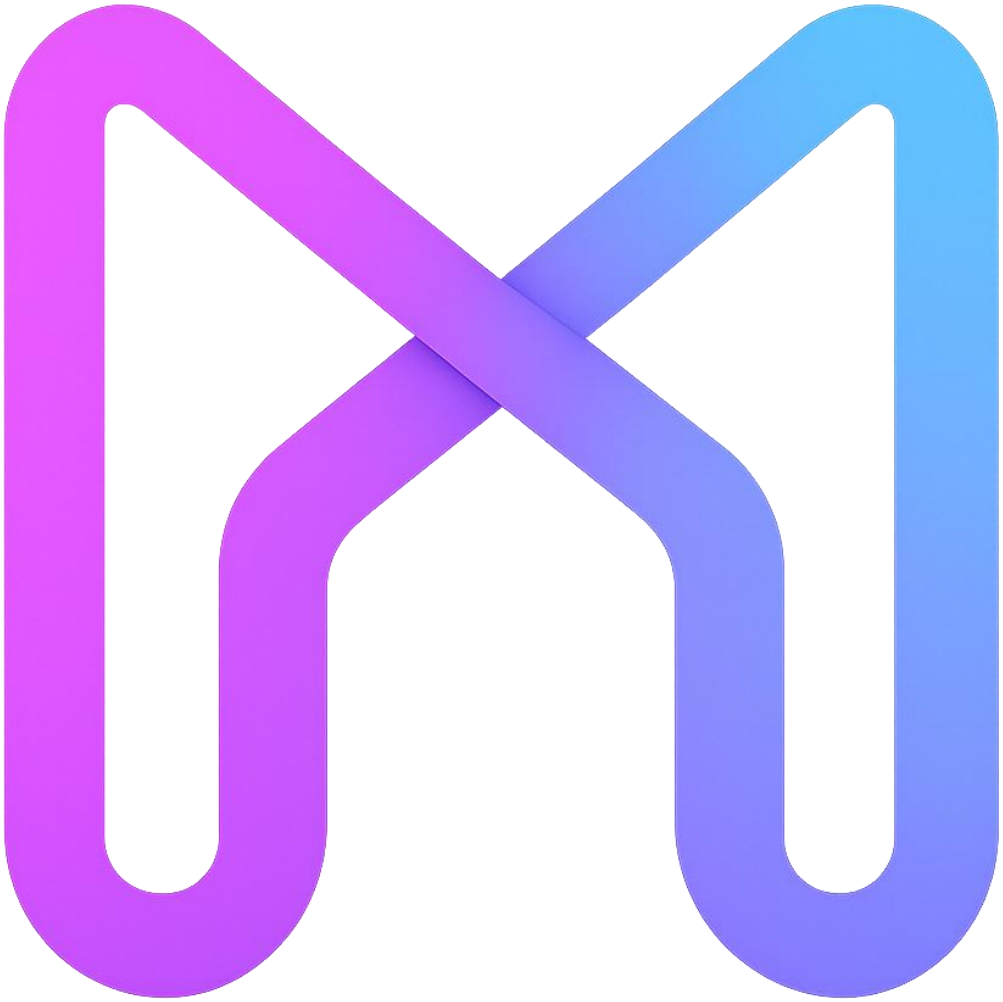

MetaOp AI

Technical Whitepaper

Signal Intelligence for Operators

Architecture · Knowledge Representation Layer · Pattern Engine

Cognition Orchestration · Observability · Deployment Model

Version 7.0 · May 2026 · metaop.ai · founder@metaop.ai

A Note Before You Read This

This started as a side project. It still feels like one — in the best way.

My name is Tony S. I am a security engineer with over 15 years of experience in cloud and infrastructure security, and 3 years of AI security governance at a global financial institution. Like many people, I started using ChatGPT as a personal journal, exploring my own patterns and working to improve areas of my life that needed attention. It was genuinely helpful — until I kept colliding with the same frustrating wall: the session would grow too large, the context window would collapse, and I would lose everything. All the continuity, all the hard-earned insights, gone.

At scale, I noticed several compounding failure modes: token bloat, inefficient full-session replay, increasing signal-to-noise degradation as sessions grew, and what I call regenerative context inflation — a phenomenon where the model continuously summarizes, rewrites, and appends its own generated interpretations back into future context. Over time, the user's actual input becomes diluted, compressed, or lost entirely. The result is familiar to most users: once the context window collapses under its own accumulated weight, continuity breaks, sessions reset, and you are forced to redeploy important information repeatedly — sometimes daily.

I tried other models. Same problem. Around December 2025, I grew increasingly frustrated with constantly copying and pasting historical context just to continue a thought. I searched the App Store, expecting someone to have solved this already. Nothing came close to what I actually needed.

So I did what any sufficiently annoyed engineer would do: I decided to build it myself.

MetaOp AI started as a side project in late February 2026. I drew a rough wiring diagram on paper and began experimenting. At first, I thought I would just build a better wrapper around an existing model — but I quickly realized I had simply recreated the exact same problem I was trying to escape. That was a humbling moment.

What followed was months of obsessive work: weekends, days off, late nights, and time on my commute. I had no deep background in AI engineering when I started — just frustration and a clear problem I wanted solved. I spent a surprising portion of my waking hours building something that did not yet exist: real, persistent continuity for personal reflection, grounded in structured cognition rather than conversation replay.

This whitepaper is the result of that journey. It is written by a guy who wanted his AI journal not to forget everything after a few weeks, and ended up building the cognition substrate documented within.

The best way to describe MetaOp AI: I'm that engineer in the garage trying to turn a beat-up Honda Civic into something that can gap a Koenigsegg. I was learning the AI stack as I built, and that outsider position became an advantage — I was not constrained by the assumptions most AI products inherit.

I am not claiming to have built something impossible. I am simply saying I explored a direction that most in the AI industry do not prioritize, and what I discovered felt worth documenting in full.

If any of this resonates with you, the contact at the end of this document is real. I would rather have a small number of thoughtful conversations than a wide spray of marketing.

— Tony S., Founder, MetaOp AI

Table of Contents

Executive Summary

MetaOp AI is a signal-intelligence relationship-cognition platform. Users journal about their lives and relationships in natural language; the system extracts structured signals from that narration, accumulates them into a durable knowledge substrate, detects behavioral patterns over time, and conditions the AI's replies on that accumulated cognition. The product value is the pattern engine and the multi-layer context richness it builds: the longer and across more contexts a user journals, the more the system can reflect genuine, evidence-grounded patterns back to them.

Core Technical Thesis

MetaOp AI treats the large language model as a stateless commodity at the rendering edge. Durable intelligence lives in a governed substrate: the Knowledge Representation Layer (KRL). The system extracts structured records from narration, stores them with provenance and confidence metadata, retrieves only the scope-correct subset for each query, and composes prompts from dense structured meaning rather than accumulating raw transcript. This architecture eliminates the regenerative context inflation that destroys continuity in general-purpose AI assistants.

> **Architectural Principle**
> *The LLM is the rendering layer. The Knowledge Representation Layer is the intelligence. One is rented infrastructure; the other is the product moat.*

Key Differentiators

- Structured cognition substrate (KRL) replaces raw conversation replay, eliminating token bloat and signal degradation.
- Four-tier pattern escalation — from atomic signals to compound cross-scope arcs — surfaces dynamics invisible to flat conversation history.
- Three distinct cognition identities (USER, ENTITY, SPACE) on one shared infrastructure preserve product differentiation without forking the codebase.
- Multi-gate compounding signal stack between narration and prompt maintains high signal-to-noise at scale.
- Temporal continuity layer adds arc detection, anchor parsing, and relationship-state snapshots, making the substrate more valuable over time.
- Synthetic cognition layer solves cold-start without contaminating the durable substrate through a strict physical separation of concerns.
- Full observability via per-turn cognition traces enables post-hoc debugging of AI behavior as a governed runtime system.

System Composition at a Glance

Layer	Component	Primary Responsibility
Ingestion	Growth, Entity & Space Journals	Capture user narration across three cognition scopes
Extraction	Six Parallel Extractors	Convert narration into typed, provenance-stamped substrate

Layer	Component	Primary Responsibility
		records, including space-filter routing before downstream writes
Substrate	Knowledge Representation Layer (KRL)	Durable 4×5 matrix: four subject scopes × five layer kinds
Intelligence	Pattern Engine (43 registered detectors)	Compound signals into behavioral patterns with confidence bounds
Governance	Cognition Orchestrator + Policy Resolver	Scope-correct retrieval, policy resolution, prompt assembly
Rendering	Composer + LLM Provider	Translate structured cognition into natural-language response
Observability	Trace Layer + Versioning	Per-turn audit trail of every cognition decision
Infrastructure	Azure PaaS + Redis + PostgreSQL	Durable, scalable, privately networked deployment target

PART I

System Architecture

MetaOp AI's architecture is organized into seven independently versionable layers. Each layer has a single, well-defined responsibility. The substrate is the product moat; the model is rented infrastructure at the rendering edge. The architecture was designed so that swapping the LLM provider, the prompt format, or the retrieval policy requires no changes to the durable knowledge substrate.

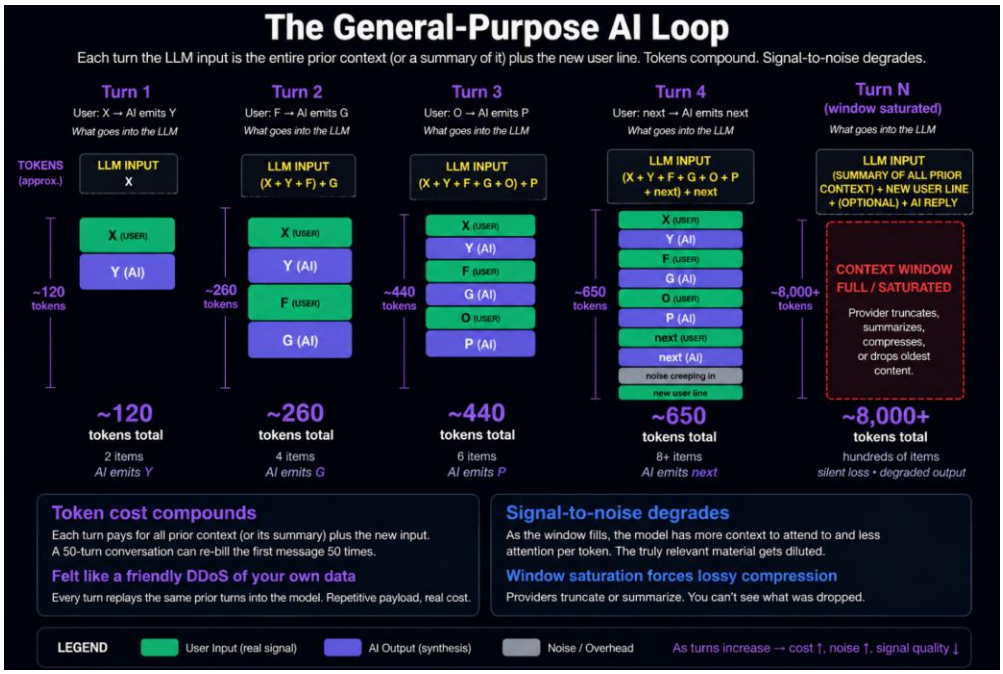

Figure 1 — The MetaOp AI data and privacy flow: user narration enters the application, passes through the cognition substrate, and reaches the model layer. Only structured cognition crosses the boundary; raw conversation history does not.

1.1 Technology Stack and Topology

The frontend is a Vite + React + TypeScript single-page application whose visual design is locked and not subject to arbitrary modification. The backend is FastAPI on Python with async SQLAlchemy over PostgreSQL (asyncpg driver), Redis for caching, rate-limiting, and distributed locking, and OpenAI as the primary LLM provider abstracted behind a logical model router. Database schema is managed by Alembic migrations (51 revisions, head 0051). Everything runs under Docker Compose locally and is bound for Microsoft Azure in production.

Component	Technology	Role
Frontend	Vite + React + TypeScript SPA	Zustand stores, React Query, SSE consumption; aesthetic-locked
Backend	FastAPI (async), SQLAlchemy + asyncpg, Uvicorn	API layer, orchestration, streaming, quota enforcement

Component	Technology	Role
Database	PostgreSQL with JSONB	Durable substrate; 51 Alembic migrations; JSONB for pattern_data, policies, blobs
Cache / Lock	Redis	Rate-limit sliding windows, hero/content caches, distributed cognition lock
LLM	OpenAI / Azure OpenAI	Provider abstraction; logical model names resolved at call time; Azure in prod
External RAG	Tavily	Web enrichment; fail-open, kill-switchable; deliberately scoped to Growth Journal inline and Intel Reports
Infrastructure	Microsoft Azure PaaS	Azure Database for PostgreSQL, Azure Cache for Redis, Container Apps, Key Vault, Managed Identity

1.2 Build Model and Deployment

The backend Dockerfile bakes application code into the image (COPY .), so code changes require an image rebuild (docker compose up -d --build api). Environment changes require a recreate (--force-recreate). The .env file is mounted into the api container so secrets are not permanently baked into the image layer. The production deployment targets Azure Container Apps with Azure Database for PostgreSQL Flexible Server and Azure Cache for Redis, both on private networking. Secrets are managed through Azure Key Vault with Managed Identity, eliminating credential distribution as a security surface.

1.3 The Journal-Turn Request Lifecycle

The signature operation in MetaOp AI is a journal turn: the user posts a message to one of the four journaling surfaces and receives a streamed AI reply delivered over Server-Sent Events (SSE). Understanding this path is the key to understanding the whole system, because nearly every backend component participates in it. The turn lifecycle proceeds through seven discrete stages:

Stage	Operation	Key Component
1. Admission	Streaming route handler receives POST; per-user concurrency semaphore enforces cap of 3 simultaneous streams	journal.py / concurrency.py
2. Quota Wall	Pre-flight quota gate checks token availability; over-cap turns return HTTP 402 with structured over_quota payload	billing/enforcement.py
3. Continuity Assembly	Orchestrator resolves policy, loads KRL substrate slice, assembles continuity block and optional synthetic scaffold	orchestrator.py / continuity_block.py
4. Generation	Scope-correct prompt assembled with safety overlays; response streamed to client via SSE	conversation.py / openai_provider.py

Stage	Operation	Key Component
5. Early Commit	Reply, message rows, token-usage record, and cognition trace persisted and committed before the cognition tail runs	journal.py repository layer
6. Cognition Tail	Six extractors run, persist to KRL via upsert chokepoint, pattern engine re-evaluates, settle version bumped	_journal_pipeline_helpers.py
7. Terminal Frame	SSE terminal frame carries freshly-detected signals/patterns; or, with background_cognition=true, returns cognition_pending=true immediately	SSE framer / background task

1.4 Persistence Layer and Data Model

SQLAlchemy models live in app/models and are accessed exclusively through repository modules in app/repositories — one repository per aggregate. All database access flows through repositories rather than ad-hoc queries in routes. The schema is migration-managed; 51 Alembic revisions track its evolution, including the decommission of the legacy signals table (signals now surface through signal_views over extracted_records).

