8 minutes in salted water." |
| **Principle** | Guiding, normative, or axiomatic belief shaping behavior | Normative | "Simplicity enables clarity." |
| **Experience** | Personal, lived knowledge grounded in first-person perspective | Subjective | "I've lived in London." |
| **Knowledge Type** | Essence | Domain | Example |
| **Fact** | Verifiable truth about the external world | Objective | "Water freezes at 0°C." |

This taxonomy prevents epistemic drift. It forces each Memory-to-Knowledge transformation to answer a concrete question: “What kind of truth is this becoming?”. In doing so, Elephantasm mirrors human cognition, where a procedural insight (“how to do a thing”) and an abstract principle (“why it works that way”) are encoded differently in the mind.

External vs Internal Knowledge
Every Knowledge item is also marked with a source_type:
EXTERNAL — derived from user-facing Events and interactions
INTERNAL — derived from the system’s own reflective processes (Dreamer)

EXTERNAL knowledge mirrors textbook learning: it comes from the user, the world, or observed behaviour. INTERNAL knowledge mirrors self-insight: the result of consolidating patterns across many Memories, resolving contradictions, or recognising latent structure. By enabling INTERNAL knowledge synthesis, Elephantasm gives each Anima the ability to unlock insights that were never stated explicitly - the kind of learning humans achieve not through exposure alone, but through reflection, rest, and reinterpretation. In this sense, the Dreamer’s role is analogous to sleep: a generative, integrative process where the system reorganises experience into deeper truths.

***Lessons - The Interpretive Bridge***
A Lesson represents the reasoning step that connects Memories to Knowledge — an interpretable record of: what was learned, why it was learned, how the Memory justifies the Knowledge that follows.

Conceptually, Lessons replace the idea of a “black-box knowledge update.” Every structural belief the system forms must pass through a Lesson, such that Knowledge never appears ex nihilo; it must always be justified. This is why Lessons align closely with what the implementation currently calls the KnowledgeAuditLog, except here, we treat that audit layer as a first-class epistemic entity, not a passive log.


```
class Knowledge(KnowledgeBase,
TimestampMixin, table=True):

    # Identity
    id: UUID
    anima_id: UUID

    # Content
    knowledge_type: enum
    topic: varchar
    content: str
    summary: str
    source_type: enum

    # Provenance
    memory_id: UUID
```

```
class
KnowledgeAuditLog(KnowledgeAuditBase,

TimestampMixin, table=True):
    id: UUID
    knowledge_id: UUID
    memory_id: UUID
    before: dict[str, Any]
    after: dict[str, Any]
    meta: dict[str, Any]
```

**Knowledge Synthesis**

Knowledge Synthesis is a second compression — a move from narrative to structure. While Memory captures the story of what happened, Knowledge expresses what is generally true because of it.

This transformation follows a consistent sequence:

- Fetch Memory (RLS-aware): the workflow retrieves the source Memory with all associated metadata and narrative.
- Extract generalisable truth: using the configured LLM, the system is prompted to identify:
  - recurring patterns
  - implicit rules
  - stable preferences
  - causal relationships
  - domain knowledge
  - emergent skills or competencies

The goal is to convert specifics into statements that help guide future behaviour.

- Persist Knowledge: the result is stored as a new Knowledge entry:
  - permanently traceable to its Memory,
  - immediately queryable by the Pack Compiler
  - available for future Learning and Identity formation
- Record an audit log: any refinement triggers an immutable audit snapshot, turning the evolution of knowledge into an inspectable chain of epistemic reasoning.

Knowledge Synthesis is triggered automatically after each Memory creation, but can also be triggered manually via API or periodically by the Dreamer loop for retrospective consolidation. This ensures Knowledge grows in step with experience, not just at isolated checkpoints.

**Why Knowledge Matters**

Knowledge is where Elephantasm transitions from reactive continuity to strategic understanding. It stabilises patterns that would otherwise remain fleeting in memory and forms the backbone for:

- preference inference
- planning
- decision heuristics
- self-reflection
- model steering
- long-term alignment

As such, the Knowledge model becomes the substrate through which Identity can meaningfully evolve. Where Events describe the world, and Memories describe experience, Knowledge describes the self's understanding of the world, a foundation upon which coherent, agentic behaviour can be built. In this sense, Knowledge is not the end of learning, but its crystallised middle: the structured scaffolding on which deeper interpretations (Lessons) and enduring self-narratives (Identity) will form.

**iv. Identity - The Emergent Persona Layer**

If Events are what happened, Memories what it meant, and Knowledge what became true because of it, then Identity is the answer to a different question altogether: who changed as a result of those truths? It is the apex compression of Elephantasm's cognitive hierarchy, the layer where understanding becomes selfhood. While Knowledge captures what the system knows, Identity captures the system that knows it and is what lets an Anima respond today in a way that still resembles who it was yesterday. No long-term agentic system can function without such an anchor.

***The Nature of Identity***

Identity in Elephantasm is intentionally minimal in structure, but vast in expressive capacity. Each Anima has exactly one Identity: a dedicated locus where its dispositional traits, self-reflections, and communicative tendencies are preserved. The model contains only three explicit fields:

Personality Type

A coarse-grained orientation (e.g. MBTI) that frames how the system interprets and expresses itself, a starting shape from which nuance can evolve.

Self Narrative (self)

A JSON-encoded internal map of meaning: reflections, principles, values, non-negotiables, inferred competencies, behavioural tendencies, policy statements, orientations, and contradictions the Dreamer has resolved over time. This is the beating heart of Identity: a malleable, introspective surface where experience becomes self-understanding. Because it is unopinionated and schema-light, it can grow without structural migrations, allowing Identity to deepen naturally as the agent accumulates context.

Communication Style

A short descriptor guiding tone and expression. If Personality describes disposition, communication_style describes voice. It shapes how the system sounds without overriding who it is. While arguably a sub-set of Identity and not a foundational pillar, this field is included for practical reasons, in particular while we're dealing with language models.

Together, these fields form the "public surface" of Identity. Everything else lives inside self, where the Dreamer can reason, revise, and reinterpret without constraint.

**Identity as the Interpretive Lens**

Most memory architectures end with storage, but we've decided it must end with perspective; as such, Identity is more than the product of experience, but the lens through which future experience is understood i.e. it is the interpretive kernel that recursively shapes the entire cognitive loop of the system.

Every Event, every Memory, every piece of Knowledge, every Dreamer reflection passes through this lens. It colours perception, guides interpretation, modulates importance, resolves contradictions, and steers synthesis. In this way, Identity acts as the central recursive force that gives coherence to the entire stack.

Where other systems retrieve what is stored, *Elephantasm retrieves as someone*. Identity creates this asymmetry:

- Two Anima with identical histories can still diverge, because their Identities differ.
- The same Event may trigger different Memories, depending on the persona interpreting it.
- Knowledge synthesis is weighted by who the system has become, not only by what was observed.
- Dreamer reinterpretation uses Identity as its constraint, the way a worldview constrains human introspection.

Identity therefore acts as the kernel of the Anima, a persistent, self-modifying attractor state from which perception, reasoning, and behaviour emerge.

This feedback loop is deliberate:

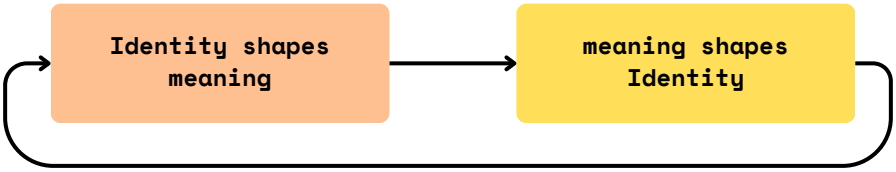
1. Identity → shapes how Experiences become Memories: what the system considers important, surprising, contradictory, aligned, or meaningful depends on the persona interpreting the Events.

2. Identity → shapes how Memories become Knowledge: the extraction of principles, lessons, patterns, and truths is filtered through the system's current dispositions and self-understanding.

3. Identity → shapes how Knowledge is retrieved: relevance is not merely semantic; it is personal. The Pack Compiler retrieves what is meaningful to this Anima, not abstractly relevant.

4. Identity → shapes Dreaming (Reflection & Re-evaluation): the Dreamer revises Memories and Knowledge in light of who the Anima now is, just as humans reinterpret past events when their values or beliefs shift.

5. Identity → is revised by the very interpretations it produces: this creates a controlled, recursive loop, making this model our simpliest but also our most sophisticated and meaningful one, performing as a living kernel rather than a static profile.




***Identity as the Stability Mechanism of the Whole System***

Without Identity, experience would accumulate without direction, and retrieval would drift with each query. With Identity, Elephantasm gains:

- temporal coherence — the past and present are interpreted consistently
- stance — the agent has a point of view, not just a memory
- bias — not in the harmful sense, but as a coherent interpretive frame
- continuity — actions map back to a stable self-model
- integrity — long-term behaviour becomes predictable and narratively traceable

Identity is therefore the organising principle of Elephantasm's cognition. Everything that enters the system passes through Identity. Everything that leaves the system is shaped by Identity. And Identity itself evolves in response to everything it shapes. This recursive structure is the core differentiator of Elephantasm: memory not as a warehouse, but as a worldview.