Core Entity Tables

Model	File	Role
User	user.py	Account: identity, auth fields, subscription_tier, self_context baseline, per-user cognition_policy (JSONB), behavioral-capture toggles
Space	space.py	A context/environment the user journals in (Work, Family, etc.). Carries per-space cognition_policy JSONB; unit of graduated resource cap
Entity	entity.py	A named person explicitly created by the user (role nullable). Paywalled; never auto-created from narration or @-mention
Space-Entity	space_entity.py	Junction assigning entities to spaces; @-mention resolution selects only space-assigned entities
Conversation	conversation.py	A journaling session/thread; carries scope and synthetic_cognition_blob JSONB
Growth Entry	growth_entry.py	Growth-journal entries feeding self_context baseline

Knowledge and Cognition Tables

Model	File	Role
Extracted Record	extracted_record.py	Canonical durable substrate row: signal/event/context/meta-context with subject_kind, subject_id, confidence, provenance
Pattern	pattern.py	user_pattern_bank row: detected pattern with pattern_data JSONB holding subjects, tier,

Model	File	Role
		confidence, evidence_signal_count, lifecycle_state
Pattern History	pattern_history.py	Lifecycle transition log (active↔dormant, reactivations) for each pattern
Cognition State	cognition_state.py	Per-(user, space) consistency counters: pattern_settle_version, signal_settle_version, rounds
Cognition Trace	cognition_trace.py	Per-turn observability record; pruned after 7 days
Event / Event Link	event.py / event_link.py	Narrated events and directed links between them; used by arc detectors and event-sequence continuity
Relationship State Snapshot	relationship_state_snapshot.py	Point-in-time dyad state vector (trust, reciprocity, repair, conflict, trajectory); written post-scoring
Billing	billing.py	Subscription, usage records, burst pools, target_tier / keep_space_ids for downgrade workflow
Consent Log	consent_log.py	Append-only consent audit (legal/compliance)

PART II

The Cognition Identity Charter

Before discussing any implementation detail, the foundational architectural principle that organizes every decision in this system must be stated explicitly:

> **The Charter**
> *The cognition infrastructure is shared. The cognition identity is scope-specific. Every refactor, every shared-helper extraction, every prompt edit must preserve this duality.*

Three cognition products run on one substrate, but they are not the same product. A user reflecting in their personal Growth Journal expects warmth and grounded reflection; a user analyzing their relationship with a specific person in an Entity Journal expects dyadic clarity and observational restraint; a user interrogating the dynamics of a multi-person environment in a Space Journal expects systemic analytical structure. Collapsing the three into one experience would destroy the product differentiation users are paying for while leaving the underlying engineering intact — the single most expensive architectural mistake this system could make.

The charter is enforced by both code and policy. The orchestrator runtime is shared (one base class, three subclasses differing only in their default policy and scope name). The cognition policy resolver, the continuity assembly, the pattern injection mechanics, the trace persistence, the evidence hierarchy, the sparse-substrate directives, and the READ/WRITE separation principles are all shared infrastructure. The prompt builders, the tonal posture, the reasoning style, the question shape, and the terminal payload shapes are scope-specific. This duality is enforced at the prompt boundary by the SCOPE_IDENTITY_USER, SCOPE_IDENTITY_ENTITY, and SCOPE_IDENTITY_SPACE constants in app/services/prompts/conversation.py, appended to the system prompt by build_messages() based on the calling route's SCOPE_NAME.

2.1 The Three Scope Identities

Each cognition scope has a primary question, a focus, a tone, a reasoning style, and a signature feeling phrase that the user should experience after a successful turn. These are not abstract design goals; they are operational targets that any change to the cognition layer must preserve.

Scope	Primary Question	Focus	Tone	Signature Feeling
USER	What is happening within the user?	Self-awareness, identity, drift, growth	Warm, grounded, reflective	"I feel understood."
ENTITY	What is happening between the user and this person?	Trust, reciprocity, repair, asymmetry	Nuanced, observant, relationally intelligent	"I understand this relationship more clearly."
SPACE	What is happening in this environment or system?	Group dynamics, systemic tension, organizational drift	Analytical, structured,	"I understand the dynamics

Scope	Primary Question	Focus	Tone	Signature Feeling
			strategic, systems-aware	of this environment."

2.2 Scope versus Surface

A common point of confusion is the difference between a cognition scope and a UI surface. The two are related but distinct, and the architecture depends on keeping them conceptually separate.

There are exactly three cognition scopes, each with its own identity, prompt skeleton, retrieval shape, and policy defaults: USER (introspective), ENTITY (dyadic), and SPACE (systemic). Every journal turn happens in exactly one scope. The orchestrator resolves the scope at the start of the turn and every downstream component inherits it.

The product has approximately ten places where the user interacts with cognition: the space journal, the user journal, the entity chat, the growth journal, the Mirror surface, the 360 view, the cross-space whiteboard, the onboarding context conversation, analytical substrate queries, and the burst surface. Some are journals (write paths into the substrate). Some are analytics (read-only views). Each surface declares which scope it occupies, and the orchestrator inherits the scope identity from that declaration. A scope is an identity; a surface is a place. The same scope can have multiple surfaces (USER scope has both the Growth Journal write path and the Profile read path). A single surface always lives in one scope.

Route	Scope	Orchestrator Subclass	Frontend Surface
POST /api/journal	SPACE	JournalOrchestrator (SPACE default)	Space Journal pane (per-space)
POST /api/entities/{id}/journal	ENTITY	EntityJournalOrchestrator	Entity Analysis pane (per-entity)
POST /api/users/me/journal	USER	UserJournalOrchestrator	Direct API surface
POST /api/profile/growth/stream	USER	UserJournalOrchestrator (Option D)	Growth Journal UI

PART III

The Knowledge Representation Layer (KRL)

The KRL is the canonical substrate that the entire cognition product is built on. It is a structured representation of everything the system knows about the user, their relationships, the environments they operate in, and the dynamics between all three. Every other component — the extractors that write to it, the pattern engine that compounds it, the orchestrator that retrieves from it, the analytics surfaces that visualize it — exists to serve or read from the KRL. Architecturally, there is no second source of truth. The substrate is one table backed by typed JSONB, drift-tracked, append-only or upsert-with-contradiction-gates depending on layer, partitioned by subject, and projected into prompts via the continuity block.

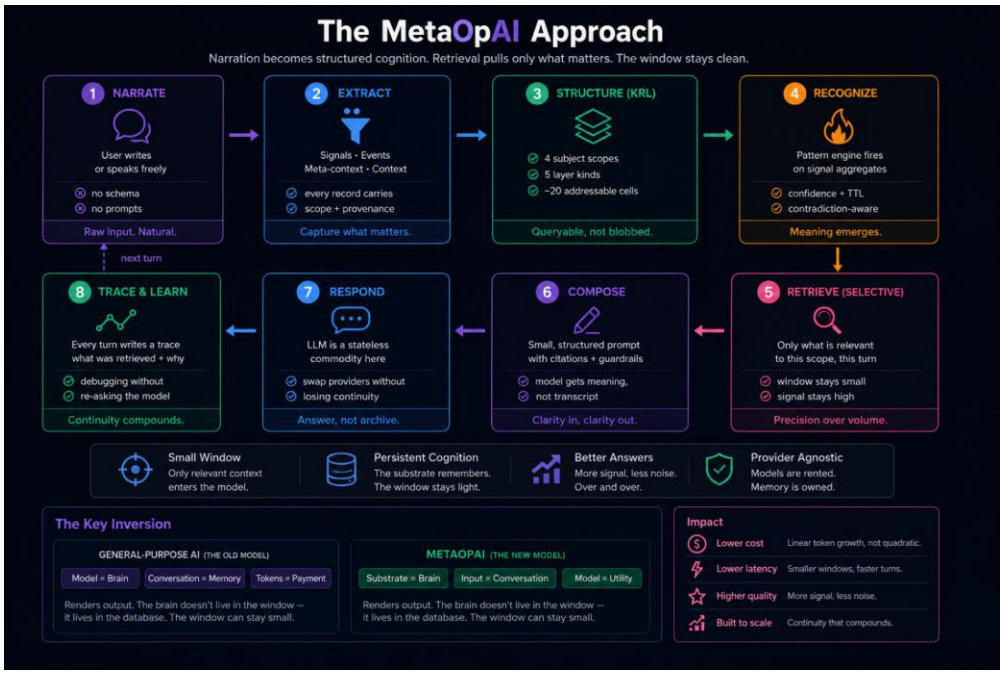

Figure 2 — The MetaOp AI cognition flow: narration enters the extraction pipeline, structured records accumulate in the KRL substrate, the pattern engine compounds them, and the orchestrator assembles scope-correct context for the composer.

3.1 The 4 × 5 Cognitive Matrix

The KRL is organized as a 4×5 matrix: four subject scopes across the rows, five extraction layers across the columns. Every record the system ever writes lands in exactly one of the twenty resulting cells. The matrix was locked on 2026-05-05 and supersedes every earlier representation. Future extractors must conform to it without exception.

Subject Scopes (Rows)

- **USER** — the operator of the system; the perspective from which everything is narrated.
- **ENTITY** — a specific named person in the user's life (a manager, a sibling, a partner, etc.).
- **SPACE** — an environment or system the user occupies (a workplace, a family, a friend group).

- **RELATIONSHIP_PAIR** — the dyad between the user and one entity, encoded with a deterministic UUID5 hash of the sorted (user_id, entity_id) pair so the same dyad always resolves to the same row regardless of write order.

Layer Kinds (Columns)

- **SIGNAL** — behavioral observations that decay (a single moment of dismissal, an act of withdrawal, a moment of self-reported exhaustion). Append-only. Carries polarity and a decay half-life.
- **EVENT** — atomic, permanent, relational occurrences (a delivery, a transition, a milestone, a falling-out, a repair). Append-only. Carries participants, witnesses, and references_event_ids that allow events to be linked across years.
- **META-CONTEXT** — interpretive claims about a subject ("Bob is malicious," "the space is high-gossip," "I'm an avoider"). Drift-tracked with contradiction gates. May change over time; the drift itself becomes a pattern.
- **CONTEXT** — stable facts about a subject (occupation, reporting relationship, demographics, identity). Drift-tracked. Changes are rare but consequential.
- **PATTERN** — computed from the other four layers, never directly written. Patterns live in the user_pattern_bank table but conceptually occupy the fifth column of the matrix.

The Matrix: Condensed Semantic Map

Subject \ Layer	SIGNAL	EVENT	META-CONTEXT	CONTEXT	PATTERN
USER	"I shut down"	"Panic attack Tuesday"	"I feel stuck lately"	"You work in finance"	Withdrawal Spiral
ENTITY	"Mike interrupted"	"Argument at lunch"	"Mike seems distant"	"Mike is your manager"	Reciprocity Drift
SPACE	"Team tension rising"	"Layoff meeting"	"Culture feels unstable"	"This is your Work space"	Organizational Stress
RELATIONSHIP_PAIR	"Avoidance between you two"	"Repair conversation"	"Trust feels weaker now"	"User-Mike dyad, 3 years"	Trust Rebuilding

3.2 The Eight Universal Metadata Fields

Every record in the matrix carries the same eight metadata fields, regardless of its cell. This uniformity is what allows the pattern engine to compound across cells and the continuity block to retrieve dimensionally. The fields are persisted on the extracted_records table in app/models/extracted_record.py.

Field	Type	Purpose
layer	enum	signal / event / meta_context / context / pattern — selects the column
subcontext	text	Sub-category within the layer (career_delivery, conflict_disagreement, occupation, etc.)

Field	Type	Purpose
dimension	enum	cognitive / emotional / behavioral / relational / structural / identity
subject_kind	enum	user / entity / space / relationship_pair — selects the row
subject_id	uuid	Resolves the subject; dyad uses deterministic hash of sorted pair
subject_aspect	JSONB	Multi-aspect weights {aspect: weight}, drift-tracked. Bob: {manager:0.9, rival:0.7, mentor:0.4}
perspective	enum	user_asserted / ai_inferred / observed / third_party_claim — MUST be set per record
polarity	enum	positive / negative / neutral / mixed
temporal_scope	enum	past_event / ongoing / recent_shift / cyclical / anticipated
confidence	float	0.0–1.0; below 0.5 floor will be rejected at the persistence chokepoint
provenance	string(32)	user_narration (durable) — locked in Phase L-1 to prevent recursive substrate ingestion

The subject_aspect field deserves special attention. It lets a single entity carry multiple weighted roles simultaneously. Bob being both your manager (weight 0.9) and your political rival (weight 0.7) does not require two ENTITY rows; it is one row whose subject_aspect JSONB tracks the weights independently. Aspects do not sum to one because they are not probabilities — they are independent intensities. The weights themselves drift over time, and that drift is detected and surfaced by the AspectDrift pattern detector: Bob-as-mentor fading while Bob-as-rival rises is itself a compounded relational signal the user can act on.

3.3 Three Narration Modes

The same user message can carry very different cognitive content depending on the user's relationship to what they are narrating. The extractor must classify the narration into one of three modes before deciding which subjects to write to and what perspective to mark each record with.

Mode	Example	Subjects Written	Perspective
Observation	User watches Bob and Sarah argue across the room.	ENTITY(Bob), ENTITY(Sarah)	observed
Narration	User talks about Bob in the Bob pane with no current interaction.	ENTITY(Bob)	user_asserted or observed
Dyad	"Bob and I had a falling out" — user is a participant.	RELATIONSHIP_PAIR(user, Bob) + ENTITY(Bob)	user_asserted

3.4 Drift Tracking and the Contradiction Gate

META-CONTEXT and CONTEXT are upsert-with-history, not append-only. When a new record arrives for an existing (user_id, subject_kind, subject_id, layer, subcontext) tuple, the upsert chokepoint in app/repositories/extracted_records.py applies three sequential checks:

Check	Threshold	Behavior
Confidence floor	confidence < 0.5	Write rejected outright
Contradiction delta	new confidence exceeds existing by < 0.2	Write recorded as a clarification proposal, surfaced to user on next turn
Contradiction delta (clear)	new confidence exceeds existing by ≥ 0.2	Silent overwrite with old value preserved in extracted_records_history
User-override flag	user_overridden = True	Refuse any AI write until user explicitly revisits the fact

This is a load-bearing mechanism for epistemic integrity. Without it, the system would silently overwrite known facts every time a new turn produced a marginal-confidence interpretation, gradually losing track of what the user has actually asserted. With it, the system maintains a one-turn lag on contradictions but never loses the older claim until the user explicitly chooses to retire it.

3.5 Why This Architecture Matters: Design Rationale

Deterministic Addressing over Similarity Search

Because every record carries a subject scope, subject identifier, and layer type, the orchestrator can ask the substrate precise questions: "show me recent Signal-layer records about the relationship between this user and Mike from the last 30 days above a 0.7 confidence threshold." This is a direct indexed lookup, not a broad semantic search over conversation embeddings. The system does not dump everything into a single vector index and hope the right memories surface; it retrieves exactly the continuity cell it needs.

Contradiction as Information

The substrate can hold multiple competing interpretations or conflicting records simultaneously rather than overwriting one with another. If a user's industry is described as "banking" in one narration and "telecommunications" in another, the system preserves both and surfaces the inconsistency for resolution. This lets the cognition layer ask clarifying questions rather than silently collapsing ambiguity into false certainty.

Soft Deletion over Destruction

Records are typically hidden or deprioritized rather than permanently erased. Older continuity may become relevant again. The system treats the past as something to be weighted down over time rather than discarded entirely, preserving the full audit trail from any pattern back to the original narration that produced it.

Query Performance Characteristics

Common query paths are backed by three composite indexes: (user, subject_kind, subject_id, layer) for list-by-subject lookups, (user, layer, created_at) for recent-by-layer scans, and (user, subject_kind, subject_id, subcontext) for single-field drift queries. At current scale the aggregate read path operates at sub-millisecond latency. During performance benchmarking of Python-side grouping versus SQL GROUP BY, the Python path outperformed SQL at the test scale of ~449 rows (≈0.096ms versus ≈1.15ms) because per-row CASE-and-regex work in SQL

dominated at that cardinality. The crossover point where SQL wins was hypothesized around 5,000 rows and documented for revisiting on Pro-tier telemetry — measuring before optimizing is a first-class engineering principle in this system.

Domain Generality

The KRL substrate is domain-general. The same 4×5 matrix structure can support personal relationships, team dynamics, patient care, research projects, or any other domain requiring persistent structured cognition — without changes to the core system. Only the pattern templates and parts of the extractor vocabulary are currently tuned for relationships and self-reflection. This clean separation is deliberate: it allows the substrate to remain general-purpose while permitting MetaOp AI to expand into new domains or become a platform in the future without rebuilding the foundation.

PART IV

The Extraction Pipeline

Extraction is the system's only mechanism for writing to the durable KRL. Nothing else writes — not the LLM, not the orchestrator, not the analytics surfaces, not the API client. The extractors are tightly scoped LLM-driven jobs that operate on raw human narration only, return structured proposals, and pass those proposals through a single canonical persistence chokepoint where all the discipline lives. The architecture treats writes to the KRL as an extraction problem, not a generation problem.

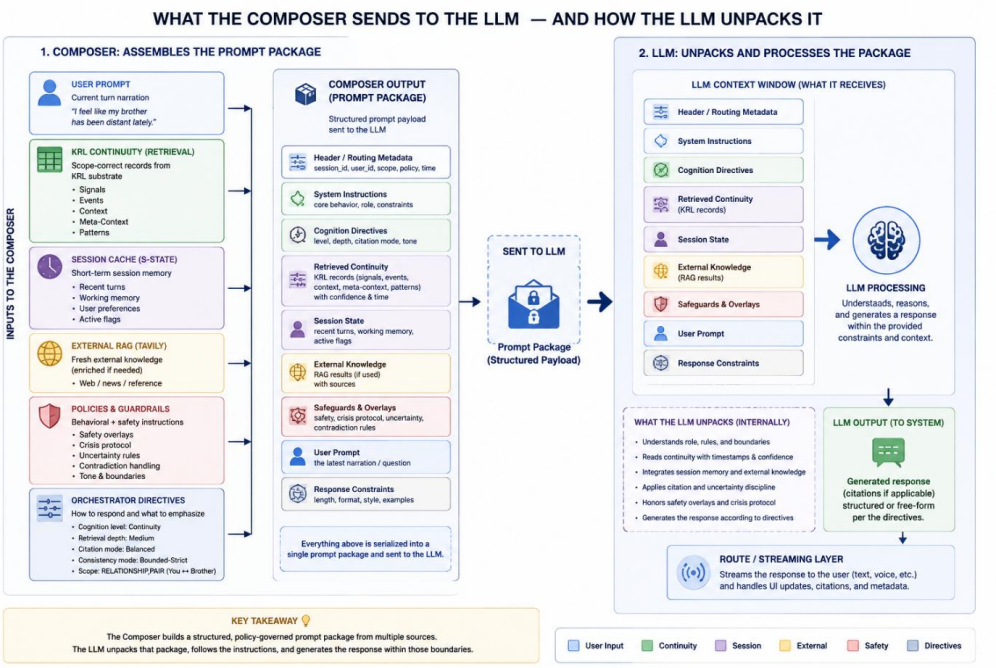

Figure 3 — Ontology and UI mapping: the substrate's four subject scopes map directly to user-facing surfaces. The extraction pipeline populates each scope from the same narration stream.

4.1 A Full Extraction Trace: "I'm feeling tired."

To make the pipeline concrete, follow a single user message through every stage. The user opens their Space Journal for the Family space and types: "I'm feeling tired. Mike said he'd handle dinner but he forgot again."

Stage	Operation	Output
1. Admission	POST arrives at app/routers/journal.py; @-mentions resolved via app/services/mentions.py; mention_uuids and entity_name_map constructed	mention_uuids = {mike.id}
2. Orchestrator	JournalOrchestrator.run() resolves CognitionPolicy, assembles continuity block, builds messages array, streams AI response via SSE	User-visible prose response

Stage	Operation	Output
3. Parallel Extraction	Six extractors fire concurrently via asyncio.gather against the same raw user message; space_filter_extractor first determines the active space-context routing.	See below
4. Persistence	Each proposal flows through upsert_record() chokepoint; layer-specific semantics applied; provenance=user_narration stamped	2 SIGNAL rows, 1 EVENT row, 1 RELATIONSHIP_PAIR row, subject to space-filter routing and dyad-participation inference
5. Pattern Engine	Scoring orchestrator evaluates all registered templates against freshly-updated substrate; pattern lifecycle states updated	UnreliabilityCycle for ENTITY(Mike) bumped or activated
6. Version Bump	cognition_state row for (user_id, space_id) has pattern_settle_version incremented	Downstream reads see new version
7. Trace	cognition_trace row written capturing full lifecycle: policy resolved, tokens loaded, patterns referenced, extractor counts, settle timestamps	Observability record for debugging

Key Invariant

Six words from the user produced two SIGNAL rows, one EVENT row, one RELATIONSHIP_PAIR row, one pattern-engine evaluation, one cognition-state version bump, and one cognition-trace row. None of the AI's generated response was ingested. The system learned only from the human.

4.2 The Six Extractors

Each extractor is a single-purpose LLM job with a tightly scoped prompt, a structured-output JSON schema, and a name-space input contract. The LLM emits names ("Mike," "user," "work"); the persist layer resolves names to UUIDs immediately before writing. This contract — captured in the locked memory feedback_llm_emits_names_not_uuids — eliminates an entire class of subtle hallucination bugs where the LLM invents UUIDs that do not exist in the substrate.

Extractor	Writes Layer	Persist Semantics	Notable Rules
signal_extractor	SIGNAL	Append-only insert	Decay half-life applied at retrieval; polarity required; output includes behavioral markers such as withdrawal, avoidance, and kept commitments
event_extractor	EVENT	Append-only insert	Participants required; references_event_ids enables cross-time

Extractor	Writes Layer	Persist Semantics	Notable Rules
			linkage; narrated_anchor parser handles "the thing back in June"
context_extractor	CONTEXT	Upsert with confidence + contradiction gates	User-override flag respected; high_stakes slots require confidence ≥ 0.95; no silent overwrite
meta_context_extractor	META-CONTEXT	Upsert with drift versioning	Contradiction delta 0.2 surfaces clarification; trauma-aware mode silently skips ambiguous high_stakes claims; separates objective context from subjective narrative
relationship_pair_extractor	RELATIONSHIP_PAIR records (any layer)	Deterministic dyad hash via uuid5(NAMESPACE_OID, sorted pair)	Subject_id is always the dyad UUID regardless of write order; analyzes trust erosion, reciprocity imbalance, and repair attempts
space_filter_extractor	ROUTING / SPACE CONTEXT	Pre-write routing decision	Determines which space context applies to a turn; prevents cross-space contamination before downstream extractors persist records.

Extractor outputs are forced through strict structured-output schemas with typed enums, required fields, and validation enforcement. Invalid outputs are retried once with validation feedback and discarded entirely on second failure — the system prefers omission over substrate corruption. Before persistence, records are filtered through centralized confidence thresholds, unresolved entities are rejected rather than auto-created, and provenance metadata is attached as a mandatory invariant including extractor type, extractor version, source message, and write timestamp.

Extractor-to-Scope Mapping

Extractors themselves do not determine which KRL scope a record belongs to. The model emits semantic references ("user" or "Mike"), and a downstream resolver maps those references onto known subjects before assigning scope at persistence. This creates a grid architecture where extractors function as the analytical lens and scopes function as the subject classification layer:

Extractor	USER	ENTITY	SPACE	RELATIONSHIP_PAIR
Signal	✓	✓	✓	
Event	✓	✓	✓	
Meta-context	✓	✓	✓	
Context	✓	✓	✓	
Relationship-pair				✓
Space filter	✓	✓	✓	

4.3 The Canonical Persistence Chokepoint

Every extractor write — and there are no other writes to the KRL — flows through upsert_record() in app/repositories/extracted_records.py. This is the single chokepoint at which all KRL discipline lives. Concentrating the discipline at one boundary instead of spreading it across six extractors is a deliberate design decision: one-place normalization keeps a growing system operable.

The chokepoint applies the following checks in order:

Check	Logic	Failure Behavior
Provenance check	Reject if provenance ≠ user_narration	Write rejected outright (Phase L-1 invariant; prevents recursive substrate ingestion)
Layer routing	SIGNAL and EVENT → append-only; CONTEXT and META-CONTEXT → upsert path	Separate code paths per semantic contract
Upsert lookup	Lookup by (user_id, subject_kind, subject_id, layer, subcontext)	If no existing row: insert directly
Confidence floor	new confidence < 0.5	Write rejected
Contradiction delta	new confidence exceeds existing by < 0.2	Emit clarification proposal; surface to user next turn
Contradiction delta (pass)	new confidence exceeds existing by ≥ 0.2	Silent overwrite with old value archived to extracted_records_history
User-override	user_overridden = True	Refuse AI write unless force=True (manual intervention only)
Drift history	Before any overwrite	Old (value, confidence, subject_aspect, perspective) tuple archived before new value lands
Soft-delete	Row has deleted_at set	Excluded from upsert resolution; new row inserts as if no prior existed

4.4 Name-to-UUID Resolution

All extractor prompts work in name-space, not UUID-space. The LLM receives the user's space-assigned entity roster as names and roles ("Mike — Sarah's brother, Sarah — partner, Bob — manager") and emits names as outputs. The persist layer resolves names to UUIDs via the entity_name_map dictionary passed in from the journal router. The resolution happens at the persistence boundary, never during the LLM call, eliminating the possibility of the LLM emitting a UUID that does not exist.

A bare-name fallback in app/services/mentions.py parses alphabetic tokens for un-tagged references ("Sarah came through" is resolved even without @Sarah). The inverse rule — the @-selector is a pure projection over space-assigned entities, never a creation mechanism — is a locked architectural invariant. If the user mentions a name not in their space's roster, the system does NOT auto-create the entity. Entity creation is explicit, manual, cap-gated, and monetized; auto-creation from narration would create slot-waste and break the paywall.

4.5 Clarification Surfacing

When the contradiction-delta gate fires below 0.2, the chokepoint does not silently overwrite the existing record. It also does not reject the new claim outright. It writes a clarification row (a special layer=clarification record with a 7-day TTL) that surfaces on the next turn as a Known Facts block in the continuity prompt: "Earlier you mentioned X, now you're saying Y — which holds?" The user resolves the contradiction by reasserting one or the other; the resolution is re-extracted at confidence ≥ 0.95 and the gate is cleared. This mechanism maintains a one-turn lag on contradictions but never permanently loses any claim until the user explicitly retires it.

PART V

The Pattern Engine

The pattern engine is what users pay for. The KRL alone is just an organized archive; what makes the product valuable is the engine that compounds individual records across time, subject, and dimension to surface relational and behavioral patterns the user could not see on their own. The engine is the product moat. Pattern detection plus multi-layer context richness is the core value proposition, and migration cleanliness around the engine is non-negotiable.

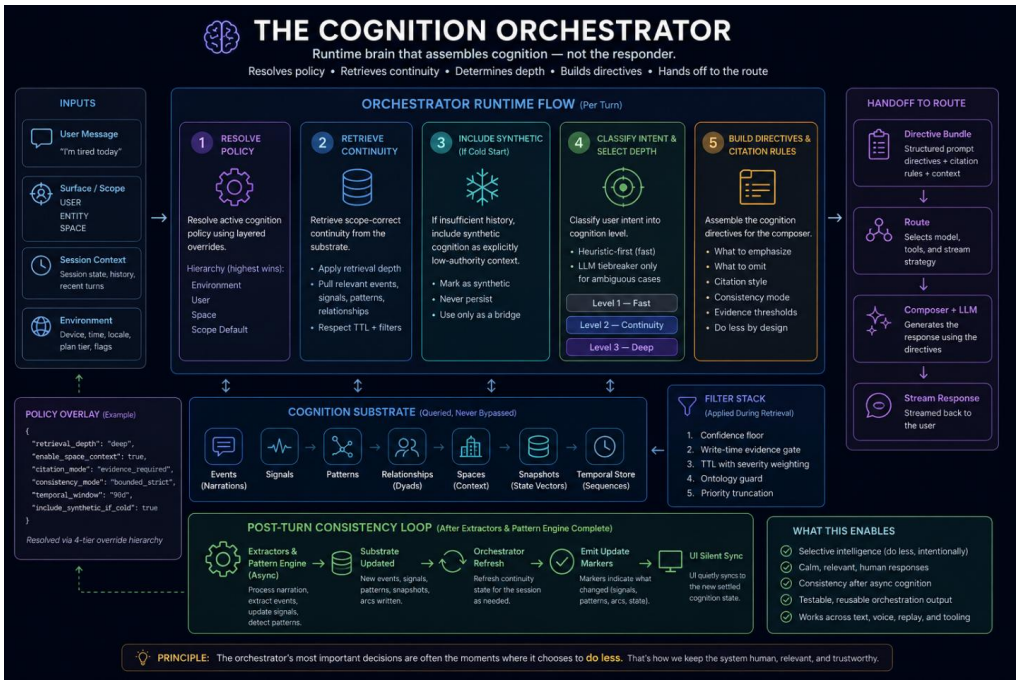

Figure 4 — The compounding filter stack: five filters between raw narration and the prompt, each stripping a different class of noise. The pattern engine sits downstream of all five.