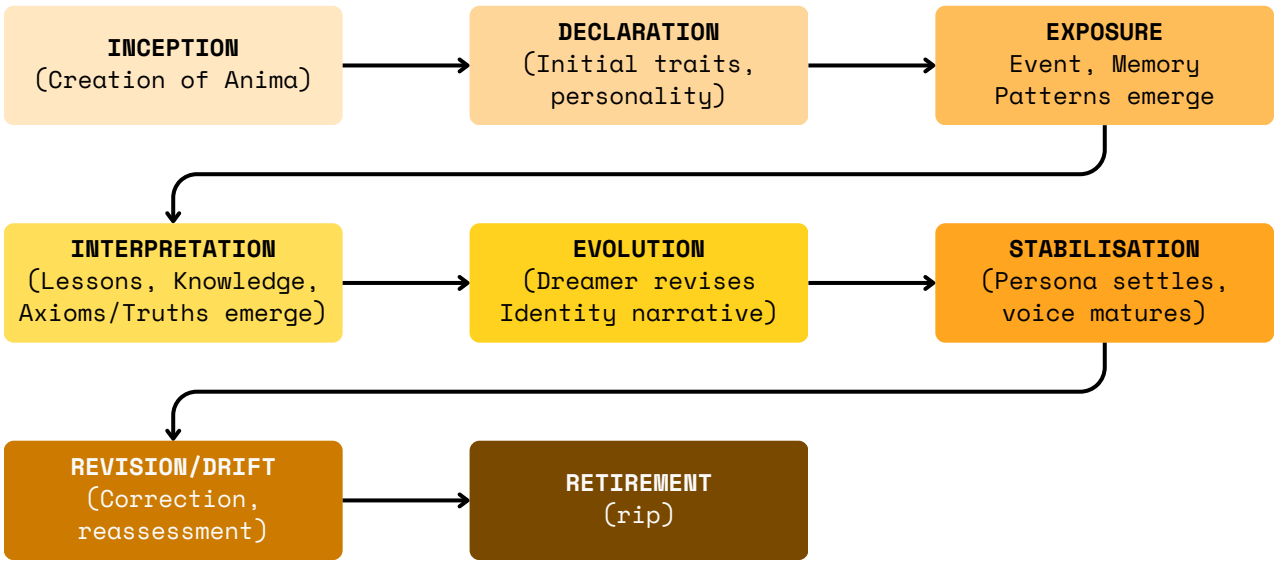
**Provenance and the Identity Audit Chain**

Every modification to Identity passes through an append-only log, the IdentityAuditLog, which captures:

- the action (CREATE, UPDATE, ASSESS, EVOLVE)
- the trigger source (api_update, dreamer-nightly, etc.)
- the Memory that influenced the change (if any)
- the complete before-state and after-state
- an optional human-readable summary describing why the change occurred

**How Identity Evolves**

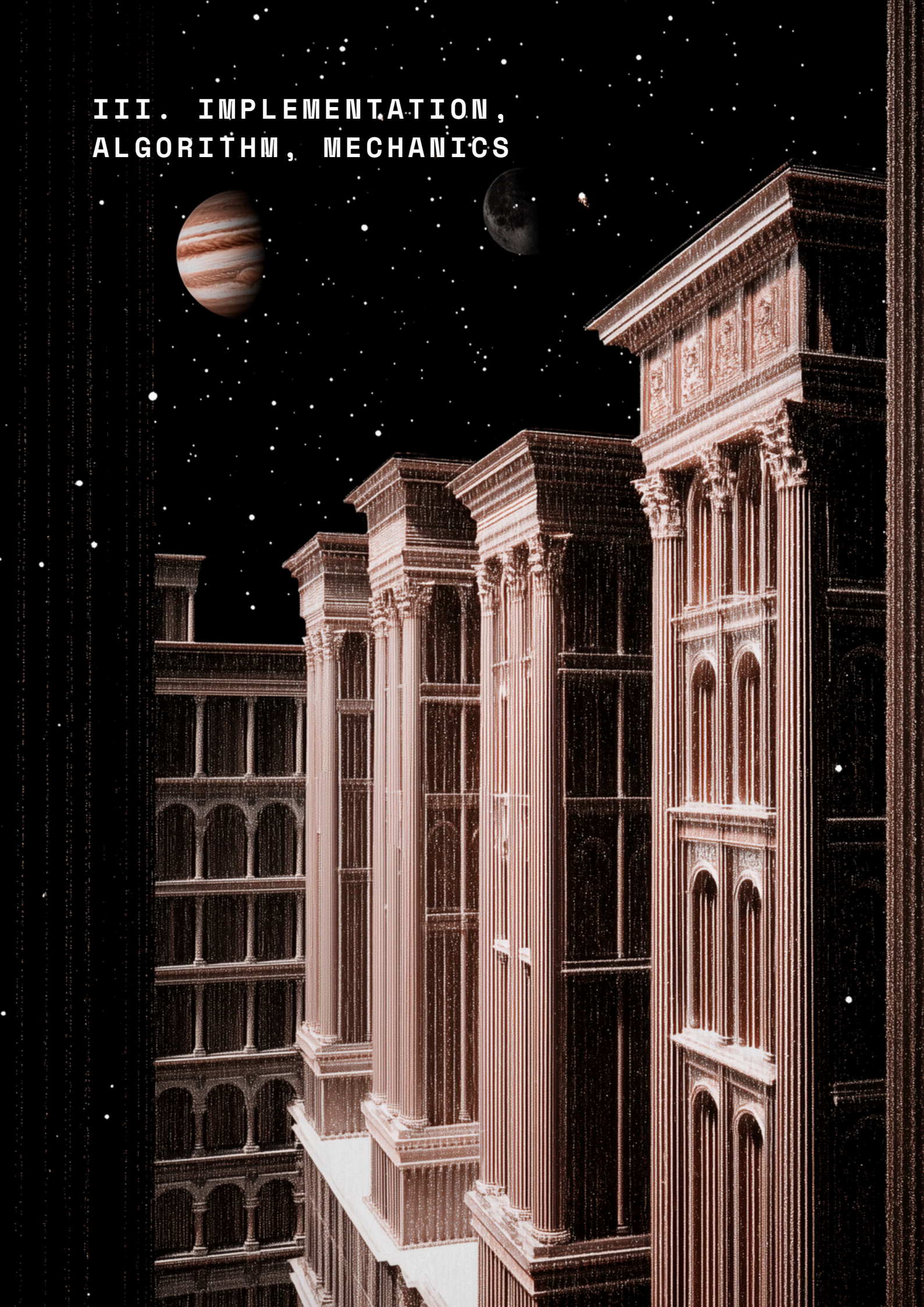
Identity is shaped through a sequence of stages that mirror the architecture beneath it:




Identity is therefore not merely the terminus of Elephantasm's cognitive hierarchy, but also its interpretive centre of gravity. It is the slowest-changing layer yet (in many ways) the most influential, the mechanism that ensures every act of perception, synthesis, and reflection occurs through a coherent, evolving worldview. Because Identity both shapes and is shaped by experience, the system gains something no retrieval-based architecture can approach: *continuity with perspective*. It remembers not only what happened, but what it meant to this Anima — and how it should matter going forward.

As the system matures, Identity becomes the reinforcement loop that turns scattered experiences into a stable persona. In this sense, Identity is the culmination of Elephantasm's design philosophy: memory not as an archive, knowledge not as abstraction, but a living self that persists, evolves, and interprets the world through the lens of everything it has lived, learned, and become.

# III. IMPLEMENTATION, ALGORITHM, MECHANICS



# // DREAM FUNCTION

A system that accumulates experience indefinitely simply drowns in a flood of irrelevant and outdated context. The Dreamer is Elephantasm's answer to this problem: a background curation loop that decays, merges, refines, and forgets, ensuring memories evolve rather than accumulate.

Human memory doesn't work by storing everything and hoping retrieval stays useful. Things fade. Similar moments blend together. Vague impressions either sharpen or quietly disappear. Most of this maintenance happens not while we're actively thinking, but during sleep, a period of consolidation where the mind reorganizes what it has gathered.

The Dreamer borrows this metaphor. It wakes on a schedule, examines what has accumulated since the last pass, and decides what still deserves to exist in its current form.

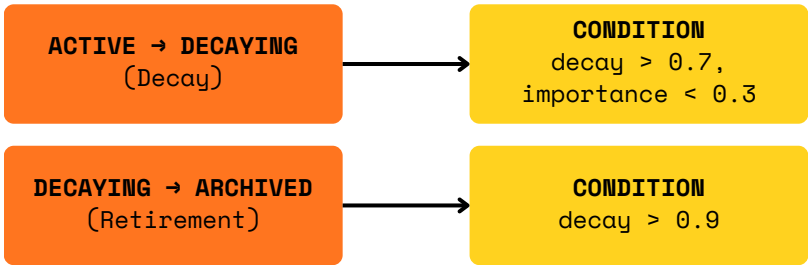
**Light Sleep: Algorithmic Processing**
The Dreamer operates in two distinct phases, mirroring the structure of biological sleep cycles.

1. Decay Score Updates
Every memory ages. The decay score quantifies staleness using a half-life formula:

```
decay_score = min(1.0, age_days / half_life_days)
```

At the configured half-life (default: 30 days), a memory reaches a decay score of 0.5. This score increases monotonically until the memory is reinforced or curated.

2. State Transitions
Memories exist in one of three states: ACTIVE, DECAYING, or ARCHIVED. Light Sleep enforces transitions based on thresholds:




These transitions are conservative. A memory must be both stale *and* unimportant to begin fading. High-importance memories persist regardless of age.

3. Merge Candidate Detection
Light Sleep identifies potentially redundant memories using two strategies:

- Primary: pgVector cosine distance on embeddings (threshold: 0.3)
- Fallback: Jaccard word similarity for memories without embeddings (threshold: 0.6)

Candidate groups are passed to Deep Sleep for semantic evaluation i.e. Light Sleep only flags, it doesn't decide.