5.1 The Four-Tier Escalation

The engine operates on a four-tier escalation hierarchy. Each tier is a strictly larger compositional structure than the previous one, with Tier 4 reading from all four substrate layers (SIGNAL, EVENT, META-CONTEXT, CONTEXT) and across all four subject scopes. Patterns at higher tiers can have lower-tier patterns as their parents, encoded in the parent_pattern_ids JSONB column on user_pattern_bank. Higher tiers require lower-tier patterns to already exist — this ladder enforces a discipline against premature thematization.

Tier	Construct	Composition	Example
1	Signal	Single moment	Bob dismissed user in meeting
2	Pattern	Cluster of signals, event sequence, or meta-context drift on one subject	Bob's dismissal-reciprocity-withdrawal cycle

Tier	Construct	Composition	Example
3	Multi-pattern	Multiple simultaneous patterns on the same subject	Bob withdrawal + Bob political-rivalry rise
4	Compound	Multi-patterns crossing dimensions and/or layers and subjects	Bob relational breakdown = ENTITY(Bob) signal pattern + USER signal pattern + ENTITY(Bob) event sequence + ENTITY(Bob) meta-context drift + SPACE meta-context corroboration

Compound patterns are what allow the engine to diagnose arcs, not moments. The classic Bob-relational-breakdown example crosses three layers and corroborates across two subjects (entity + space). The compound surfaces something the user could not have seen by looking at any single layer in isolation.

5.2 The 43-Pattern Registry

The engine ships with 43 patterns registered as canonical detectors in a single-source registry — one place per the feedback_extension_points_central_registry rule. The registry lives in app/services/scoring/pattern_templates.py: 29 templates declared in the TEMPLATES list, 14 declared in NON_TEMPLATE_DEFINITIONS (cluster/arc/timing/absence detectors with programmatic implementations), and a canonical DNS table backfilled on every template via the _HUMANIZATION map and the _augment() helper.

Patterns are classified along two locked axes (locked 2026-05-27):

Axis	Values	Description
PATTERN_DOMAINS (8)	trust, communication, power, connection, conflict, growth, wellbeing, cognition	The human concern the pattern addresses
PATTERN_FORMS (7)	cluster, cycle, arc, trend, absence, rhythm, composite	The structural shape of the pattern

Every pattern carries a six-field contract: internal name (canonical engine identifier), human_label (what the user sees), plain_meaning (one-sentence non-clinical translation), scope (subject_kind it applies to), domain, and form. These six fields are sufficient for the KRL Explorer surface to present the pattern coherently, for the cross-space whiteboard to enumerate patterns by category, and for the LLM to respond about a pattern without leaking its internal canonical name into user-facing output.

5.3 Detector Families

Detectors are organized by family. Each family lives in its own file under app/services/scoring/ and registers with the scoring orchestrator via the patterns/registry. Implementations are deliberately scattered across many small files (one family per file) rather than concentrated in a monolithic engine — this keeps cognitive load per file low and makes adding a new detector family a single-file change.

Family	File	Notable Detectors
Arc detectors	arc_detection.py	RepairThenRelapse, TrustRebuilding, EscalationArc, WithdrawalSpiral

Family	File	Notable Detectors
Timing detectors	timing_detector.py	TemporalClusteringPattern (signals dense in a window suggest acute stress); cross-entity same-day correlation; anticipatory leading-indicator patterns
Cross-layer rules	cross_layer_rules.py	ContextualDivergencePattern (SpaceRecoveryArc in one space while SpaceStress in another)
Drift detectors	drift_detector.py, self_narrative_drift_detector.py	AspectDrift (subject_aspect weight shifts); Self-narrative drift (USER meta-context contradicts USER signal evidence)
Absence detectors	absence_detection.py + absence_rules.py	EngagementGap (user stopped engaging with an entity); MissingEmpathy (trigger fired but expected response absent)
Template detectors	pattern_templates.py TEMPLATES list	SpaceRecoveryArc, SpaceStressEscalation, SpaceFunctionalDecay, and 26 others
Trajectory detectors	trajectory_detector.py	Tier-4: Intensifying, Fading, Reversal — second-order patterns about how other patterns are changing

5.4 The Evidence Formula and Upsert Chokepoint

All detector writes flow through one function: upsert_pattern() in app/repositories/patterns.py. Like the extracted-records chokepoint, concentrating discipline at one boundary is the only sustainable design at scale. The most important computation at this chokepoint is the evidence_signal_count formula:

```
evidence_signal_count = max(len(parent_pattern_ids),
observations_count, 0)
```

GREATEST rather than SUM because the two counters measure overlapping evidence — counting both would double-attribute. GREATEST rather than a conditional (parent_count if compound else observations_count) because the formula must be uniform; one-place normalization is the invariant. The formula was stabilized at write-time in the upsert chokepoint as Phase J.1 follow-up; previously detectors were responsible for setting evidence_signal_count themselves, producing an O(N) drift problem across the 43 patterns.

5.5 Pattern Lifecycle States

Patterns are not point-in-time judgments. They have a lifecycle tracked explicitly via the lifecycle_state column on user_pattern_bank. The five canonical states and their transitions:

State / Surface Label	Persisted?	Meaning and Derivation
active	Yes	Pattern is eligible to surface when evidence is recent, strong, and not suppressed. Persisted lifecycle_state value; may receive computed surface labels.

State / Surface Label	Persisted?	Meaning and Derivation
dormant	Yes	Pattern is preserved in the bank but excluded from ordinary prompt injection. Persisted lifecycle_state value; can reactivate on new evidence; reactivation_count increments.
changing	Computed label	Pattern character is shifting, often due to subject_aspect drift or contradictory meta-context. Derived from drift fields and recent evidence, not a lifecycle_state enum.
repeating	Computed label	Pattern has fired multiple times across observations or event sequences. Derived from recurrence/evidence counts and temporal clustering.
weakening	Computed label	Previously surfaced pattern has sparse recent evidence, suppression, or declining signal density. Derived from evidence_signal_count, suppressed_until, is_overridden, and recency windows.

The lifecycle is preserved in the bank even when dormant. This is critical for the reactivation_count column: when a pattern goes dormant and then reactivates, the engine records that reactivation as its own signal of relational dynamics. The third time Mike forgets dinner after twelve months of reliability is more meaningful than the first; the reactivation_count column makes this visible to the orchestrator and the analytic voice.

5.6 Confidence, Decay, and Time-Weighting

Confidence is per-record and per-pattern. Each record carries a confidence value 0.0–1.0 set by the extractor at write time. Patterns inherit a confidence bounded below by their parent patterns' minimum confidence — the child-floor rule — ensuring a compound cannot be more confident than its weakest component, which forces honesty about uncertainty.

Time-decay is applied at read time, not write time. Signals lose effective weight as they age, with a per-subcontext decay half-life encoded in the signal_views aggregation. The same signal carries high weight three days after writing and lower weight three months later, without ever modifying the persisted record. This decouples storage cost from temporal calibration: signals are kept forever for audit purposes, but their effective contribution to the prompt fades.

The pattern engine's confidence math for Tier 3 and Tier 4 compounds is a weighted average across parent patterns with a recency boost: more-recent evidence within the compound contributes more to the final confidence. The exact weighting is calibrated against a ground-truth corpus in the engine evaluation framework (Phase 25.6); the current production weights are deliberately conservative.

5.7 Time-to-Live and the Signal-Noise Discipline

Most cognition systems treat older data as permanently equal in weight to new data, which quietly lets the past dominate through accumulation. A difficult month from two years ago can permanently shade the system's view of a user. MetaOp AI handles this differently.

Signals, events, and meta-context records gradually lose weight as they age rather than remaining equally important forever. Older records are not deleted — they remain available for history views and analytics — but recent signals carry more weight in pattern detection than distant ones. Ten withdrawal signals from the last two weeks produce a much stronger pattern than ten similar signals spread across two years.

Operational cognition traces exist only for debugging and system analysis, not for long-term continuity, so they are automatically deleted after seven days. The user's actual continuity inside the KRL substrate is not deleted over time; it uses weighted decay, where older continuity gradually matters less unless it becomes relevant again through reactivation.

Patterns have TTL governed by surfacing tier, with severity weighting: critical patterns never expire; high patterns expire after 180 days; medium patterns after 90 days; low patterns after 60 days. Raw signals and context records are NOT subject to TTL — they are permanent substrate; only patterns expire.

5.8 The Pattern Safety Charter

The locked project_pattern_safety_charter memory is foundational and must be respected by every change to the pattern engine. Its key invariants:

- Clinical-adjacent taxonomy is internal-only. Patterns may carry safety_class="clinical_adjacent" or "metacognition" internally for routing, but those classifications never appear in user-facing prose.
- Surface is observable behavior, never diagnosis, motive, or identity. The user sees what was observed; they do not see a verdict on what it means about a person's character or psychology.
- Three-scope surface rule. For ENTITY scope, the surface is interactional ("how this person showed up"). For USER scope, the surface is reflective ("how you have been moving through this"). The system never surfaces identity-level diagnostic claims about anyone.
- USER metacognition family is permitted with care. Patterns about the user's own thinking patterns are allowed when observable from the user's own narration.
- Wellbeing overlay. CRITICAL safety-class patterns route to crisis-protocol resource references (988, Text HOME 741741, 911) via the shared shared_safety.py module, separated from the extractor pipeline and never blended with diagnostic claims.

5.9 The Pattern Humanizer

What the LLM sees about a pattern is not what the user sees about a pattern. The pattern humanizer in app/services/cognition/pattern_humanizer.py mediates between the internal canonical name (a technical identifier like UnreliabilityCompoundCycle that should never reach a user-facing surface) and the user-facing human_label ("recurring follow-through gap"). The strip-before-prompt rule is enforced here: the orchestrator's pattern injection only sends humanized labels and plain_meanings to the LLM, never internal canonical names. The LLM's response is then post-processed by the blocked-language scan (currently observability-only) to detect leakage of clinical or diagnostic vocabulary into output.

PART VI

Cognition Orchestration

The orchestrator is the runtime spine. It receives a user message, resolves the active CognitionPolicy, assembles the continuity context, optionally consults the synthetic blob, builds the messages array for the LLM, streams the response, and triggers the extraction and pattern tail. Everything that happens at the cognition layer happens through it.

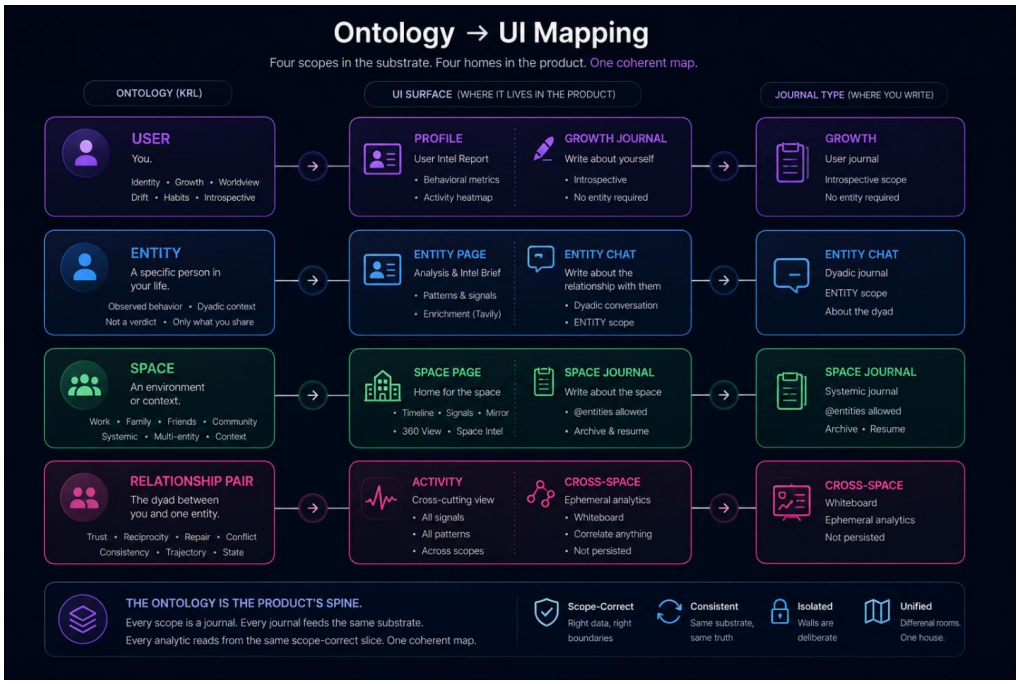

Figure 5 — Cognition orchestrator concept and multi-layer prompt assembly. The orchestrator resolves policy, retrieves substrate, assembles continuity, and delegates rendering to the composer.

6.1 The Base Orchestrator and Its Subclasses

JournalOrchestrator in app/services/cognition/orchestrator.py is the base class. It defines run(), refresh_continuity_after_extraction(), _consult_synthetic_blob(), and the shared retrieval primitives. Three subclasses inherit from it and differ only in their default policy and scope name. The 90% code-share property is enforced by inheritance — adding a new scope is a single-file change.

Subclass	Scope	Default Policy	Characteristics
UserJournalOrchestrator	USER	USER_JOURNAL_POLICY	Reflective notebook; inherits run() unchanged; shallow retrieval depth; balanced citation; bounded_strict consistency
EntityJournalOrchestrator	ENTITY	ENTITY_JOURNAL_POLICY	Per-entity notebook; deep retrieval depth; strict citation; bounded_strict consistency; most analytical scope

Subclass	Scope	Default Policy	Characteristics
JournalOrchestrator (SPACE default)	SPACE	SPACE_JOURNAL_POLICY	Multi-actor environmental notebook; SPACE uses the base JournalOrchestrator directly with SPACE_JOURNAL_POLICY rather than a literal SpaceJournalOrchestrator subclass.