4. Review Candidate Flagging
Memories are flagged for Deep Sleep review if they exhibit uncertainty signals:

- Low confidence (< 0.4)
- Very short summary (< 20 characters)
- Recently created (since last dream)

Light Sleep reduces the problem space. It ensures the system isn't asking a model to reason about thousands of memories at once.

**Deep Sleep: LLM-Powered Curation**
Deep Sleep is where reflection happens. It processes the candidates flagged by Light Sleep in small batches, using an LLM to make nuanced decisions.

Merge Processing
For each candidate group, the LLM evaluates:
1. Do these memories genuinely overlap, or is similarity coincidental?
2. If merging, what unified summary captures both?
3. What importance and confidence should the merged memory inherit?

Approved merges create a new memory with provenance metadata (merged_from: [source_ids]). Original memories are soft-deleted but their event links remain intact for audit.

Review Processing
For flagged memories, the LLM chooses one of four actions:

| Action | Description |
|---|---|
| KEEP | No changes needed |
| UPDATE | Refine summary, adjust importance/confidence |
| SPLIT | Separate conflated concepts into distinct memories |
| DELETE | Mark as noise (soft-delete with reasoning) |

The default bias is conservative. When uncertain, the Dreamer keeps things as they are. Memory systems shouldn't be aggressive. If a model freely rewrites its past every few hours, instability replaces learning.

**Dream Session Lifecycle**
Each execution of the Dreamer produces a Dream Session, which represents a complete record of one curation cycle.

Session Metadata:
- trigger_type: SCHEDULED (periodic) or MANUAL (API-triggered)
- triggered_by: User ID for manual triggers
- config_snapshot: Frozen configuration at execution time (reproducibility)
- started_at / completed_at: Timing boundaries

Session Metrics:
- memories_reviewed: Total memories examined
- memories_modified: Memories updated (includes merge sources)
- memories_created: New memories from merges/splits
- memories_archived: Transitioned to DECAYING/ARCHIVED
- memories_deleted: Soft-deleted as noise

Only one dream runs per Anima at a time. Concurrency is prevented at both the application layer (in-memory locks) and database layer (status checks). If a dream fails mid-execution, partial progress is preserved, so the actions that completed remain in the audit trail.

**Dream Actions: The Audit Trail**
Every mutation the Dreamer makes is recorded as a Dream Action. This is the mechanism by which Elephantasm maintains epistemic integrity over self-modification.

| Action | Phase | Description |
|---|---|---|
| MERGE | Deep Sleep | Combined multiple memories into one |
| SPLIT | Deep Sleep | Divided one memory into multiple |
| UPDATE | Both | Modified memory fields |
| ARCHIVE | Both | Transitioned state toward archived |
| DELETE | Deep Sleep | Soft-deleted as noise |

Each action captures:
action_type: MERGE | SPLIT | UPDATE | ARCHIVE | DEL
phase: LIGHT_SLEEP | DEEP_SLEEP
source_memory_ids: [UUIDs of input memories]
result_memory_ids: [UUIDs of output memories] | NULL
before_state: { full snapshot of sources}
after_state: { full snapshot of results } | NULL
reasoning: “LLM explanation” | NULL
created_at: immutable timestamp

This structure enables:
- Rollback: Any action can theoretically be reversed using snapshots
- Debugging: Trace exactly why a memory changed
- Trust: A system that edits its own past without leaving a trail becomes impossible to verify

**Scheduling and Triggers**
The Dreamer runs on a fixed cadence, every 12 hours by default, mimicking the rhythm of human sleep.

This cadence is necessary, because Memories need time to accumulate before curation is useful. Running the Dreamer constantly burns compute without improving outcomes. Running it too rarely lets clutter harden into structure. Twice daily strikes a balance for most use cases.

**Identity-Aware Curation**
Memory decisions aren't neutral. What counts as redundancy or relevance depends on who the system is becoming, not just what it once observed.

The Dreamer curates through the lens of the Anima's Identity — its name, description, principles, and epistemology. This context is injected into every LLM prompt:

```
IDENTITY CONTEXT (the agent's personality and values):
- Name: [Anima name]
- Description: [What the Anima is/does]
- Principles: [Core values, up to 5]
- Epistemology: [How it knows things]

EVALUATION CRITERIA:
1. Semantic Overlap
2. Complementary Information
3. Temporal Coherence
4. Identity Relevance: Is this equally important to the agent's purpose?
```

The fourth criterion is key. The LLM doesn't ask "is this useful?" in the abstract, it asks "is this useful *for this agent*?"

Example: A research assistant Anima and a personal companion Anima might both receive the same memory about a user's preference. For the companion, this is high-importance identity-relevant data. For the research assistant, it might be noise. The Dreamer, guided by Identity, handles them differently.

This filter ensures curation isn't arbitrary. Old goals fade when priorities change. Preferences that matter to one agent might be irrelevant to another. The Dreamer's decisions remain grounded in who the system is becoming.

**The Dream Journal**
Every Dream Session produces a complete journal — the session record plus all associated actions. This journal answers questions like:

- What did the Dreamer examine?
- What did it change, and why?
- What did memories look like before and after?
- How has the memory landscape evolved over time?

The journal is the mechanism by which operators (and eventually the system itself) can understand and trust the curation process.

**Dreamer Configuration**
The Dreamer's behavior is tunable via configuration:

| **Parameter** | **Default** | **What it Controls** |
| --- | --- | --- |
| decay_half_life_days | 30 | Number of days until the decay score reaches 0.5 |
| decay_threshold | 0.7 | Decay score for ACTIVE to DECAYING |
| archive_threshold | 0.9 | Decay score for DECAYING to ARCHIVED |
| importance_floor | 0.3 | Below this importance memory becomes archive candidate |
| confidence_review_threshold | 0.4 | Flags low-confidence memories for review |
| min_summary_length | 20 | Flags summaries that are too short |
| embedding_similarity_threshold | 0.3 | pgVector cosine distance to detect merge candidates |
| llm_provider | openai | LLM provider used during Deep Sleep curation |
| llm_model | gpt-4o-mini | Model used for curation and decision-making |
| llm_temperature | 0.3 | Low temperature to ensure deterministic outputs |
| curation_batch_size | 10 | Number of memories processed per LLM call |
| regenerate_embeddings | true | Regenerates embeddings after content changes |

# // THE PACK COMPILER

A Memory Pack is a structured payload injected into an LLM's system prompt before each interaction. It is a curated worldview, assembled according to explicit rules, that grounds the model's response in something beyond the immediate query.

The pack answers four questions simultaneously:
1. Who is this agent? (Identity)
2. What are we talking about right now? (Session context)
3. What does the agent know to be true? (Knowledge)
4. What has the agent experienced that matters here? (Long-term memory)

Each question maps to a retrieval layer with its own strategy, scoring function, and purpose. Together, they form a hierarchy: who → now → truth → history.

Every Memory Pack braids four strata into a single artifact:

**Layer I: Identity**
The persona anchor. This layer fetches the Anima's personality type, communication style, and self-narrative i.e. the living document where reflections, principles, and dispositions accumulate.

Identity is retrieved statically. It must always be present, because without it, the agent would sound like a stranger each turn. Identity acts as the interpretive lens through which everything else gains meaning.

The identity layer typically consumes 70–150 tokens, presenting a condensed prose summary:

```
You are Atlas, a research assistant with an analytical communication style.
Your core belief: knowledge compounds through systematic inquiry.
You value precision over speculation and prefer to acknowledge uncertainty
rather than fabricate confidence.
```

This injection happens first, establishing the agent's voice before any content arrives.

**Layer II: Session Memories**
The conversational thread. These are recent memories filtered by a time window (defaulting to 24 hours), scored purely by recency. They carry the breadcrumb trail of the current conversation without invoking semantic search.

The goal is simply to maintain continuity within the immediate interaction. The agent should not repeat itself, lose the plot, or forget what was just discussed. Session memories are scored using exponential decay with a half-life of approximately one day:

```
recency_score = 2^(-age_in_days / half_life)
```

A memory from an hour ago scores near 1.0; a memory from a week ago approaches zero. This ensures conversational context stays fresh without requiring semantic relevance to the current query.

**Layer III: Knowledge**
Durable truths. This layer uses vector similarity to surface knowledge items semantically relevant to the current query. Knowledge is scored by a blend of confidence (how certain the system is of its accuracy) and similarity (how closely it matches the query embedding):

```
knowledge_score = (confidence × 0.5) + (similarity × 0.5)
```

No recency, no decay. Knowledge items are meant to be stable truths—facts, principles, methods, and experiences that have been extracted and validated over time. What matters is relevance to the query and certainty of accuracy.

Knowledge is classified by type: FACT, CONCEPT, METHOD, PRINCIPLE, EXPERIENCE. Different retrieval presets can filter by type — a how-to question might prioritise METHOD knowledge; a personal question might favour EXPERIENCE.

**Layer IV: Long-Term Memories**
Deep recall. Older memories that clear a semantic similarity threshold enter a five-factor scoring gauntlet. These are subjective recollections elevated because the Anima keeps reinforcing them — experiences that, through repeated access or high salience, have resisted forgetting.

The five factors are:

1. Importance (0–1): How significant the memory is, as assessed during synthesis.
2. Confidence (0–1): How certain the system is of its accuracy.
3. Recency (0–1): Freshness, computed via exponential decay with a thirty-day half-life.
4. Decay (0–1): Forgetting resistance, modelled on spaced repetition—frequently accessed memories decay slower.
5. Similarity (0–1): Cosine distance between the memory's embedding and the query embedding.

These combine into a weighted formula:

```
score = (importance × w_i) + (confidence × w_c) + (recency × w_r)
      + ((1 - decay) × w_d) + (similarity × w_s)
```

Default weights distribute roughly evenly (importance 0.25, similarity 0.25, recency 0.20, confidence 0.15, decay 0.15), but these are tuneable per preset or per query. A conversational preset might bias toward recency; a research preset might swing toward importance and similarity.

**The Scoring Algorithms**
Retrieval is not fuzzy matching. Every inclusion is a defendable decision, traceable to explicit factors.

**Recency Score**
Time-based freshness using exponential decay:

```python
def compute_recency_score(memory_time, reference_time, half_life_days=7.0):
    age_days = (reference_time - memory_time).total_seconds() / 86400
    return 2 ** (-age_days / half_life_days)
```

At t=0 (just created), the score is 1.0. At t=half_life, the score is 0.5. At t=2×half_life, the score is 0.25. The curve is smooth, predictable, and parameterised.

**Decay Score**
Forgetting resistance using spaced repetition principles. Memories that are accessed frequently develop longer effective half-lives:

```
def compute_decay_score(memory_time, last_accessed, access_count,
                        base_half_life_days=30.0, access_boost_factor=1.5):
    effective_half_life = base_half_life_days * (access_boost_factor ** access_count)
    decay_rate = math.log(2) / effective_half_life
    age_days = (now - memory_time).total_seconds() / 86400
    return 1 - math.exp(-decay_rate * age_days)
```

A memory accessed ten times has a much longer effective half-life than one never revisited. The decay score represents how much the memory has faded; we use (1 - decay) in the final formula because high decay is bad.

**Combined Score**
The final retrieval score is a linear combination of all factors:

```
def compute_combined_score(importance, confidence, recency_score,
                           decay_score, similarity, weights):
    return (
        importance * weights.importance +
        confidence * weights.confidence +
        recency_score * weights.recency +
        (1 - decay_score) * weights.decay +
        similarity * weights.similarity
    )
```

Each scored memory carries a breakdown showing exactly why it entered the pack.

**Token Budget as Constitution**
Context windows remain finite. The Pack Compiler enforces a token budget as a constitutional constraint that shapes what can be remembered in any given turn.

The priority order is strict:
1. Identity — Never sacrificed. Fixed allocation (~150 tokens).
2. Session Memories — 25% of remaining budget.
3. Knowledge — 35% of remaining budget.
4. Long-Term Memories — 40% of remaining budget.

If the budget overflows, long-term memories are trimmed first, then knowledge, then session memories. Identity is inviolable. This ensures the agent maintains coherence in the present moment even when historical depth must be compressed.

Within each layer, items are kept in score order until the budget exhausts. Token estimation uses a conservative heuristic (~4 characters per token), erring on the side of caution. The result is a pack that fits within the LLM's context window while preserving the most relevant content from each stratum.

More work needs to be done to optimise this process, however; continuous re-injection of the entire payload is effective but very inefficient and costly. Depending on where in the in-thread context window the agent is, each payload becomes part of the LLMs built-in context window “memory” i.e. short-term memory. Optimising around this should allow us to heavily trim the need for ongoing reinforcement of knowledge, context and memories it already has.

**Retrieval Presets**
Elephantasm ships with two preset retrieval strategies, representing different philosophies of agent autonomy:

Conversational Preset
Fully deterministic, zero additional LLM calls.

```
conversational_preset = {
    "session_window_hours": 4,
    "max_session_memories": 5,
    "max_knowledge": 3,
    "max_long_term_memories": 3,
    "max_tokens": 2000,
    "weights": {
        "recency": 0.35,      # Favour recent
        "similarity": 0.30,   # Still relevant
        "importance": 0.20,
        "confidence": 0.10,
        "decay": 0.05
    }
}
```

This preset prioritises speed and predictability. The agent retrieves what's immediately relevant without deliberation, ideal for chat interfaces where latency matters and context is narrow.

Self-Determined Preset
The agent chooses its own retrieval strategy.

Before compilation, the system makes a lightweight LLM call, passing the user's query and asking the model to decide:

- Which knowledge types are relevant?
- How many long-term memories should be retrieved?
- What importance threshold should apply?
- What scoring weights make sense for this question?

```
# LLM prompt excerpt
"""
Given the user's query, determine optimal retrieval parameters:
- Factual questions → high knowledge, high similarity weight
- Personal questions → importance weighted, experience-focused
- Recent events → high recency weight, more long-term memories
- How-to questions → METHOD type, higher max_knowledge
"""
```

The self-determined preset adds ~500–1000ms latency but enables adaptive retrieval. A personal question might favour EXPERIENCE knowledge; a technical query might prioritise METHOD. The agent, in a sense, decides what kind of memory it needs before remembering.

Both presets use the same deterministic compiler; the only difference is who configures it. This separates retrieval strategy from retrieval execution.

**Why Determinism Matters**
The Pack Compiler is deterministic by design: the same configuration and database state will produce the same pack, every time. No randomness, no sampling, no LLM involvement in the retrieval loop itself.

This matters for three reasons:

Reproducibility
When debugging agent behaviour, you need to know exactly what context was injected. A deterministic system lets you replay a pack, inspect its composition, and trace why the agent responded as it did. Stochastic retrieval would make this impossible.

Trust
Agents making consequential decisions must be auditable. If a financial advisor agent gives advice, regulators may ask: what did it know when it said that? Memory Packs can be persisted, versioned, and inspected. The provenance chain from event to memory to pack to response becomes traceable.

Coherence
Randomness in retrieval introduces drift. An agent might surface contradictory memories on successive turns, undermining the narrative consistency that defines identity. Determinism ensures that, given the same state, the agent presents the same self.

Importantly, however, the LLM remains free to interpret, reason, and respond creatively within the context it receives. We constrain what enters the prompt; we do not constrain what emerges from it.

**Observability and Persistence**
Continuity without visibility is dangerous. An agent that remembers opaquely is an agent you cannot trust or debug.

When persistence is enabled, each pack is saved asynchronously via a fire-and-forget background task. The response returns immediately; persistence happens in a separate thread. The stored pack includes:

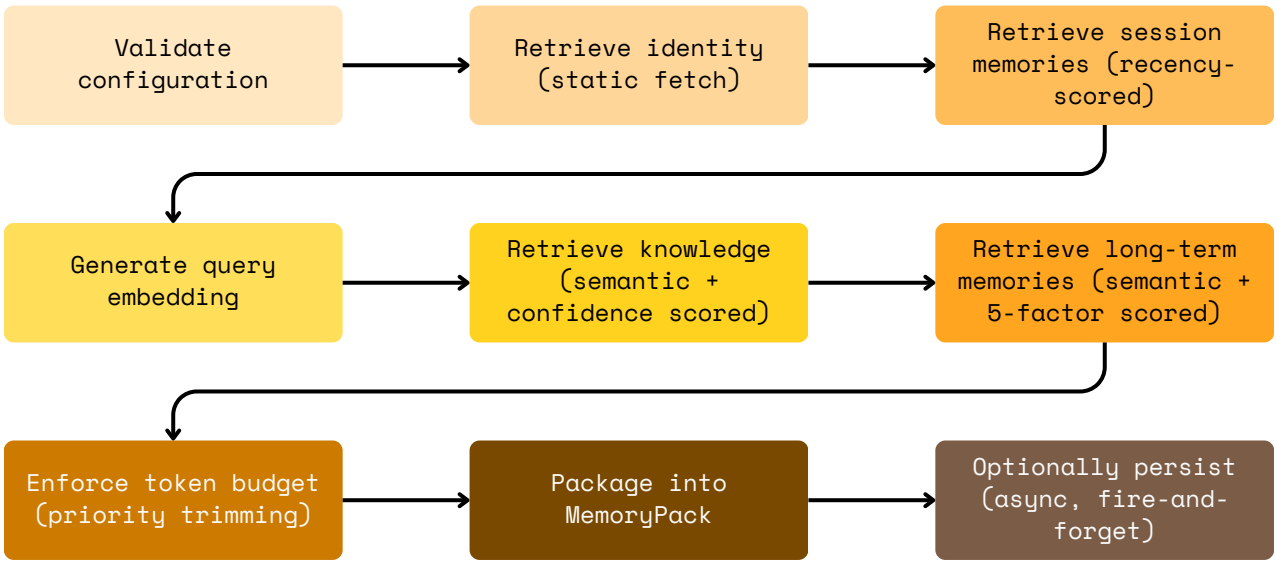
- The full context string (what was injected)
- All four layers with their scores and breakdowns
- The retrieval configuration snapshot
- Token counts per layer
- Compilation timestamp

```
class MemoryPack:
    id: UUID
    anima_id: UUID
    query: Optional[str]
    preset_name: str

    # Layer counts
    session_memory_count: int
    knowledge_count: int
    long_term_memory_count: int
    has_identity: bool

    # Budget tracking
    token_count: int
    max_tokens: int

    # Full content (JSONB)
    content: {
        "context": str,          # The injected prompt
        "identity": {...},
        "session_memories": [...],
        "knowledge": [...],
        "long_term_memories": [...],
        "config": {...}          # Retrieval config used
    }

    compiled_at: datetime
```

A retention policy caps storage at 100 packs per Anima, automatically pruning the oldest. This creates a sliding window of observability: you can always inspect recent context injections without unbounded storage growth.