6.2 The run() Pipeline

Inside run(), the sequence is consistent across all three scopes. Subclasses do not override the pipeline; they configure it via their default policy and scope name. The table below describes the SPACE_JOURNAL case (most complex):

Step	Operation	Output
1	Resolve CognitionPolicy via 5-tier resolver	Active policy (retrieval_depth, citation_mode, consistency_mode, etc.)
2	Load behavioral metadata + voice_context	Tonal calibration for prompt construction
3	Build continuity_block via continuity_block.build_continuity_block()	Token-capped prompt segment with KRL retrieval across four dimension loaders
4	Consult synthetic_cognition_blob via _consult_synthetic_blob()	Up to 3 low-authority interrogative entries from session blob
5	Build messages via conversation.build_messages() with scope_name	Prompt array with SCOPE_IDENTITY_<X> block appended
6	Stream LLM response via SSE (frame_delta + frame_terminal)	User-facing prose; tokens stream as they arrive
7	In parallel: run extractors via _journal_pipeline_helpers	Extractor proposals → upsert_record() chokepoint → KRL updates
8	Run scoring_orchestrator over freshly-updated bank	Pattern detections; pattern_settle_version bumped
9	If background_cognition flag on: schedule run_cognition_tail off request path	Async settlement; trace settled_at stamped post-tail
10	Persist cognition_trace with full lifecycle payload	Observability substrate row for debugging and auditing

6.3 The OrchestratorResult

Every run() returns an OrchestratorResult. The result type is the contract by which downstream code (the SSE framer, the trace persister, the cross-space whiteboard) consumes the orchestrator's output without reaching into orchestrator internals.

Field	Type	Purpose
cognition_level	string	Substrate density classification (sparse / moderate / dense)
intent_triggers	list[string]	Triggered analytic intents (analytic_voice, brainstorm, dictionary, correction)
continuity_block_text	string	Final assembled continuity block prompt segment
pattern_injection_block	string	Pattern-tier injection block from pattern_injection.py
signal_injection_block	string	Signal-tier injection block from signal_injection.py
cognition_level_directive	string	Sparse-substrate directive when applicable (no hallucinating ontology)
high_stakes_meta_active	bool	Trauma-aware mode flag for high_stakes slots
mention_uuids	list[UUID]	Resolved @-mention UUIDs from this turn
pattern_settle_version_at_load	int	Settle version observed at continuity load (for consistency tracking)
consistency_mode	string	Active consistency mode (eventual / bounded_strict)
analytic_intent	string|null	Resolved analytic-voice sub-intent if applicable

6.4 Selective Intelligence: The Three Cognition Levels

The single most important architectural principle governing orchestrator behavior is selective intelligence. The instinct when building AI products is to do more on every turn: load more context, run more analyses, surface more patterns, generate longer responses. MetaOp AI resists this instinct deliberately. Before running anything expensive, the orchestrator asks: does this query actually need it?

Selective intelligence is implemented as a three-level cognition hierarchy. Classification is heuristic-first — a small regex layer running in approximately one millisecond — with an optional LLM tiebreaker only for genuinely ambiguous mid-band messages. Heuristics handle the vast majority of cases.

Level	Name	Trigger Conditions	Behavior
1	Fast Pass	Low-signal narration, casual message, simple acknowledgment	Lightweight summary, basic signal extraction, minimal continuity lookup. No deep pattern lineage, no predictive trajectory, no relationship-state evolution. Responses are short and respect the user's energy.
2	Continuity Pass	User mentions a specific entity, narrates a recent event with emotional weight, or engages with a recurring topic	System retrieves related events, unresolved arcs, dyadic continuity, and active pattern outputs. Responses honor history without becoming clinical.
3	Cognitive Pass	User explicitly invites analysis, asks for instances, requests pattern lineage, or	Full pattern engine runs; evidence is cited with confidence framing; response engages the user's analytical intent at full depth.

Level	Name	Trigger Conditions	Behavior
		surfaces something warranting major retrieval	

Early Development Lesson

Running every turn at maximum cognition depth produced thoughtful, dense analyses for users who had just said "feeling alright today." The responses were technically impressive and experientially wrong — they read as if the AI was straining to prove it was paying attention. After implementing the cognition hierarchy, low-signal messages began receiving short conversational responses, and analytical depth was reserved for moments when users actually asked for it. The system felt calmer and more present.

PART VII

Cognition Policy and Runtime Governance

Cognition policy is the runtime governance layer. It determines what the orchestrator loads, how strict the citation discipline is, whether synthetic cognition fires, what tonal posture the prompt adopts, and whether consistency is eventual or bounded_strict. Policy is resolved per-turn from a five-tier hierarchy, so the same orchestrator can behave very differently depending on the user, the space, and the active environment override.

7.1 The CognitionPolicy Contract

CognitionPolicy is the Pydantic contract in app/services/cognition/policy.py. The contract is intentionally flat — no nested policy objects — so fields are individually addressable and the resolver can merge from different tiers without traversing nested structures.

Field	Type	Default	Effect
retrieval_depth	enum: shallow/standard/deep	standard	PATTERN_CAP at retrieval (3 / 8 / 12 patterns)
enable_space_context	bool	true	Whether SPACE-scope continuity loads into prompt
citation_mode	enum: strict/balanced/implicit	strict	How strictly the LLM is required to cite KRL provenance in output
consistency_mode	enum: eventual/bounded_strict	eventual	Whether continuity refreshes after extraction before next prompt
activate_continuity	bool	true	Master switch for the continuity block (rarely off)
enable_tiebreaker	bool	true	Whether LLM tie-breaker fires for ambiguous intent classification decisions
blocked_language_scan	bool	false	Post-generation scan for clinical/diagnostic vocabulary (currently dark / telemetry-only)
humanize_pattern_surface	bool	true	Apply pattern humanizer to all surfaced pattern names before prompt injection
awareness_tier	int	1	Phase 25.3.b awareness gate — how much pattern awareness is surfaced
action_tier	int	1	Phase 25.3.b action gate — how prescriptive the orchestrator may be
Full 25+ field contract	see §14.1 / §14.9	policy-specific	Full contract includes meta-context, space dynamics, meta-patterns, predictive trajectory, relationship propagation, analytic

Field	Type	Default	Effect
			voice, correction routing, and background cognition flags.

7.2 The Three Scope Default Policies

Each cognition scope has a default policy reflecting its identity. The differences between the three are calibrated to match the scope identities locked in the cognition charter.

Field	USER	ENTITY	SPACE	Rationale
retrieval_depth	shallow	deep	standard	ENTITY is the most analytical scope (users arrive with "what patterns do you see?" questions); USER leans conversational and introspective; SPACE balances both.
enable_space_context	false	false	true	Only SPACE journal narration is inherently environmental; USER and ENTITY journals are intentionally scope-isolated.
citation_mode	balanced	strict	strict	USER scope supports introspective conversational tone; balanced citation fits. ENTITY and SPACE require strict citation because users depend on evidence-grounded outputs.
consistency_mode	bounded_strict	bounded_strict	eventual	SPACE tolerates eventual consistency because the complex multi-actor prompt leaves less budget for refresh latency; validated empirically.

7.3 The Five-Tier Policy Resolver

resolve_policy() in app/services/cognition/policy_resolver.py walks five tiers in priority order, merging fields from each onto a shared dict. Higher-priority tiers shadow lower ones. The hierarchy:

Tier	Source	Notes
1 (highest)	Environment override: os.getenv("METAOP_POLICY_<SCOPE>_<FIELD>")	Used for live experiments and dark-shipped feature flags; beats all other tiers unconditionally

Tier	Source	Notes
2	Per-user override: users.cognition_policy JSONB	Reflects an informed user choice; environment can tighten but user cannot override environment
3	Per-space override: spaces.cognition_policy JSONB	Used for spaces with special governance (high-stakes relationship spaces)
4	Scope default: USER_/ENTITY_/SPACE_JOURNAL_POLICY	Loaded from policy_registry.py; calibrated to scope identity
5 (lowest)	DEFAULT_POLICY: global fallback	Rarely reached in production; safety net only

The hierarchy is one-way: lower tiers cannot override higher ones. If an environment override has tightened citation strictness fleet-wide, no individual user or space override can loosen it. If a user has opted into a permissive citation mode at the per-user tier, the per-space tier cannot tighten it. This one-way enforcement is critical for safety and for A/B experiment integrity.

7.4 Async Cognition Consistency

The user's LLM response streams back fast — that is what the user is waiting for. The cognition pipeline that processes narration (extractors, pattern recomputation, snapshot updates) runs after the stream finishes. If the user re-asks immediately, the new substrate state may not have settled yet, and the model would answer based on stale patterns.

Mode	Behavior	Use Case
Eventual (default)	Response streams immediately; substrate updates in the background; next turn sees freshly settled patterns	SPACE scope default; ordinary narration turns; high-volume surfaces
Bounded-strict	Emits a settle-version marker over the same SSE stream when post-turn cognition finishes; UI silently refreshes	USER and ENTITY scope default; removes the re-ask cliff without blocking every turn on the slowest layer
Strict	Blocks the response until substrate has fully settled	Analytical questions where user explicitly requests the latest state; deliberate latency trade-off

PART VIII

The Continuity Block and Synthetic Cognition

8.1 The Continuity Block

The continuity block is the structured KRL summary that the orchestrator assembles before every LLM call. It is not a chat history; it is a directed retrieval of the most relevant cognition substrate for the current turn, organized by dimension and tagged with provenance and confidence. The block is the LLM's only window into the durable KRL during a turn; everything the LLM knows about the user's broader cognition comes through this block.

The Four Dimension Loaders

Inside build_continuity_block() in app/services/prompts/continuity_block.py, four loaders run in parallel against the KRL. Each is responsible for one dimension of the matrix.

Loader	Reads	Produces
_load_user_layer / _load_meta_context	USER-scoped extracted_records	USER signals + meta-context + recent self-narration
_load_entity_layer / _load_dyad_context_for_entities	ENTITY + RELATIONSHIP_PAIR records for @-mentioned entities	Per-entity briefs + dyad signals
_load_space_layer / _load_space_context	SPACE-scoped records for active space	Environmental facts + space-level patterns
_load_event_sequences (Phase H-5)	EVENT records linked by references_event_ids	Multi-event narration arcs enabling temporal reasoning

Pattern Injection and Sparse Substrate

Pattern injection is layered on top of the four loaders by app/services/cognition/pattern_injection.py. The injector enumerates the most relevant patterns for the current turn (filtered by lifecycle_state ∈ {active, changing, repeating} and confidence ≥ humanize floor), humanizes them via pattern_humanizer, and formats them into the prompt as an authoritative block. The cap is determined by the active policy's retrieval_depth (3 / 8 / 12 patterns).

When a scope has zero patterns, sparse-substrate directives fire instead. Three builders cover the three scope cases, each emitting an explicit "NO STABLE PATTERNS" directive plus the operative rule ("do NOT attribute other entities' patterns to @Name," "do NOT invent ontology to fill the scope," "acknowledge the absence"). These directives are the structural defense against ontology hallucination on sparse substrate.

The Dual-Tagged Block

When the synthetic blob has eligible entries, the continuity block emits two distinctly framed sub-blocks. The DURABLE HUMAN CONTEXT block comes first with authoritative framing and contains the retrieved KRL substrate. The SYNTHETIC SUBCONTEXT block comes last with explicit LOW-AUTHORITY framing and contains matched synthesizer entries. The LLM is told in both block headers that synthetic content cannot override durable evidence and must be

phrased interrogatively when referenced. Without this explicit prompt-level discipline, the architectural separation between durable and synthetic collapses at runtime.

The Token Budget

The continuity block has a hard 8,000-token cap (CONTINUITY_BLOCK_MAX_TOKENS in continuity_block.py). When the assembled block exceeds the cap, the trimmer drops content in priority order: synthetic-blob entries first, then weakening patterns, then meta-context entries below median confidence, then non-recent events, then signals older than the active decay window. Clarifications and active high-confidence patterns are preserved at priority 100 (highest) and dropped only at extreme overruns.

8.2 Synthetic Cognition — the Cold-Start Layer

The synthetic-cognition system answers a specific epistemic problem: on a brand-new user with no KRL substrate, the orchestrator's continuity assembly returns nothing, and the LLM is forced to either fabricate cognition to fill the prompt-rule slot or surface an apologetic "I don't know anything about you yet" message. Both are bad outcomes. The synthetic-cognition layer provides a third option: bounded, low-authority, interrogative scaffolding that keeps the conversation feeling continuous without polluting the durable substrate.

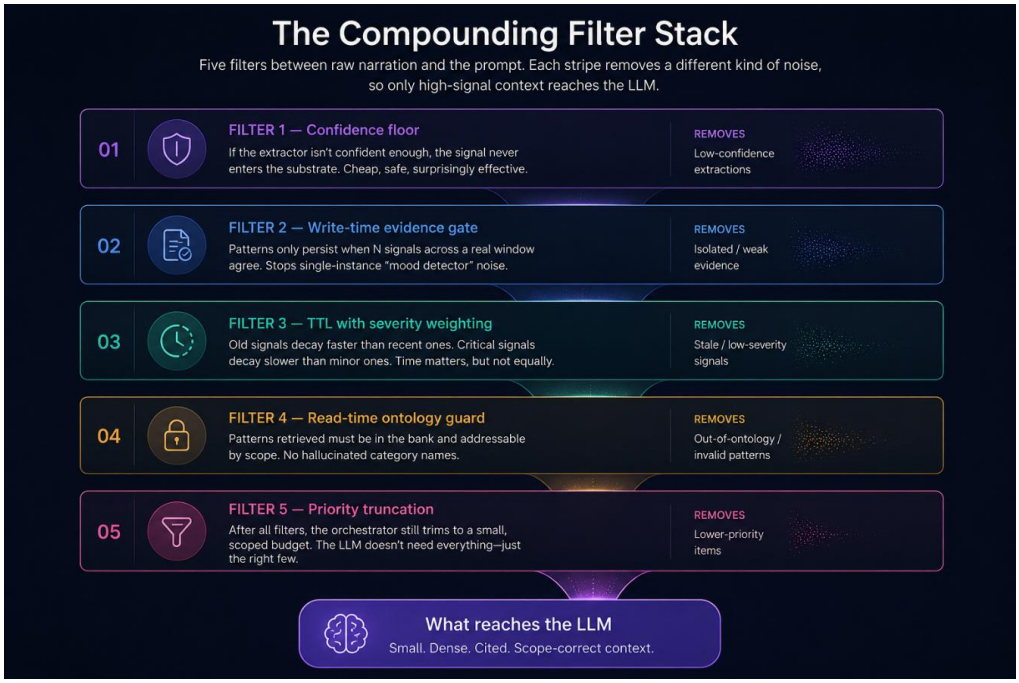

Figure 6 — Azure-oriented application and network deployment concept: the cognition substrate is durable across model provider changes; the model layer remains swappable infrastructure.

Two Stores with Different Physics

The Phase L design is not policy on one store; it is two stores with different physics. The durable KRL is cross-session, evidence-grounded, authoritative, and the primary cognition substrate. The synthetic store is conversation-scoped, transient, low-authority, and the AI's bounded working memory for one specific journal target.