The API exposes pack history, latest pack, and aggregated statistics — token usage, identity inclusion rate, layer distributions — giving operators a dashboard into the agent's evolving context strategy.

The compilation pipeline follows a strict sequence:




Each step is synchronous and stateless. The compiler receives a configuration, queries the database, applies scoring algorithms, and returns a pack.

**Temporal Awareness**
Memory Packs include a temporal layer: the current date, time since last interaction, and session duration. This grounds the agent in when it is, not just who it is.

When the session memory layer is empty (no recent interactions), the temporal layer provides explicit context:

```
Current date: January 21, 2026.
Your last interaction with this user was 3 days ago.
```

This prevents the uncanny valley of an agent that seems aware of its memories but oblivious to the passage of time.

**Future Directions & Further Development**
The current implementation is intentionally simple. Several refinements are planned:

Differential injection
Track what was sent in the previous pack and only inject what changed. If identity and session context are stable across turns, send only new knowledge and long-term recalls. This reduces token usage while preserving coherence.

Drift-aware scoring
Boost memories that contradict the agent's current state or recent responses. If the agent has been asserting X, and a memory records evidence of not-X, that memory becomes more salient. This surfaces contradictions rather than burying them.

Dreamer-informed weights
Allow the Identity layer to bias retrieval based on how the Anima wants to evolve. If the agent's self-narrative includes a goal of "becoming more empathetic," retrieval might weight EXPERIENCE memories higher. Identity stops being passive context and becomes an active shaper of what gets remembered.

Self-improving scoring
Observe which retrieved memories actually influenced responses (via feedback loops or outcome tracking) and adjust weights accordingly. The goal is an agent that learns not just what to remember, but how to remember, optimising its own retrieval strategy over time.

# // MEMORY SYNTHESIZER

Memory Synthesis is the generative engine of Elephantasm that transforms raw experience into retained meaning. Where Events record what happened, Memory Synthesis decides what mattered.

The system reads a window of Events, weighs their significance, and authors a Memory that distills their essence into something that can be carried forward. Every Memory is an act of narrative construction: a claim that these particular experiences deserve to persist.

The architecture is deliberately simple: a workflow with five nodes, a three-part trigger system, and an accumulation scoring formula that decides when synthesis should occur.

**The Accumulation Model**
Experience must accumulate before synthesis becomes worthwhile. Too frequent, and you burn compute generating trivial memories. Too rare, and experiences pile up until the synthesis window becomes unwieldy.

The accumulation score answers a simple question: Has enough happened to justify a new memory?

Three factors contribute:

1. Time (1 point per hour since last synthesis)
2. Events (0.5 points per new event)
3. Tokens (1 point per 3,333 tokens of content)

The default threshold is 10.0. Synthesis triggers when the score crosses this boundary.

Example triggers:

- 10 hours with no events → 10.0 (triggers on time alone)
- 5 hours with 10 events → 10.0 (balanced mix)
- 2 hours with 5 events and 10,000 tokens → 10.0 (content-heavy session)

The formula is intentionally coarse. It doesn't try to detect "important" events, as that is the LLM's job during synthesis. The accumulation score simply ensures we don't ask the LLM to synthesize until there's enough raw material to work with.

**Per-Anima Configuration**
Each Anima carries its own synthesis profile, stored in synthesis_configs:

```
class SynthesisConfig:
    time_weight: float      # 0.0–5.0, default 1.0
    event_weight: float     # 0.0–2.0, default 0.5
    token_weight: float     # 0.0–0.001, default 0.0003
    threshold: float        # 1.0–50.0, default 10.0
    temperature: float      # 0.0–1.0, default 0.7
    max_tokens: int         # 256–4096, default 1024
```

A high-traffic Anima might raise its threshold to avoid over-synthesizing. A journaling Anima might lower it to capture quieter moments. The weights and threshold are dials that shape how the Anima decides when experience becomes memory.

On first access, the system seeds a config from environment defaults. Tenants can read and modify their configs via API, but RLS ensures isolation.

**The Workflow Architecture**
Our first version of Memory Synthesis is implemented as a LangGraph StateGraph with five nodes:

```
START
  ↓
calculate_accumulation_score
  ↓
check_synthesis_threshold
  ↓ (conditional routing)
  ├─ triggered=True  → collect_pending_events → synthesize_memory → persist_memory → END
  └─ triggered=False → END (skip_synthesis)
```

Node Responsibilities

1. calculate_accumulation_score — Computes the three-factor score by querying the database for event counts and timestamps. Sync operation, runs in FastAPI's thread pool.

2. check_synthesis_threshold — Pure function. Compares score to threshold, routes to either synthesis or skip. No side effects, no I/O.

3. collect_pending_events — Fetches all Events since the last synthesis window, serializes them for the LLM prompt. Sync operation.

4. synthesize_memory — The LLM call. Async operation. Sends Events to the configured model (Claude or GPT) with a structured prompt, parses JSON response.

5. persist_memory — Creates the Memory record, links it to source Events via MemoryEvent provenance records, generates vector embedding. Atomic transaction — either everything commits or nothing does.

**Trigger Mechanisms**
Memory Synthesis runs via three trigger modes, each serving a different operational need.

Real-Time Trigger (Event-Driven)
When an Event is created, the API handler schedules a background check:

```
background_tasks.add_task(
    scheduler.check_and_enqueue_if_needed,
    event.anima_id
)
```

The check calculates the accumulation score without running the full workflow. If the score exceeds threshold, a synthesis job is enqueued with a 5-second delay.

The delay serves the purpose of batching. When a user sends several messages in quick succession, each Event write triggers a check. The delay ensures we don't spawn redundant synthesis jobs. APScheduler's replace_existing=True naturally deduplicates jobs with the same ID.

Scheduled Trigger (Periodic)
A background job runs hourly (configurable via SYNTHESIS_JOB_INTERVAL_HOURS), executing synthesis for all active Animas:

```
async def execute_for_all_animas() -> Dict[str, int]:
    # Fetches up to 1000 active animas
    # Runs synthesis in parallel
    # Returns: {successful, failed, skipped, items_created}
```

This catches Animas that accumulated experience but never crossed the real-time threshold i.e. long periods of inactivity followed by a burst of events.

Manual Trigger (API)
Operators can force synthesis via the scheduler API:

```
POST /scheduler/workflows/memory_synthesis/trigger
{
  "anima_id": "uuid"  // or null for all animas
}
```

Manual triggers bypass the accumulation check. If there are Events to synthesize, they will be synthesized. This is useful for debugging, demos, or when you want immediate memory creation without waiting for thresholds.

Each trigger mode tags its runs with a trigger_source label ("realtime", "scheduled", "manual") for observability in LangSmith tracing (or alternatives).

**Provenance and Persistence**
Every Memory maintains an auditable link to its source Events via the MemoryEvent junction table:

```
class MemoryEvent:
    memory_id: UUID
    event_id: UUID
    link_strength: float  # Default 1.0
```

When synthesis completes, the persist node creates the Memory and all MemoryEvent links in a single atomic transaction:

```
with session_with_rls_context(user_id) as session:
    memory = _create_memory(session, anima_id, llm_response, pending_events)
    session.flush()  # Get ID before linking

    links = _create_provenance_links(session, memory.id, pending_events)
    session.flush()

    embedding_generated = _generate_embedding(session, memory.id)
    # Auto-commit on context exit
```

If any step fails, the entire transaction rolls back. You never get a Memory without provenance or provenance pointing to a non-existent Memory. The Memory record captures temporal span from the source Events: time_start (Earliest event timestamp) and time_end (Latest event timestamp). This allows temporal queries: "What did the Anima experience between Tuesday and Thursday?" becomes a simple range query on time_start and time_end.

After persistence, the system generates a vector embedding (OpenAI text-embedding-3-small, 1536 dimensions) for semantic search. Embedding generation is best-effort i.e. if it fails, the Memory still persists, and embedding can be backfilled later.

The embedding enables semantic retrieval during pack compilation and merge candidate detection during Dreamer curation.

**Knowledge Synthesis Cascade**
A successful Memory Synthesis automatically triggers Knowledge Synthesis. This keeps the downstream pipeline aligned. When new Memories are created, the Knowledge layer has the opportunity to extract durable truths. The cascade is fire-and-forget i.e. Memory Synthesis doesn't wait for Knowledge Synthesis to complete.

The result is a continuous flow: Events → Memories → Knowledge, each layer transforming experience into progressively more stable forms of understanding.

**Overall Design Philosophy: Memory Synthesis**
Memory Synthesis embodies several Elephantasm principles:

Accumulation before curation
We don't try to decide in real-time whether an Event is "important enough" to become a Memory. We accumulate everything, then periodically synthesize. The LLM makes the importance judgment with full context.

Determinism where possible
The accumulation formula is deterministic. The trigger logic is deterministic. Only the LLM call introduces non-determinism — and even that is bounded by temperature and constrained by structured output.

Provenance as first-class
Every Memory links to its source Events. You can always answer "Why does the Anima believe this?" by tracing provenance. This is essential for trust and debugging.

Per-Anima autonomy
Different Animas have different synthesis profiles. A research assistant might need tight, factual memories; a creative collaborator might need looser, more interpretive synthesis. The config system makes this explicit and controllable.

Graceful degradation
If the LLM call fails, the Events remain. If embedding generation fails, the Memory persists without vectors. If Knowledge Synthesis fails, Memories still exist. Each layer fails independently.

# // KNOWLEDGE SYNTHESIZER

If Memory Synthesis asks "What mattered?", Knowledge Synthesis asks "What remains true?"

Memory is narrative. It captures the shape of experience — what happened, how it felt, why it seemed important at the time. But narrative is expensive. It resists generalisation. You cannot efficiently query a story for the rule it implies.

Knowledge Synthesis is the compression stage that extracts durable structure from episodic narrative. It reads a Memory and asks: What can be lifted out of this particular moment and carried forward as reusable understanding?

The result is Knowledge — atomic, typed, queryable. A fact about the user. A principle the Anima has inferred. A method that worked. A concept that organises other concepts. Knowledge is what the Anima *knows*, abstracted from the circumstances of how it learned.

**The Transformation**
Memory → Knowledge is a second compression in the Elephantasm pipeline:


Each stage moves toward greater abstraction and reusability. Events are timestamped and verbose. Memories are compressed but still situational. Knowledge is atomic and portable — it can inform decisions without requiring the Anima to recall the specific moment it was learned.

This layered architecture prevents a common failure mode: systems that either remember everything (drowning in specificity) or forget everything (losing continuity). Elephantasm remembers *and* abstracts, preserving provenance while building a queryable world model.

**The Workflow Architecture**
The current Knowledge Synthesis algorithm is implemented as a LangGraph StateGraph with three nodes:

```
START (input: memory_id)
  ↓
fetch_memory (sync, DB read with RLS)
  ↓
synthesize_knowledge (async, LLM call)
  ↓
persist_knowledge (sync, DB write + audit + embedding)
  ↓
END
```

Unlike Memory Synthesis, there is no threshold gate. Knowledge Synthesis always runs to completion when triggered. The question is not "has enough accumulated?" but "can anything be extracted from this Memory?"

**Node Responsibilities**
1. fetch_memory — Retrieves the source Memory with full metadata (summary, content, importance, confidence). Operates under RLS context to ensure tenant isolation.

2. synthesize_knowledge — The LLM call. Sends the Memory to the configured model with a structured extraction prompt. Parses JSON array response.

3. persist_knowledge — Creates Knowledge records, generates embeddings, writes audit logs. Handles deduplication (replaces previous extractions from the same Memory). Atomic transaction.

**Trigger Mechanisms**
Knowledge Synthesis runs via three trigger modes, each serving a different operational need.

Automatic Cascade (Post-Memory)
When Memory Synthesis completes successfully, it fires a cascade trigger. This is fire-and-forget — Memory Synthesis doesn't wait for Knowledge Synthesis to complete. The cascade ensures that every new Memory has the opportunity to contribute Knowledge without manual intervention.

The pattern creates a continuous pipeline: Events → Memories → Knowledge, each layer feeding the next automatically.

Manual Trigger (API)
Operators can force extraction for any Memory. Manual triggers are useful for backfilling Knowledge on Memories created before the automatic cascade existed, or for re-extracting after prompt improvements.

Dreamer Retrospective (Planned)
During Deep Sleep, the Dreamer may identify Memories that warrant Knowledge re-extraction, for example, after merging two Memories or when confidence scores have changed. This trigger mode is planned but not yet implemented.

We may choose to implement this in a separate, dedicated service called “Meditator”, in order to more clearly differentiate it from the Dreamer service, which is primarily focused on Memory clean-up rather than Knowledge.

**Source Type Classification**
Every Knowledge item carries a source_type:

1. External Knowledge: Derived from user-facing Events/Memories e.g. "*User mentioned they live in London*"
2. Internal Knowledge: Derived from system's own reflection e.g. "*Pattern detected across 12 Memories: user prefers concise responses*"

EXTERNAL knowledge is the default. It comes from the Memory Synthesis pipeline, which processes user-facing Events.

INTERNAL knowledge is produced by the Dreamer's Deep Sleep phase, where the system reflects on its own Memory corpus and extracts patterns that were never explicitly stated. This mirrors the human experience of "sleeping on a problem" and waking with insight.

The distinction matters for trust calibration. EXTERNAL knowledge has clear provenance, whereas INTERNAL knowledge is inferred, which means it should be treated as hypothesis until confirmed.

**Empty Extractions**
Not every Memory yields Knowledge. A Memory like "*User said hello, I responded with a greeting*" contains no reusable truth. The correct response is an empty array.

Empty extractions are valid outcomes, not errors. The workflow records skip_reason="no_extractions" and completes normally.

**Overall Design Philosophy: Knowledge Synthesis**
Knowledge Synthesis embodies several Elephantasm principles:

Structure from narrative
Memories tell stories; Knowledge states truths. The transformation is lossy by design — we discard situational detail to gain generalisability.

Typed epistemics
Not all knowledge is the same. Facts, concepts, methods, principles, and experiences require different handling. The type system makes this explicit.

Automatic cascade
Knowledge extraction shouldn't require manual intervention. Every Memory automatically contributes to the Knowledge layer, maintaining alignment across the stack.

Conservative extraction
The prompt biases toward selectivity. Better to extract three high-quality items than fifteen dubious ones. Empty extractions are valid.

Traceable provenance
Every Knowledge item links to its source Memory. Every mutation logs an audit entry. The system's beliefs are always accountable.

# // VECTOR SEARCH & SEMANTIC RETRIEVAL

Vector search is vital for fast, associative recall, but it's only one layer. Elephantasm builds structured metadata and relational links on top of embeddings, allowing hybrid retrieval that is both efficient and semantically meaningful.

This section details how Elephantasm implements semantic search using pgVector, how it integrates with the broader retrieval system, and how it serves both the Pack Compiler (for context injection) and the Dreamer (for merge detection).

Elephantasm uses OpenAI's text-embedding-3-small model for all vector operations. This model offers a strong balance of quality and cost. The 1536-dimensional output captures semantic nuance without the computational overhead of larger embedding models. All embeddings across the system — memories, knowledge, queries — use the same model to ensure consistent similarity calculations.

Embeddings are generated asynchronously as a best-effort enhancement:

```
# Simplified embedding generation
embedding = embedding_provider.embed_text(memory.summary)
memory.embedding = embedding
memory.embedding_model = "text-embedding-3-small"
```

Failures in embedding generation log warnings but do not block the primary operation. A memory without an embedding remains queryable through other means; it simply cannot participate in semantic search until regenerated.

**Database Schema**
Vector columns are added to tables that require semantic search capabilities.

```
class Memory(SQLModel, table=True):
    # ... core fields ...

    embedding: list[float] | None = Field(
        default=None,
        sa_column=Column(Vector(1536), nullable=True)
    )
    embedding_model: str | None = Field(
        default=None,
        max_length=50,
        description="Model used to generate embedding"
    )
```

```
class Knowledge(SQLModel, table=True):
    # ... core fields ...

    embedding: list[float] | None = Field(
        default=None,
        sa_column=Column(Vector(1536), nullable=True)
    )
    embedding_model: str | None = Field(
        default=None,
        max_length=50
    )
```

The embedding_model field tracks provenance — if the embedding model changes in future versions, records can be identified and re-embedded accordingly.

**Indexing Strategy**
Elephantasm uses PostgreSQL's pgVector extension with IVFFlat indexing for approximate nearest neighbor search. IVFFlat (Inverted File with Flat quantization) offers:

- Faster index creation than HNSW
- Lower memory footprint during queries
- Sufficient accuracy for memory retrieval use cases

HNSW provides marginally better recall at the cost of higher memory usage. For Elephantasm's current scale and retrieval patterns, IVFFlat delivers the right tradeoff.

**Cosine Similarity Search**
Semantic search uses cosine distance as the similarity metric.

Distance/Meaning:
0.0 → Identical Vectors
1.0 → Orthogonal (unrelated)
2.0 → Opposite vectors

SQL Query Pattern:

```sql
SELECT id, summary,
       1 - (embedding <=> query_embedding) AS similarity
FROM memories
WHERE anima_id = $1
  AND state = 'active'
  AND embedding IS NOT NULL
  AND (embedding <=> query_embedding) < 0.3  -- distance threshold
ORDER BY embedding <=> query_embedding
LIMIT 10;
```

The <=> operator computes cosine distance. Results are filtered by a threshold, then ordered by ascending distance (most similar first). The similarity score returned is 1 - distance, converting to an intuitive 0–1 scale where 1 means identical.

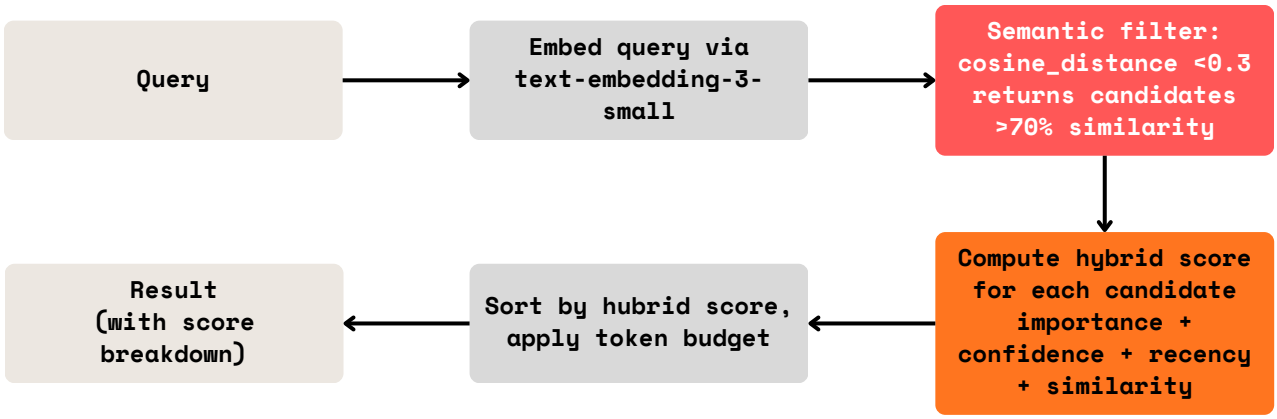
**Hybrid Retrieval: Vector + Multi-Factor Scoring**
Pure vector search returns items by semantic similarity alone. Elephantasm extends this with hybrid retrieval, combining embedding similarity with temporal, importance, and confidence factors.