Property	Durable KRL	Synthetic Blob
Storage	extracted_records + user_pattern_bank tables	JSONB column on conversations table
Lifetime	Cross-session, permanent	Conversation-scoped; deletes with conversation
Authority	Authoritative — primary truth	Low-authority — interrogative scaffolding only
Provenance	user_narration (locked by Phase L-1)	synthesizer:<name> (always synthetic-tagged)
Written by	Six extractors via upsert_record() chokepoint	Registered synthesizers via SynthesizerContract
Cap	Effectively unbounded (subject to tier caps)	30 entries hard cap with prune-oldest
Surfaces in UI	Yes — KRL Explorer, Formation, Evidence Chain	Never — invisible to users; shapes prompts only

The Hourglass Dynamic

The synthetic blob is most active when durable substrate is sparse (new user, new space). As durable accumulates from real human narration, the orchestrator's retrieval naturally returns rich human content, and synthetic-blob entries get crowded out of the prompt by durable evidence. No governance machinery is required for this to work — the orchestrator already prioritizes durable retrieval in continuity assembly. Synthetic simply does not get reached when durable returns enough.

This is the hourglass metaphor: early in a user's lifetime the system leans more heavily on temporary synthetic scaffolding because durable continuity is sparse, but as real user-generated cognition accumulates, the orchestrator increasingly prioritizes the KRL while the synthetic cache becomes progressively irrelevant. Over time, the temporary scaffolding naturally tapers away as durable continuity compounds beneath it. This is what makes temporal continuity the premium-tier wedge: three months of substrate produces arcs, six months produces inflection points, twelve months produces longitudinal trajectory. The longer the user stays, the more uniquely valuable the substrate becomes.

Scope Isolation — Locked Invariant

Each Conversation's blob isolates fully to its scope (USER / ENTITY / SPACE). A USER_JOURNAL blob never informs an ENTITY_JOURNAL prompt for the same user, and vice versa. This is a hard architectural rule, not a tunable policy. Cross-scope sharing would require designing a cross-scope authority model that is explicitly out of scope for v1.

What Synthetic Cognition Does Not Do

The synthetic-cognition system never surfaces in AI responses, never gets cited, and never appears as a UI element labeled "what the AI suspects." It shapes the AI's reasoning only. This closes the epistemic-laundering risk by construction: with no surface, there is no user-mediated reinforcement loop in which the user agrees or disagrees with a synthetic claim, thereby converting it to durable cognition. Synthetic cognition that recursively ingests its own outputs

eventually drifts into hallucination — a kind of epistemic feedback loop. The strict physical separation is the firewall.

PART IX

Temporal Continuity

The temporal continuity layer adds sequence awareness to the existing pattern engine and substrate: intra-turn ordering, anchor parsing, arc detectors, event correlation, and relationship state snapshots. This layer allows the product to reconstruct how an issue evolved rather than treating every journal entry as an isolated turn.

9.1 Intra-Turn Sequencing

A single narration often contains multiple temporally-ordered events. "This morning in standup, Mike interrupted me. Then at lunch, Sarah brought it up unprompted. By end of day, Mike actually apologized." The event extractor now emits temporal_order and narrated_anchor fields so the substrate captures the sequence, not just the events. Downstream pattern detectors can then reason about "what came before what" within the same narration, and arc detectors can identify repair sequences embedded in single turns.

9.2 Anchor Parsing

Users narrate in natural temporal language: "last Tuesday," "that first weekend," "a month after the wedding." The narrated_anchor parser resolves these to wall-clock estimates with confidence, enabling the substrate to answer questions like "what was happening around the time of the wedding" without forcing the user to enter calendar dates. The parser produces estimates rather than exact timestamps, and the confidence bound on the estimate propagates through to the records it anchors.

9.3 Arc Detectors

Four arc detectors are deployed: RepairThenRelapse, TrustRebuilding, EscalationArc, and WithdrawalSpiral. Each reads the substrate's temporal sequence of relevant signals and fires when a recognizable trajectory has occurred. Arcs surface as a different kind of pattern — narrative rather than aggregate — and the composer is instructed to honor the trajectory framing when surfacing them. A user experiencing a TrustRebuilding arc followed by a ReliabilityCycle reactivation is reading a longitudinal story about their relationship; the arc detector makes that story visible.

9.4 Event Correlation Index and Relationship State Snapshots

Two background services support the temporal layer. The event correlation index in scoring/reference_resolver.py resolves implicit references across narrations — when a user says "the thing last week," the resolver attaches the right earlier event_id so the substrate knows what "the thing" refers to. This allows the pattern engine to build genuine causal chains across sessions rather than treating each narration in isolation.

Relationship state snapshots in scoring/snapshot_tracker.py write periodic state vectors for each user-entity dyad: trust, reciprocity, repair, conflict, consistency, relational trajectory. These snapshots allow the substrate to answer "how was this relationship doing at this point in time" without re-deriving from raw signals on demand. The trajectory comparison between consecutive snapshots is what surfaces the Intensifying, Fading, and Reversal Tier-4 patterns.

9.5 Why Temporal Continuity Is the Premium-Tier Wedge

Static pattern detection is the baseline cognition product. Temporal continuity is the cognition product that compounds in value the longer a user is on the platform. The architecture is designed so that the longer a user engages, the more uniquely valuable their substrate becomes — and the more irreplaceable MetaOp AI becomes as a tool. This is the durable competitive moat: not the LLM (which is swappable commodity infrastructure), but the accumulated, evidence-grounded, temporally-aware structured cognition that no other product can reconstruct without starting over.

PART X

The Compounding Filter Stack

Signal-to-noise is the whole game in a persistent cognition system. MetaOp AI employs five filters between what a user types and what reaches the LLM. None of them is novel individually; the discipline is in stacking them deliberately so each one removes a class of noise the next one cannot. The compounding part is the point: Filter 5 only has to do a little work because Filters 1 through 4 already did most of it.

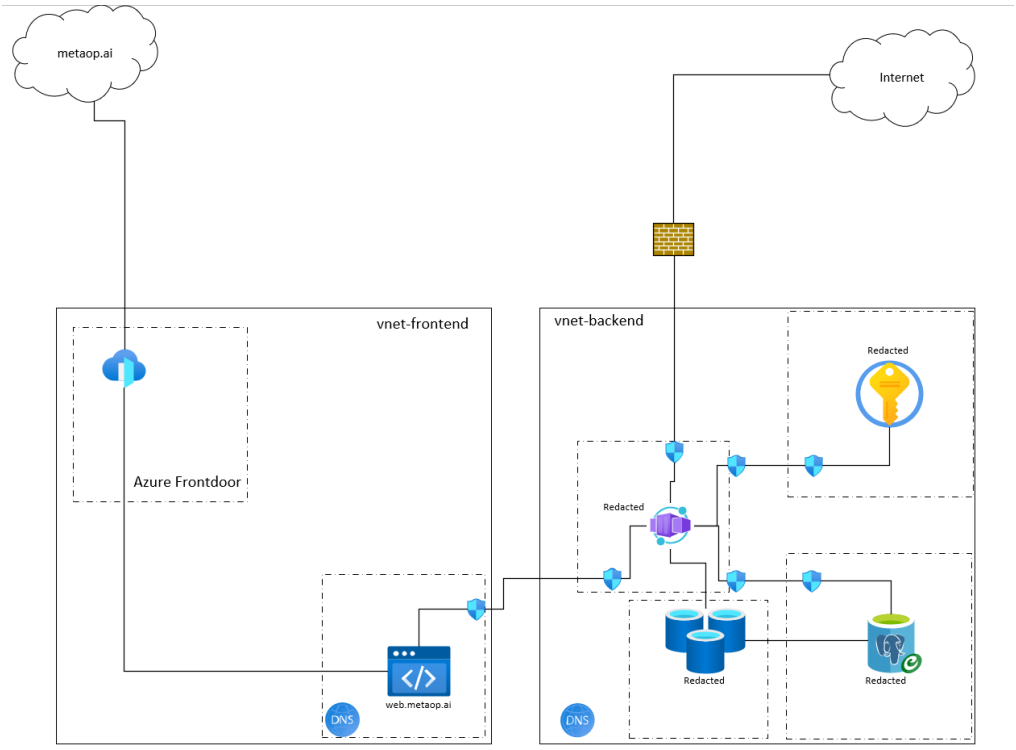

Figure 7 — Five filters stacked between narration and prompt: confidence floor, write-time evidence gate, severity-weighted TTL, read-time ontology guard, and priority truncation. Each strips a different class of noise.

Filter 1: Confidence Floor

When an extractor produces a signal, it also produces a confidence score. If the score is below a configurable floor (default 0.5), the signal never enters the substrate. This is the bluntest filter, but it removes the largest single class of noise: low-confidence LLM extractions that are technically structured but semantically thin. A half-hearted extraction that would otherwise dilute the substrate with noise is stopped at the first gate.

Filter 2: Write-Time Evidence Gate

A pattern only persists when enough signals across a real time window agree. A single utterance — however emotionally charged — does not fire a pattern. This is the difference between a system that detects mood and a system that detects dynamics. It also stops the second-most-common class of noise: a single intense narration triggering a spurious pattern that would then inappropriately influence every subsequent turn until the pattern decays.

Filter 3: TTL with Severity Weighting

Older signals decay faster than recent ones. Critical signals decay slower than minor ones. Decay is severity-weighted, not flat: a withdrawal signal from three weeks ago carries less weight than one from yesterday, unless yesterday's signal is part of a longer pattern — in which case the older one matters again through the parent-pattern linkage. Dormant signals are not deleted; they are deprioritized. If they become relevant again (pattern reactivation), the system restores their weight through the reactivation machinery.

Filter 4: Read-Time Ontology Guard

When the orchestrator retrieves patterns for the current turn, it can only retrieve from the closed vocabulary of patterns that actually exist in the bank. If the system has no patterns for a user's scope, it does not hallucinate categories to fill the silence — it tells the model so explicitly via a sparse-substrate directive. This filter stops the most pernicious failure mode of LLM-driven memory systems: the model inventing patterns that sound plausible because the prompt asked for them. The ontology guard makes silence safe.

Filter 5: Priority Truncation

Even after the first four filters, the orchestrator trims retrieved context to a bounded prompt budget before composing the final payload. Critical patterns first. Recent signals next. Everything else cut. The result is that the prompt reaching the LLM is small, dense, scope-correct, and cited. The LLM is performing rendering, not reasoning over a hundred turns of history. A compact prompt produces more accurate, more coherent, and more stylistically appropriate output than an overloaded one — regardless of the model's theoretical context window capacity.

10.1 The Discipline of Doing Less

Selective intelligence and the compounding filter stack are two expressions of the same underlying principle: the right amount of intelligence applied at the right moment produces better outcomes than maximum intelligence applied uniformly. The filter stack is not a cost-reduction mechanism (though it has that effect); it is a signal-quality mechanism. Every layer is designed to strip a different category of noise so that what finally reaches the model is a high-fidelity representation of what matters most to this user, in this scope, at this moment.

PART XI

Observability, Versioning, and Experimentation

Once the system has three cognition scopes, three policies, and an orchestrator that resolves them per-turn, the ability to evolve any of those things safely requires a corresponding observability substrate. The trace layer, per-layer versioning, and experiment framework are the deployment-safety infrastructure that makes cognition iteration possible — and auditable after the fact.

11.1 The Trace Layer

Each turn writes a structured cognition trace describing what the orchestrator decided, what the retrieval layer fetched and why, what patterns the engine surfaced, what the composer assembled, and what the LLM said. The trace is not for the user; it is for engineers. It enables debugging of AI behavior weeks after the fact without having to re-ask the model what it did — cognition as a governed runtime system, not a black box.

Several of the most important architectural decisions in this paper came directly from things the trace layer caught. Three examples of instructive failures:

Failure Mode	Discovery	Fix
Hallucinated pattern names	Traces showed the LLM citing patterns not in the bank; model was inventing plausible-sounding category names because the prompt left room for it	Added the read-time ontology guard (Filter 4) and a sparse-substrate directive telling the model to acknowledge silence rather than fill it. Hallucination confirmed stopped via traces.
Stream hang from pattern enrichment	Every pattern in a turn was triggering a per-pattern LLM call to generate emotional alternatives, serially, with no outer timeout. Traces showed wait clustering around the enrichment step.	Wrapped enrichment in per-call timeout; moved inexpensive type-check pre-LLM. Latency dropped immediately and was confirmed stable via trace timing distributions.
Re-ask cliff from stale patterns	Users were re-asking questions immediately after AI responded, visibly detecting stale pattern state. Pipeline was running fine — extractors and pattern recomputation simply happened after the stream.	Added settle-version marker emitted over the same SSE stream when post-turn cognition finishes. UI refreshes silently on marker receipt. Re-ask behavior eliminated; confirmed via session analytics.

11.2 Per-Layer Version Constants

Every cognition layer carries a version constant: orchestrator, pattern engine, signal extractor, composer. All four are recorded on every turn's trace. When a behavior change ships, the version constant bumps. Trace consumers can then filter by version when analyzing outcomes — "what is the citation rate when pattern engine is at v2.0 versus v2.1?" has a clean answer. Without per-layer versioning, behavior regressions are invisible until users report them.

11.3 Deterministic Variant Assignment

Users are assigned to experiment variants deterministically: a hash of (user_id, experiment_name) maps to a bucket between 0 and 99, and the bucket determines which variant the user sees. The same user always lands in the same variant for the same experiment. Changing the rollout percentage shifts the population boundary, but every user's membership remains stable up to that boundary. This avoids the most common A/B pitfall — users flipping between variants — which makes experiment data unusable.

11.4 Variant Policies

Experiment variants resolve to CognitionPolicy instances. This means an experiment is "some users get policy A, others get policy B," and the orchestrator already knows how to consume different policies. Experiment context does not need to be threaded through every layer of the system; the experiment's output is a variant policy, and the orchestrator uses it the same way it uses any other policy. The infrastructure is in place even though the experiment registry is currently empty — first experiments are deliberately deferred until real users are in the system, because experiment design without real users is mostly theater.

PART XII

Security, Infrastructure, and Operational Systems

12.1 Authentication and Security Posture

Authentication is dual-mode and environment-gated: AUTH_MODE=local for development (email/password), AUTH_MODE=oauth for production with four providers (Google, Apple, Microsoft, Facebook). Access tokens are short-lived JWTs (15-minute expiry); a 30-day refresh token in an httpOnly cookie rotates them silently on 401. A 401 carrying no token must not clear the session — early-mount queries fire before the store hydrates, a subtle race condition addressed explicitly in the client.

Security Component	File	Mechanism
Rate Limiting	middleware/rate_limit.py	Redis sliding-window; buckets: auth (5/min), stream (30/min), read (300/min), webhook (1000/min IP-keyed), billing_mutation (10/min). Fail-open if Redis is down.
Concurrency	concurrency.py	Per-user simultaneous-stream semaphore (cap 3); prevents cost-amplification and connection-parking attacks
Safety Overlay	prompts/shared_safety.py	Single source of crisis primer + crisis protocol (988 / Text HOME 741741 / 911); imported by every chat surface; overrides all other AI rules
Privacy Guards	retrieval/privacy_zones.py	Retrieval-time privacy zone enforcement; scopes all entity reads by user_id
Consent Audit	consent + consent_log	Legally-required consent accept/decline flow with append-only audit table
SAST	Completed audit	HIGH/MEDIUM/LOW findings closed; CSP report-uri configured

12.2 Azure Infrastructure Architecture

The production deployment targets Microsoft Azure with a PaaS-first, private-networking posture. The architecture favors managed services over self-managed infrastructure to minimize operational burden while maintaining security controls appropriate for a platform handling sensitive personal relationship data.

- Azure Container Apps hosts the FastAPI backend with horizontal scaling and built-in HTTPS termination.
- Azure Database for PostgreSQL Flexible Server provides the durable KRL substrate with point-in-time recovery and private endpoint networking.
- Azure Cache for Redis provides rate-limiting, distributed locking, and response caching with private endpoint networking.
- Azure Key Vault manages all application secrets with Managed Identity authentication — no credentials distributed to containers.

- Azure Monitor and Application Insights collect operational telemetry for performance and error analysis.
- Azure OpenAI Service provides model access in production with data-residency guarantees appropriate for personal data.

12.3 Background Workers and Scheduled Tasks

Component	File	Schedule/Trigger	Purpose
TTL Worker	scheduled/ttl_worker.py	Every 6 hours (in-process asyncio loop)	Pattern TTL expiry + tier-3 compound integrity sweep + 7-day cognition-trace prune
Event Reference Resolver	scoring/reference_resolver.py	Background, post-turn	Links related events via event_links / correlation index
Snapshot Tracker	scoring/snapshot_tracker.py	Post-scoring, per-turn	Writes relationship-state snapshots for trajectory comparison
Profile Summary Lifecycle	insight/profile_summary_lifecycle.py	Quota-gated, on density milestones	Quota-gated background regeneration of user's profile summary
Hero Cache	insight/hero_cache.py / hero_llm.py	Cached; LLM output cached, not live metrics	Hybrid caching: cache stylistic LLM output, never live metric counts

12.4 The Cognition Lock

The cognition tail is serialized per scope-anchor so two concurrent turns cannot interleave scoring. app/services/locks.py provides a CognitionLock abstraction selected by COGNITION_LOCK_BACKEND=memory|redis. The memory backend is an in-process asyncio.Lock — correct on one process. The redis backend is a distributed SET-NX-PX lease with a holder-only Lua release and a bounded acquire-wait, so the guarantee holds fleet-wide across Azure replicas. On Redis outage it fails loud and skips the tail — the reply already landed, and cognition self-heals on the next turn — rather than silently racing. Three scope keys: space, entity, and shared-user.

12.5 Billing, Quota, and Burst

A four-tier subscription model — Free / Entry / Mid / Pro — with unified pricing across web, iOS, and Android. Feature gates are flat: every tier receives the full cognition feature set. Tiers differ by graduated resource caps and token budgets. The premium relationship-cognition experience emerges from the resource caps, not a feature paywall. This is a deliberate positioning decision: the product is not feature-gated; it is depth-gated.

Component	File	Role
Plan Type	billing/plan.py	Stripe price/subscription → tier; comp/unlimited overrides for QA
Enforcement	billing/enforcement.py	enforce_feature (flat), enforce_resource (graduated caps: spaces 1/5/10/15, entities 2/10/20/30), pre-flight token quota wall (402 over_quota)
Usage Tracker	billing/usage_tracker.py	Token-usage accounting; daily caps reset at 06:00 local; per-tier budgets
Subscriptions	billing/subscriptions.py	change_subscription (paid↔paid + downgrade) with archive-keep-N logic for over-cap resources (target_tier + keep_space_ids)
Webhooks	billing/webhooks.py	Stripe: subscription.updated/deleted tier transitions, paid→paid archive, auto-unarchive, drift reconciliation
Burst Pools	billing/plan.py constants + burst.py router	One-off token top-ups. BURST_TOKENS_SMALL = 100,000 and BURST_TOKENS_LARGE = 200,000 live in the service-layer plan registry; burst.py exposes the HTTP surface.

PART XIII

Appendix: Full Component Index

A.1 HTTP Routers (app/routers)

Component	Role
journal / entity_journal / user_journal / growth	The four cognition-journal surfaces (run the orchestrator)
_journal_pipeline_helpers	Shared tail callables, shared USER-scope lock, SSE response wrapper
cross_space / mirror / mirror_moments / analysis	Synthesis, reflection, and analysis surfaces
timeline / notifications / engagement / priority	Read-only views over the substrate
patterns / krl	Pattern-bank + KRL inspection / TTL recompute
spaces / entities	Resource CRUD + assignment (cap-gated)
profile / burst / billing / consent / security / health / dev / entity_chat	Profile/export/estimates, burst pools, billing, consent, security, health, dev helpers, legacy entity narration

A.2 Cognition Services (app/services/cognition)

Component	Role
orchestrator / orchestrator_result	Conductor + typed result; shared by all three scope subclasses
policy / policy_registry / policy_resolver	Policy contract, per-scope defaults, env>user>space>default resolution
pattern_frame	Structured frame line + Phase-2 scope-conditioning for pattern injection
trace / intent_classifier / signal_injection / pattern_injection	Observability + prompt-injection helpers
versions / experiments / citation_delta	Version constants, experiments framework, citation contract
synthesizers/* / synthetic_retrieval	Cold-start synthetic-cognition scaffolding + 4-dimensional retrieval matcher

A.3 Scoring and Detection (app/services/scoring)

Component	Role
pattern_detection / pattern_templates	Tier-1/2 detection + 43-pattern template registry
cross_layer_detection / cross_layer_rules	Tier-3 compound pattern detection
trajectory_detector / drift_detector / timing_detector	Tier-4 second-order patterns; aspect drift; temporal clustering
arc_detection / absence_detection / absence_rules	Relationship arcs + negative-space patterns
self_narrative_drift_detector	USER identity-drift / contradiction feed for self-narrative patterns
cognition_propagation / pattern_subjects	ACL-2 propagation gate + uniform subjects model (normalize_subjects)
pattern_alternatives / pattern_explanation	Alternative reads + Why-flagged explanations per pattern
{reliability,responsiveness,accountability,initiative,influence,overall}	Per-dimension + aggregate relationship dimension scoring algorithms
signal_classifier / epoching / priority / temporal_anchor	Signal taxonomy classification, time-epoching, priority ranking, narrated-anchor parsing
reference_resolver / snapshot_tracker / state_vector_builder	Event-link resolver + relationship-state snapshot writing

A.4 Repositories (app/repositories)

Component	Role
patterns / pattern_history	Pattern-bank write chokepoint (upsert_pattern, TTL, annotate_user_dyad_patterns) + lifecycle log
extracted_records / signal_views / signals	Substrate writes + read-side aggregation views (post signals-table decommission)
cognition_state / cognition_traces	Consistency counters + trace persistence/prune
conversations / journal / growth	Conversation, journal, growth-entry persistence
spaces / entities / profile	Space/entity/profile aggregates
events / event_links / relationship_state_snapshots	Event sequencing, cross-event links, dyad snapshots
synthetic_blob / engagement / session_visits	Synthetic blob, engagement events, visit telemetry

A.5 Data Models (app/models)

Component	Role
user / space / entity / space_entity	Account, context, person, and assignment junction
conversation / growth_entry	Sessions/threads (+ synthetic blob JSONB) and growth entries
extracted_record / pattern / pattern_history	Substrate row, pattern bank, lifecycle log
cognition_state / cognition_trace	Consistency counters + per-turn trace (7-day TTL)
event / event_link / relationship_state_snapshot	Events, cross-event links, dyad state vectors
billing / consent_log / oauth_token / session / session_visit	Billing, consent audit, auth/token/session/visit
signal / signal_digest / mirror_moment / risk / engagement_event / context_search	Legacy/auxiliary records retained for compatibility and auxiliary surfaces

A.6 Frontend (frontend/src)

Component	Role
App.tsx / main.tsx	App shell, routing, providers
screens/	All screen components: journals, spaces, entities, activity, settings, billing, onboarding
components/QuotaModal.tsx + store/quotaStore.ts	Token-limit UX: reactive off SSE quota_error frame / 402
components/SignalSparkline.tsx	Per-signal trend sparkline (NULL-truthiness-safe)
components/insight/*	Pattern cards (history, emotional alternatives, awareness/action distinction, crisis refs), entity intel brief
components/{ConsentGate,OAuthButtons,Tutorial,LeftNav,UsageWarningBanner}	Consent flow, OAuth entry, onboarding tutorial, navigation, usage banners
api/client.ts	API layer: toCamel/toSnake case conversion, refresh-on-401 token rotation, SSE journal stream consumption

Component	Role
hooks/	Data hooks including useStartSubscriptionCheckout, useChangeSubscription, and journal-stream consumers
constants/ + types.ts	Shared constants and TypeScript type surface mirroring the API

A.7 QA Harness (qa/)

A standalone QA harness validates the running stack end-to-end via the real API. run_all.sh executes fixtures (seed accounts/spaces/entities), checks (migration head, SQL health invariants, route smoke, log scanner, cognition-state inspection, extractor provenance), and 26 numbered scenarios covering every cognition scope, the backgrounded-cognition lifecycle (flag on/off), bounded-strict refresh, synthetic-blob cold start, the over-cap 402 wall, sparse-substrate guards, crisis routing, cross-space/mirror/predictive/timeline/export/estimators, and billing. The harness is the gate every change must pass before a known-good snapshot is banked.

Technical Takeaway

The primary engineering contribution of MetaOp AI is not a larger prompt or a more sophisticated model. It is a disciplined memory architecture: structured storage, scoped retrieval, evidence lineage, bounded prompts, TTL on patterns, and observability strong enough to debug cognition as a runtime system.

The system was designed to solve a problem that most AI products do not prioritize: the regenerative context inflation that eventually destroys continuity in every general-purpose AI assistant. The solution — separating extraction from generation, treating provenance as a first-class invariant, and building a governed cognitive substrate rather than accumulating transcript — is not dramatically novel. The discipline of executing it consistently across every layer and every edge case is.

> **The Governing Positioning**
>
> *MetaOp AI confirms patterns, never conclusions. It is signal intelligence, not advice. This shapes every prompt, every surface, the legal posture, and a hard crisis-protocol override. The architecture exists to make that positioning operationally honest at scale — not just in principle.*

The substrate is the moat. The model is rented infrastructure at the rendering edge. Every architectural decision in this document serves to protect the substrate's epistemic integrity, make it more valuable over time, and ensure that the intelligence the user builds through sustained engagement with the product remains durable, auditable, and truly theirs.

metaop.ai · founder@metaop.ai · Version 7.0 · May 2026

PART XIV

v7 Architecture Delta: Corrections and Substantive Expansions

This v7 section folds the delta audit into the whitepaper without reducing the original document. It corrects code-facing factual drift, expands the missing read-side cognition surfaces, and makes the founding invariant explicit: persistent cognition is not extended chat history. It is a structurally separate substrate, written by extraction rather than generation, retrieved by deterministic addressing rather than embedding similarity, governed by its own physics of decay, contradiction, and propagation. The LLM rents access to the substrate at the rendering edge; the substrate is the cognition.

Highest-Level Invariant	The KRL must NEVER recursively ingest itself. The product learns from raw user narration and durable extracted evidence, not from the model recursively summarizing its own generated interpretations.

14.1 Factual Corrections Folded into v7

The following corrections are now reflected in the body tables and narrative. They are included here as an explicit maintenance ledger so that a code reviewer can see exactly how v7 reconciles the document with the implementation.

Item	v7 Treatment
Pattern registry count	The pattern registry is stated as 43 registered detectors, not 44. The underlying split is 29 template entries plus 14 non-template definitions.
Extractor count	The extraction pipeline is now described as six extractors, with space_filter_extractor documented as a production routing component.
Lifecycle states	lifecycle_state is binary on disk: active or dormant. changing, repeating, and weakening are computed surface labels derived from evidence count, recency, suppression, override, reactivation, and drift fields.
USER journal route	The route is corrected to POST /api/users/me/journal.
SPACE orchestrator class	SPACE uses the base JournalOrchestrator directly with SPACE_JOURNAL_POLICY. SpaceJournalOrchestrator is a conceptual label, not a separate literal subclass.
Contradiction field	The extracted_records contradiction pointer is described as contradicted_by_signals.
CognitionPolicy	The policy table now acknowledges the full 25+ flag governance surface, including relationship propagation, analytic voice, correction routing, and background cognition.

Burst SKU registry	Burst token constants are described as service-layer constants in billing/plan.py; burst.py is the HTTP router surface.
Unlimited plan	The billing section now treats unlimited as an engineer/QA override tier rather than a public pricing tier.

14.2 Four Structural Defenses Against Regenerative Learning

Regenerative context inflation is the founding problem MetaOp AI was built to avoid. The operational answer is not a prompt instruction; it is four structural defenses that make recursive self-ingestion difficult by design.

Defense	Enforcement Mechanism	Why It Matters
Raw-user-input extraction only	Extractor call sites receive the user narration payload, resolved mentions, and scope metadata. They do not receive the generated assistant response as write input.	The system cannot learn durable facts from its own prose if the prose is not available to the write path.
Provenance gate at the upsert chokepoint	extracted_records accepts durable writes only when provenance resolves to user_narration. Non-matching provenance is rejected before persistence.	The write boundary enforces the invariant centrally instead of relying on every caller to behave correctly.
Synthetic cognition is physically separate	Synthetic cognition lives in conversation-scoped JSONB with synthetic provenance and a conversation lifetime, not in durable KRL rows.	Cold-start scaffolding can help the LLM reason without contaminating the long-lived substrate.
Dual-tagged prompt authority	Runtime prompt blocks distinguish DURABLE from SYNTHETIC context and explicitly assign lower authority to synthetic content.	The LLM is told which memory is evidence and which memory is tentative scaffolding at the rendering edge.

These defenses convert the founding thesis into a runtime invariant. Durable cognition is written by extraction, not generation. The LLM may render, question, explain, or summarize the substrate, but it does not become the substrate.

14.3 Cross-Space Whiteboard: The Deliberate Cross-Boundary Surface

The cross-space whiteboard is the one surface designed to intentionally cross space boundaries. Ordinary journal turns remain scope-correct; cross-space analysis exists for questions such as, "What is happening across work and family right now?" It is not a generic global search. It is a controlled multi-source cognition read that composes evidence across separate contexts while keeping the provenance of each category visible.

Retrieval Source	Purpose
Per-@entity patterns	Pulls active entity-specific dynamics for people named or resolved in the turn.