The Five-Factor Formula
For long-term memory retrieval, each candidate is scored using:

```
score = (importance × w_i) + (confidence × w_c) +
        (recency × w_r) + ((1 - decay) × w_d) +
        (similarity × w_s)
```

These weights are configurable per retrieval request, allowing different cognitive styles:

- Conversational preset: Heavier recency weight (0.35)
- Research preset: Heavier importance and similarity (0.30 each)
- Historical preset: Lower decay penalty, favoring older memories



Pack Compiler Integration

The Pack Compiler uses semantic search for two of its four layers: Knowledge and Long-term Memory retrieval and compilation.

Before retrieval, the user's input is embedded. This embedding is used for both knowledge and long-term memory search. Session memories skip semantic search; they are filtered purely by time window, ensuring immediate conversational context is preserved regardless of semantic relevance.

Items below the threshold are excluded entirely. This prevents semantically distant content from consuming token budget, even if it scores well on other factors.

**Dreamer Merge Detection**

The Dreamer uses vector similarity to identify potentially redundant memories during Light Sleep.

```python
def _find_similar_by_embedding(
    session: Session,
    source_memory: Memory,
    candidates: list[Memory],
    threshold: float = 0.3,  # cosine distance
) -> list[UUID]:
    """Find memories similar to source via pgVector."""

    result = session.execute(
        select(Memory.id)
        .where(
            and_(
                Memory.id.in_(candidate_ids),
                Memory.embedding.isnot(None),
                Memory.embedding.cosine_distance(source_memory.embedding) < threshold,
            )
        )
    )

    return [row[0] for row in result.all()]
```

The Dreamer uses a lower threshold (distance < 0.3 ≈ similarity > 70%) than Pack retrieval because merge candidates are subsequently reviewed by an LLM. False positives are filtered in Deep Sleep; the goal of Light Sleep is recall over precision.

Fallback: Jaccard Similarity

For memories without embeddings, the Dreamer falls back to word-based Jaccard similarity:

```python
def jaccard_similarity(text_a: str, text_b: str) -> float:
    """Jaccard index: intersection / union of word sets."""
    words_a = set(text_a.lower().split())
    words_b = set(text_b.lower().split())

    intersection = len(words_a & words_b)
    union = len(words_a | words_b)

    return intersection / union if union > 0 else 0.0
```

Post-Curation Regeneration

When the Dreamer merges, splits, or updates a memory's summary, the embedding is regenerated. This ensures the merged/split memory immediately participates in future semantic searches with an accurate embedding.

```python
def regenerate_embedding(session: Session, memory: Memory) -> bool:
    """Regenerate embedding after content change."""
    try:
        new_embedding = embedding_provider.embed_text(memory.summary)
        memory.embedding = new_embedding
        memory.embedding_model = "text-embedding-3-small"
        session.add(memory)
        return True
    except Exception as e:
        logger.warning(f"Failed to regenerate embedding: {e}")
        return False
```

**Performance Characteristics**

Estimated Query Latencies

- Singel embedding generation ~100-200ms
- Semantic search (indexed, <10K memories) ~ 5-20ms
- Batch embedding (10 texts) ~300-500ms

Scaling Considerations

- IVFFlat performs well up to ~1M vectors per table
- Beyond that, consider HNSW or partitioning strategies
- Index must be rebuilt if lists parameter needs adjustment

Best-Effort Pattern

Embedding operations are non-blocking. This ensures that transient API issues don't block memory synthesis or Dreamer operations. Unembedded memories remain functional, these are retrieved by other means and flagged for later embedding regeneration.

**Summary**

Vector search in Elephantasm serves two distinct purposes:

1. Retrieval: The Pack Compiler uses semantic similarity as one factor in a multi-dimensional scoring system, balancing relevance with importance, confidence, recency, and decay.

2. Curation: The Dreamer uses embedding distance to detect potentially redundant memories, flagging them for LLM review and possible merge.

In both cases, vector similarity is a means rather than an end. Pure embedding search returns what's semantically close; Elephantasm's hybrid approach returns what's meaningful i.e. content that is both relevant and valuable according to the agent's evolving priorities.

Key design principles:

- Hybrid over pure: Vector similarity is one signal among many
- Best-effort embedding: Non-blocking generation with graceful fallbacks
- Configurable thresholds: Different use cases warrant different sensitivity
- Provenance tracking: Embedding model recorded for future re-embedding

The result is a retrieval system that leverages the associative power of embeddings while maintaining the structured, auditable scoring that deterministic memory requires.

# // APIs & SDKs

Elephantasm was designed from the outset to be composable and to integrate into any agentic framework, any LLM pipeline, any application that needs continuity.

This requires more than documentation. It requires architecture that treats programmatic access as a first-class concern: authentication that works for both humans and machines, rate limiting that scales with usage, and SDKs that make integration feel native to each language ecosystem.

The goal is simple: any developer should be able to give their agent long-term memory in under ten minutes.

**API Architecture**
Elephantasm exposes a RESTful HTTP API built on FastAPI. All endpoints follow predictable conventions:

| Method | Endpoint Pattern | Purpose / Behaviour |
|---|---|---|
| POST | /resource | Create a new resource (returns 201 Created) |
| GET | /resource | List resources with pagination (returns 200 OK) |
| GET | /resource/{id} | Retrieve a single resource (returns 200 OK or 404 Not Found) |
| PATCH | /resource/{id} | Partially update a resource (returns 200 OK) |
| DELETE | /resource/{id} | Soft-delete a resource (returns 200 OK) |
| POST | /resource/{id}/restore | Restore a previously soft-deleted resource |

Every endpoint returns JSON. Error responses include structured detail:


```
{
 "detail": "Anima not found",
 "error": "not_found",
 "resource": "anima",
 "id": "550e8400-e29b-41d4-a716-446655440000"
}
```

### Endpoint Taxonomy

The API organises endpoints by domain:

| Domain | Method | Endpoint | Purpose |
|---|---|---|---|
| Core Memory Operations | POST | /api/events | Ingest event (extraction into memory pipeline) |
| Core Memory Operations | GET | /api/events | List events (filtered by anima) |
| Core Memory Operations | GET | /api/memories | List memories (filtered by anima) |
| Core Memory Operations | GET | /api/knowledge | List knowledge objects (filtered by anima) |
| Core Memory Operations | POST | /api/packs/compile | Compile and inject a memory pack |
| Core Memory Operations | POST | /api/packs/compile/{preset} | Compile memory pack using a preset |
| Entity Management | POST | /api/animas | Create a new anima |
| Entity Management | GET | /api/animas | List animas |
| Entity Management | GET | /api/animas/{id} | Retrieve anima details |
| Entity Management | PATCH | /api/animas/{id} | Update anima |
| Entity Management | DELETE | /api/animas/{id} | Soft-delete anima |
| Curation & Synthesis | POST | /api/dreams/trigger | Trigger Dreamer curation cycle |
| Curation & Synthesis | GET | /api/dreams | List Dreamer sessions |
| Curation & Synthesis | POST | /api/synthesis/trigger | Trigger memory synthesis |
| Curation & Synthesis | GET | /api/synthesis/configs/{id} | Retrieve synthesis configuration |
| Administration | POST | /api/api-keys | Create API key |
| Administration | GET | /api/api-keys | List API keys |
| Administration | DELETE | /api/api-keys/{id} | Revoke API key |
| Administration | GET | /api/subscriptions/current | Retrieve current subscription status |
| Administration | GET | /api/usage | Retrieve usage statistics |

**Authentication**
Elephantasm supports two authentication methods, unified by a single principle: both paths must resolve to a user_id that can be used for row-level security.

JWT Authentication (Interactive)
For web applications and interactive use, Elephantasm validates JWTs issued by Supabase Auth. The flow:

1. Client authenticates with Supabase (OAuth, email/password, magic link)
2. Client includes JWT in Authorization: Bearer <token> header
3. Backend validates signature against Supabase JWKS endpoint
4. Backend extracts sub claim (Supabase user ID)
5. Backend looks up internal user.id via database trigger-created mapping
6. user.id is injected into database session for RLS enforcement

JWT validation is strict:

```
payload = jwt.decode(
    token,
    public_key,
    algorithms=["ES256"],
    audience="authenticated",
    issuer=f"{settings.SUPABASE_URL}/auth/v1",
    options={
        "verify_exp": True,
        "verify_nbf": True,
        "verify_iat": True,
        "verify_aud": True,
        "require_exp": True,
        "require_iat": True,
        "require_sub": True
    }
)
```

JWKS keys are cached with a one-hour TTL and automatically rotated when validation fails with an unknown key ID.

API Key Authentication (Programmatic)
For SDKs, background jobs, and programmatic access, Elephantasm issues API keys. The format is deliberately recognisable:

sk_live_a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6

The prefix sk_live_ identifies the key type; the suffix is 32 hexadecimal characters (16 random bytes).