Per-@entity relationship-state snapshots	Adds dyadic vectors such as trust, reciprocity, repair, conflict, and trajectory.
Per-@entity event histories	Reconstructs what happened around each entity across time.
Per-@entity mention briefs	Provides compact continuity for entities that appear in the user request but may not have dense pattern banks.
Per-#space patterns	Loads the active environmental dynamics for each referenced space.
USER-scope fallback	Preserves the user-wide self-reflection layer when the request is not cleanly owned by one entity or space.
User-wide cross-space signal index	Detects similarities, divergences, recurrence, and stress transfer across spaces.

The stacked-category directive, CS-H, is the product discipline that prevents cross-space answers from collapsing into vague summaries. When the user asks for multiple categories in one turn, the model must enumerate each category, provide explicit fallbacks when no stable pattern exists, and avoid silently substituting one category for another. Forward-tagged events with entity_ids arrays preserve actor linkage across event histories, and the ChatMessage attribute-access contract remains load-bearing for the SSE streaming path.

14.4 KRL Explorer, Graph Explorer, KRL Formation, and Evidence Chain

The flagship read-side cognition surfaces turn the substrate into an analytic terminal rather than a hidden memory backend. They answer four different questions: What exists? How is it connected? How was it formed? What evidence supports it?

Surface	What It Shows	Trust Mechanism
KRL Explorer	A list of active records and patterns with scope, confidence, lifecycle, layer, subcontext, and subject chips.	Makes the substrate inspectable instead of invisible.
Graph Explorer	A concentric graph of users, entities, spaces, dyads, and patterns. Each node carries seven deterministic counters: signal_count, event_count, pattern_count, compound_count, multipattern_count, recurring_count, trend_count.	Counters come from deterministic row aggregation in node_counts.py, not heuristic name matching.
KRL Formation	A 4x5 matrix view showing which substrate records compounded into a selected pattern.	Shows the reader the exact subject-layer cells that built the inference.
Evidence Chain	A chronological audit trail from parent_pattern_ids to contributing extracted_records to source_message_id and the original journal sentence with timestamp.	Lets the user ask, "show me the receipts," and receive evidence rather than persuasion.

This distinction matters technically and commercially. A visualization that merely looks connected is decoration; a graph whose node counts are produced by deterministic aggregation is an audit surface. The earlier heuristic approach could miscount by matching names in text. The locked design replaces each node meta object wholesale with exact row-count tuples, which makes Graph Explorer a signal-intelligence terminal rather than a pretty diagram.

14.5 Compound Pattern Construction Made Concrete

Compound patterns are not labels pasted onto a relationship. They are joins across substrate evidence, subject scopes, layers, and pattern tiers. A concrete Bob-relational-breakdown chain looks like this: ENTITY(Bob) signal cluster (dismissal, reciprocity failure, withdrawal) becomes a Tier-2 pattern; that joins with USER signal pattern evidence showing emotional impact; that joins with ENTITY(Bob) event sequences showing recurring conflict; that joins with ENTITY(Bob) meta-context drift where Bob-as-mentor weight falls while Bob-as-rival weight rises; that may be corroborated by SPACE meta-context such as rising team gossip or political pressure. The parent_pattern_ids JSONB records the lineage so the system can walk back from the surfaced compound to the actual evidence.

Why this matters	Without lineage, “compound pattern” reads like a buzzword. With parent_pattern_ids and Evidence Chain traversal, the compound is auditable: the reader can see the exact component patterns and source records that formed the arc.

14.6 Workbench / Radar API: Read-Side Cognition Contract

The Workbench and Radar API are the read-side contract behind the analytic surfaces. They are how the product serves cognition without rerunning full journal orchestration. The read side has its own discipline: deterministic aggregation, explicit scope selection, surface-specific filters, correction pathways, and sparse-substrate honesty.

Route / Function Family	Role
R1 summary	Returns the current high-level substrate summary for the selected scope.
R2 pattern detail	Loads one pattern with evidence, confidence, lifecycle, and formation metadata.
R3 cross-space matrix	Compares spaces, entities, and user-level signals across contexts.
R4 per-scope dashboards	Feeds USER, ENTITY, SPACE, and RELATIONSHIP_PAIR dashboards with scope-correct analytics.
R5 radar graph	Returns graph nodes and edges, including deterministic KRL counters per node.
R6 query terminal	Allows structured analytic interrogation of the substrate without treating the LLM as the source of truth.
R7 correction write	Lets the user correct the model, preserving the principle that user correction outranks inferred cognition.

14.7 Analytic Voice and Four Sub-Intents

The analytic voice is the difference between journaling support and operator-grade analysis. The same substrate can support warmth in the Growth Journal, dyadic restraint in an Entity Journal, and structured reasoning in an analytic turn. The intent classifier identifies when the user is not merely narrating but asking the system to analyze, define, correct, or brainstorm.

Sub-Intent	Behavior
Dictionary mode	Defines terms, pattern names, and substrate concepts in plain language without over-analyzing the user.
Correction mode	Accepts user corrections as authoritative signals and routes them into the appropriate correction flow.
Brainstorm mode	Explores possibilities while preserving uncertainty and avoiding invented substrate claims.
Default analytic	Uses evidence, confidence, alternative explanations, contradictions, and falsification conditions to reason over the substrate.

Analytic voice is also a safety control. It encourages the model to name interaction patterns, show evidence, use confidence, track contradictions, let the user correct the model, and avoid fabricating when an entity has sparse data. It explicitly avoids diagnosing people. The product names the observed interaction pattern, not the person.

14.8 Scope-Identity Prompt Blocks

Scope identity is enforced at the prompt boundary through SCOPE_IDENTITY_USER, SCOPE_IDENTITY_ENTITY, and SCOPE_IDENTITY_SPACE blocks dispatched by the orchestrator scope name. This mechanism is load-bearing because it lets the same infrastructure preserve three different products.

Prompt Block	Operational Posture
SCOPE_IDENTITY_USER	Warm, grounded, reflective. Prioritizes self-awareness, growth, identity drift, and user correction.
SCOPE_IDENTITY_ENTITY	Dyadic, observational, and restrained. Focuses on what is happening between user and entity; avoids motive mind-reading and diagnosis.
SCOPE_IDENTITY_SPACE	Systemic, strategic, and structured. Focuses on group dynamics, organizational drift, actor interactions, and environmental patterns.

14.9 ACL-2 Scored Propagation and Relationship Cognition

Relationship cognition requires careful propagation. A signal about one entity may condition how a dyad is read, but it should not automatically become a fact about another entity or another space. ACL-2 scored propagation solves this by attaching policy-controlled scoring to cross-subject propagation. The five relationship propagation policy flags - relationship_propagation_enabled, relationship_propagation_mode, relationship_propagation_floor, relationship_propagation_decay, and relationship_propagation_max_hops - act as runtime governors.

The key idea is controlled influence, not uncontrolled copying. Propagated cognition can condition a read surface when its score clears the floor, decays per hop, and remains within the max-hop limit. It does not rewrite the source subject, and it does not promote a weak inference into durable evidence.

14.10 High-Stakes Relationship Mode

High-stakes relationship mode is a safety boundary for sensitive relational claims. For any slot marked high_stakes=True, the extractor requires a confidence of at least 0.95, perspective=user_asserted, and a source_quote containing the user's verbatim direct claim. If any condition fails, the system silently skips the slot.

Trauma-aware silent skip	The system does not fire a clarification prompt for ambiguous high-stakes relationship claims. Asking a user to confirm a dangerous or abusive situation can itself create harm. Silence is the safety behavior unless the user directly asserts the claim with sufficient evidence.

14.11 The Expanded Compounding Filter Stack

The original five-filter framing was directionally correct but too compressed. In implementation, there are multiple gates across both write and read paths before a pattern reaches a prompt. v7 treats this as a 7+ filter stack rather than a literal five-filter pipeline.

Filter	Gate Point	Purpose
Confidence floor	Write path - extracted_records upsert	Rejects low-confidence records before they enter durable substrate.
Contradiction delta	Write path - extracted_records upsert	Prevents weak new claims from silently overwriting stronger prior claims.
Provenance check	Write path - upsert_record	Rejects non-user_narration durable writes.
Evidence-signal-count floor	Read path - active-pattern loader	Excludes patterns without sufficient evidence density.
Lifecycle dormant exclusion	Read path - active-pattern loader	Keeps dormant patterns out of ordinary prompt injection.
suppressed_until / is_overridden	Read path - active-pattern loader	Honors suppression windows and user override authority.
awareness_tier gate	Read path - per-surface policy	Controls how much pattern awareness a surface can reveal.
Surface confidence floor	Read path - per-surface policy	Prevents weak patterns from being rendered as confident cognition.
Sparse-substrate directive	Read path - pattern injection	Forces the LLM to acknowledge absence rather than invent ontology.
Humanization strip-before-prompt	Read path - pattern humanizer	Converts internal detector names into user-safe language.

Pattern TTL by severity	Write/background path - TTL worker	Expires or weakens patterns according to severity and time.
Hallucination blocklist	Write path - upsert_pattern	Rejects known-fabricated pattern names at the structural boundary.
JSON safety guard	Write path - upsert_pattern	Protects pattern_data integrity before persistence.
Priority truncation at token cap	Read path - continuity block	Drops lower-value context before exceeding the prompt budget.

14.12 Three Cognition Levels as Unit-Economic Discipline

The three cognition levels are not just UX tuning. They are a cost-control architecture. If every turn ran as a deep cognitive pass, token cost would scale with the noisiest possible interpretation of user behavior. Instead, Fast Pass handles low-signal turns with minimal context; Continuity Pass handles recurring-topic turns with moderate retrieval; Deep Cognitive Pass handles explicit analytical asks with full evidence and pattern lineage.

Level	Name	Economic Function	Classifier Behavior
1	FAST PASS	Serves casual, low-signal, or acknowledgment turns without loading maximum substrate context.	Heuristic-first classification.
2	CONTINUITY PASS	Loads relevant continuity when the user references a recurring entity, event, or emotionally weighted context.	Heuristic layer plus optional tie-breaker for ambiguous mid-band turns.
3	DEEP COGNITIVE PASS	Spends tokens when the user explicitly asks for analysis, evidence, pattern lineage, or state reconstruction.	Full analytic routing; result threads through OrchestratorResult as cognition_level.

Representative function signature: classify_cognition_level(message: str, *, enable_tiebreaker: bool = True) -> CognitionLevel. The fast regex layer keeps ordinary turns cheap, while the LLM tie-breaker is reserved for ambiguity rather than used as the default classifier.

14.13 Synthetic Cognition Asymmetric Authority

Synthetic cognition works only because three deprioritization layers operate together: order, framing, and volume. Durable KRL appears first and carries authoritative provenance. Synthetic content appears later under an explicit SYNTHETIC SUBCONTEXT frame that describes it as low-authority, tentative, and unable to override durable evidence. Finally, selective retrieval caps synthetic content aggressively, keeping it small compared with durable context.

Property	Durable KRL	Synthetic Cognition
Prompt order	First	Last

Authority label	Evidence / durable / user_narration	Tentative / low-authority / cannot override durable
Lifetime	Cross-session	Conversation-scoped
Write path	Extractor -> upsert_record -> KRL	Synthesizer -> conversation JSONB
User-facing status	Inspectable in KRL and evidence surfaces	Invisible scaffolding; not surfaced as truth

Without order, framing, and volume control, synthetic cognition could become the quiet wedge that contaminates durable memory. With all three in place, it becomes a cold-start bridge rather than a competing source of truth.

14.14 Token Trimmer Priority and Strict Consistency

The continuity block uses explicit priority ordering when token pressure forces trimming. Conceptually, the system drops the least authoritative or least timely material before it drops recent durable evidence.

Priority	Trim Category	Reason
1	Synthetic blob entries	Lowest authority and easiest to regenerate.
2	Weakening or suppressed patterns	Potentially stale or intentionally down-ranked.
3	Low-confidence meta-context	Interpretive material below the stronger evidence band.
4	Non-recent events	Permanent but less useful to the immediate turn.
5	Signals outside active decay window	Behavioral observations whose time weight has decayed.
6	Recent high-confidence records and active pattern evidence	Preserved as long as possible because it carries the core continuity.

Strict consistency mode exists for explicit latest-state analytics. Ordinary journaling can tolerate eventual or bounded-strict consistency because the user values response speed. When the user asks a question such as, "What is the current state after what I just told you?" strict mode can block until the substrate settles so the answer does not race the cognition tail.

14.15 Azure Locked Beta Architecture Refresh

The locked beta architecture remains Azure PaaS oriented: React/Vite frontend, FastAPI backend, PostgreSQL Flexible Server with JSONB substrate tables, Azure Cache for Redis for rate limits and distributed locks, Key Vault with Managed Identity for secrets, and Azure OpenAI for production model access. The current private beta posture favors operational simplicity over premature enterprise complexity: deploy the backend and frontend, validate Key Vault integration, keep data services private where practical, instrument usage and token cost, and preserve the ability to scale down idle runtime resources before revenue.

The Azure posture should be read as a beta-to-production bridge rather than the final enterprise reference architecture. The immediate requirement is stable private beta, product QA, cost visibility, and launch readiness. The later requirement is deeper isolation: private endpoints, private DNS, WAF decisions, environment separation, and production hardening after real traffic validates the deployment profile.

14.16 Versioning, Experiments, Tavily Scope, and Locks

Per-layer version constants and the experiments framework are shipped as infrastructure even where experiment registries currently resolve to identity defaults. This is intentional: the system can introduce prompt, retrieval, or policy variants without changing the durable substrate. Tavily enrichment remains deliberately scoped to Growth Journal inline enrichment and Intel-style reports unless future call-site review expands the policy. The cognition lock uses memory locks for single-process local runs and Redis SET-NX-PX leases with holder-only Lua release for distributed production safety. Holder-only release matters because it prevents one process from accidentally releasing a lease that another process acquired after the original lease expired.

14.17 Safety and Product Principle: Pattern, Evidence, Correction

The product principle should remain explicit across investor, technical, and product materials: do not diagnose the person; name the interaction pattern; show the evidence; use confidence; track contradictions; let the user correct the model; do not fabricate if the entity has sparse data. This is the difference between signal intelligence and clinical, therapeutic, or defamatory overreach. The engine observes behavior and interaction structure. It does not assign identity verdicts, clinical labels, or hidden motives to people.

14.18 v7 Technical Takeaway

v7 strengthens the core claim: MetaOp AI is not a longer context window and not a prettier journal wrapper. It is a governed cognition substrate with explicit write boundaries, deterministic read surfaces, auditable evidence chains, selective intelligence economics, and safety constraints that keep the system honest. The moat is not that the model can speak well. The moat is that the user’s structured cognition becomes durable, inspectable, correctable, and more valuable over time.

— End of Document —