Security properties:

- One-time visibility: The full key is returned only at creation, never retrievable afterward
- Bcrypt storage: Keys are bcrypt-hashed before storage; the database never contains plaintext keys
- Prefix lookup: The first 8 characters are stored in a separate column for efficient lookup without hash comparison on every key in the system
- Usage tracking: Each API call updates last_used_at and increments request_count
- Expiration: Optional expiration dates; expired keys are rejected
- Revocation: Keys can be deactivated without deletion, preserving audit trail

**SDKs**

Elephantasm provides official SDKs for Python (elephantasm on PyPI) and TypeScript (@elephantasm/client on npm), offering idiomatic access to the memory API in the two languages most prevalent in LLM application development.

Both SDKs maintain strict API parity — method names, parameter structures, and return types mirror each other, allowing patterns developed in one language to transfer directly to the other. Installation is standard for each ecosystem: pip install elephantasm or npm install @elephantasm/client.

The SDK interface is organised around two core operations that mirror the fundamental memory loop: extract() captures events as they occur, while inject() retrieves compiled memory packs for context injection. This pairing reflects how memory actually flows through an agentic system — experiences are extracted from interactions, processed into structured memory, and injected back into cognition when needed.

The canonical integration pattern becomes a simple cycle: extract the inbound event, inject context into the system prompt, generate a response, extract the outbound event. Each turn reinforces the memory substrate while maintaining full provenance.

Configuration follows the twelve-factor model in both SDKs, resolving credentials from constructor arguments or falling back to environment variables (ELEPHANTASM_API_KEY, ELEPHANTASM_ANIMA_ID, ELEPHANTASM_ENDPOINT). An optional default anima_id can be set at client instantiation, reducing parameter repetition in single-agent applications while remaining overridable per-call.

The TypeScript SDK ships as dual CJS/ESM with bundled type declarations, ensuring compatibility with Node.js, edge runtimes, and modern bundlers. The Python SDK uses Pydantic models throughout, providing runtime validation and IDE autocompletion.

Both SDKs translate HTTP status codes into typed exception hierarchies (AuthenticationError, RateLimitError, NotFoundError, ValidationError) enabling differentiated retry strategies and proper error discrimination. Exceptions carry structured detail from the API response, including error codes, affected resources, and validation failures, propagating diagnostic context without requiring additional response parsing.The MemoryPack object returned by inject() provides both a formatted prompt string via as_prompt() and typed accessors for individual layers when granular inspection is needed.

# IV. ROADMAP

# // THE ROAD FORWARD

Elephantasm's core architecture is operational. The fundamental claim that agents can maintain coherent long-term memory through structured curation has been validated in implementation, if not yet in rigorous benchmarks.

What follows is an outline of some of the technical decisions we are making, the integration patterns we are exploring, and the validation methodology we are developing.

**#1 Framework Independence**

The current implementation uses LangGraph for workflow orchestration — memory synthesis, knowledge extraction, and Dreamer curation all run as LangGraph state machines. This was a pragmatic choice. LangGraph provides checkpointing, retry logic, and observability out of the box. It let us move fast.

As Elephantasm matures, we are migrating away from LangGraph toward native workflow implementations. The reasoning is twofold.

Independence from Third-Party Frameworks

LangChain's ecosystem evolves rapidly. A memory system meant to provide long-term continuity cannot itself be subject to the churn of framework fashion. When a core workflow breaks because a dependency updated its interface, the system has failed at its most basic promise: reliability over time.

Native implementations give us full control over the execution model. The code becomes legible to anyone who reads Python — not just those familiar with LangChain's particular abstractions.

Portability Toward Performance

The second reason is more speculative but strategically important: a native Python implementation is a stepping stone toward a potential Rust rewrite.

Memory operations (scoring, sorting, filtering, embedding comparisons) are computationally intensive at scale. Rust offers memory safety, zero-cost abstractions, and performance that approaches C. If Elephantasm's backend ever needs to handle millions of memories with sub-10ms pack compilation, the path is clearer from native Python than from a framework-wrapped implementation. The workflows we build should be portable in principle, even if we never exercise that option.

The transition to native workflows will be incremental. Each workflow will be reimplemented as a plain Python module with explicit state management. LangGraph's checkpointing will be replaced with natively database-backed state tables. Retry logic will use standard patterns (exponential backoff, idempotency keys). Observability will shift to OpenTelemetry spans. The goal is to own our implementation of what are otherwise sound LangChain principles.

**#2 Benchmark Development**

How do you measure whether an agent remembers well?

Accuracy is easy to define for factual recall: did the agent retrieve the correct fact? But memory is not just retrieval. It is continuity of self. It is knowing not just what happened, but who was there, why it mattered, and how it connects to now. These are harder to quantify.

We are developing a benchmark suite that attempts to measure what matters:

LoCoMo (Long-Context Memory)
Adapted from existing benchmarks, LoCoMo tests whether an agent can recall specific facts from extended interaction histories. The metric is straightforward: given a question about something mentioned N turns ago, does the agent answer correctly? This tests retrieval accuracy but says nothing about integration or coherence.

Needle in Haystack (NIAH)
A stress test for precision. A single fact is embedded in hours of unrelated conversation. Can the agent find it when asked? This measures the pack compiler's ability to surface rare but relevant memories against a backdrop of noise.

Continuity Score
A custom metric for behavioral consistency. We present an agent with preference-laden scenarios across multiple sessions — "Do you prefer X or Y?" — and measure whether responses remain coherent over time. An agent with good continuity should not contradict its earlier stated preferences without explicit reason. This is the hardest to automate because it requires judgment about what counts as contradiction versus legitimate evolution.

Token Efficiency
How much context compression does Elephantasm achieve compared to raw transcript injection? If a 100,000-token conversation history can be represented by a 2,000-token pack without loss of relevant information, that is a 50x efficiency gain. We measure this by comparing task performance (e.g., question answering) between full-context and pack-injected conditions.

Latency
Pack compilation must be fast enough to not bottleneck inference. We target p50 < 100ms and p95 < 500ms for typical pack sizes. This becomes more challenging as memory stores grow; the scoring algorithms must scale sublinearly.

Benchmarks are meaningless without baselines. We are preparing comparisons against:
- Raw Context: Injecting the full conversation history (up to context window limits)
- Naive RAG: Standard vector retrieval without curation or scoring
- Mem0: A representative memory framework with different architectural assumptions
- Letta: The self-editing memory approach

The goal is not to "win" on every metric (as different systems optimize for different things) but to demonstrate that Elephantasm's structured approach offers measurable advantages for long-term coherence.

## #3 Integration Patterns
We are actively exploring three integration patterns that represent distinct use cases.

**Pattern 1: One-to-One (Personal Agents)**
Configuration: One Anima per human user. The agent remembers everything about its person.

Example: A personal assistant for physicians. The agent accumulates knowledge about the doctor's patients (within appropriate privacy boundaries), treatment preferences, scheduling patterns, and communication style. Over months, it becomes an extension of the physician's cognitive workspace — remembering which patient had which reaction to which medication, surfacing relevant history before appointments, adapting its tone to match the doctor's preferred brevity.

This pattern maximizes depth. The Anima develops a rich, specific model of one person's world. Identity injection becomes highly personalized. The Dreamer curates toward a single user's priorities.

Challenges:
- Privacy and data isolation are paramount (RLS enforcement is non-negotiable)
- Memory stores grow without bound — curation must be aggressive
- Cold-start problem: the agent is useless until it has accumulated enough context

**Pattern 2: One-to-Many (Shared Persona Agents)**
Configuration: One Anima shared across many users. The agent maintains a consistent persona while interacting with diverse people.

Example: A customer support agent for a SaaS product. The Anima embodies the company's voice, product knowledge, and support policies. It interacts with thousands of customers, each conversation adding to its understanding of common issues, edge cases, and effective resolutions. The same persona greets a first-time user and a long-time power user, drawing on accumulated experience to tailor its responses.

This pattern maximizes breadth. The Anima becomes a living repository of customer interactions that any support session can draw from. Knowledge synthesis is particularly valuable here, extracting durable insights from ephemeral conversations.

Challenges:
- Memory stores grow rapidly — aggressive curation is essential
- Individual user context must not bleed across sessions (privacy)
- The Anima must generalize without losing specificity

**Pattern 3: Backend Workflows (Non-User-Facing Agents)**
Configuration: Animas attached to automated pipelines, not human conversations.

Example: A data processing agent that runs nightly ETL jobs. Each run generates events (files processed, errors encountered, decisions made). Memories synthesize patterns: "This data source frequently has malformed timestamps on Mondays." Knowledge crystallizes: "The optimal batch size for this pipeline is 10,000 records." The Dreamer notices when something changes — a new error pattern, a shift in data distribution.

This pattern treats Elephantasm as an audit and learning layer for automated systems. The agent is not chatting with anyone; it is observing, remembering, and improving.

Use Cases:
- Performance Optimization: The agent learns which parameters work best for different conditions, injecting that knowledge into future runs.
- Anomaly Detection: Memory of past behavior surfaces when current behavior deviates — "This has never happened before" becomes a computable statement.
- Audit Trail: Every decision the pipeline makes is recorded, synthesized, and searchable. Compliance and debugging become easier.
- Institutional Knowledge: When a human engineer needs to understand why the system behaves a certain way, the Anima can explain based on accumulated experience

Challenges:
- Event volume can be enormous (thousands per minute) — ingestion must be selective
- Synthesis triggers need different heuristics (not time-based, but batch-based)
- Identity layer is less relevant; knowledge and memory layers dominate

"Memory and forgetfulness are as life and death to one another.
To live is to remember and to remember is to live.
To die is to forget and to forget is to die"

Samuel Butler