

Borrowable Skills as Lean Un-Ganglia Subgraphs: trainable, composable, bounded-loop standard operating procedures in a co-located, owner-held, indelible knowledge multi-graph

Author: Nelu Lazar **Affiliation:** Nehloo Interactive LLC · Nehloo Foundation, Inc. **ORCID:** 0009-0008-5548-4109 **Status:** Preprint · contact email: nelu.lazar@nehloo.org **Companion paper:** *The Un-Brain* — the declarative memory substrate this work builds on (10.5281/zenodo.20843387). **Artifacts:** Graphnosis App skill layer (github.com/nehloo-interactive/graphnosis-app, `apps/desktop-sidecar`); SDK @nehloo/graphnosis (Apache-2.0, github.com/nehloo/Graphnosis); public trained-skill pack github.com/nehloo/graphnosis-trained-skills — a signed `.gsk` **curated engineering subset**; the full 76/108-skill corpus measured in this paper is not published; and `route-b-artifacts/` in that pack repo — the reproducibility bundle carrying every reproducible claim. **DOI:** 10.5281/zenodo.21205599 · continues 10.5281/zenodo.20843387.

Abstract

Agentic systems encode know-how as prompt templates, tool catalogs, or external workflow-engine graphs — artifacts that live *apart* from the agent’s memory and decay the moment their assumed facts change. We present the **Graphnosis Skill Subgraph**, a procedural layer in which a trained **Skill** — a standard operating procedure (SOP) with eight goal categories (Trigger, Prerequisites, Requires, Produces, Success, Out of scope, On failure, On completion) and a numbered step body — is compiled into a **typed-edge subgraph stored in the same owner-held, encrypted, indelible multi-graph that holds declarative memory** (the substrate of our companion paper, *The Un-Brain*). The procedure *is* the subgraph: there is no separate skill data type. Five evidence-tagged edge classes — `skill:seq`, `skill:loop`, `skill:branch`, `skill:ctx`, `skill:calls` — carry control flow as graph structure over generic nodes, and skill nodes inherit the substrate’s encryption key, operation log, and snapshot versioning unchanged.

What is new is not any single mechanism but *where* the mechanisms sit and how they compose. A trained skill and the private memory it grounds in live together in one **engram** — a per-domain region of the owner’s single indelible **cortex** that co-locates a family of skills with the facts they draw on. Each engram *is* a **Graphnosis agent** — what we call an **Agempus** (an **agent** with a **hippocampus**), the engram its host — a domain agent whose procedures, dispatch index, and recall bindings are co-located with its memory rather than scattered across a prompt file, a workflow engine, and a separate vector store. Because these agents share that one cortex, any one can resolve and call a sibling’s skill *directly* — `@skill: name(args) -> $capture`, composing procedures like functions with argument binding and return capture, depth-capped at three — instead of spawning a fresh sub-agent that must re-warm context. We measure this directly (§10.7): a full sub-agent cold spawn costs $\approx 21.3k$ tokens against a ≈ 475 -token in-cortex walk-and-recall for the borrowed skill — a $\approx 39\text{--}45\times$ **measured separation (two datapoints)** on this harness: a **live single run** (≈ 475 vs $\approx 21.3k$, $\approx 45\times$) corroborated by a **reproducible sandbox** re-measurement (637 vs $\approx 25.0k$ over $N=5$, $39.3\times$); the separately-quoted $\sim 30\times$ is against the ~ 700 -token *modeled* typical borrow, not a measured datapoint — and the architectural claim is the *constant warm-up avoided*, $O(1)$ on both sides under lazy dispatch. The everyday shape of the difference: ask *how do I turn on the sink* and spawning a sub-agent hands you the whole kitchen — the fridge and its contents, the stove and oven, the drawers and cabinets, the lazy-Susan of pans — a whole worker stood up and onboarded when one skill was wanted; a cross-skill call just turns the tap, passing the arguments in and the answer out. The control flow is **Un-Ganglia**: as the basal ganglia selects and sequences a learned procedure, a skill is selected and walked — but bounded by construction, since a `skill:loop` edge may carry a finite cap (`@loop: N max=M`) and a depth-3 recursion cap bounds composition, a *capped* loop cannot run away and an executor honoring the caps terminates — a bound the shipped reference walker now enforces itself: it re-executes loop bodies with the cap enforced in code (the authored `max=M`, else a default cap of three for an uncapped `@loop` — the “executor decides” clause made deterministic — with a fixed-point early exit), under a model-free regression suite (§5, §11); parallel-sibling dispatch remains the one plan feature the walker defers to external executors. Each step

may carry a `@needs` annotation that routes it to the cheapest model whose capabilities cover it under a privacy→strategy→cost precedence (most of the original 76-skill corpus carried none, but a corpus-wide retrofit later tagged 100 of 108 skills — the mechanism behind the E2 routing figures — with router-side inference reserved as a fallback for future untagged steps, §9, §10.7); a generic SOP binds to private facts only at walk time through recall recipes — so, unlike retrieval-augmented generation, which injects fetched text into a prompt, the procedure and the memory it draws on are co-located and owned, and the skill is *walked*, not merely retrieved; runs resume across sessions from encrypted state; and a vitality model drives a governed retrain loop instead of letting procedures silently rot. **The contribution is a synthesis, not any single first:** each ingredient — SOPs, prompt-skill libraries, workflow DAGs, model-cascade routing, retrieval-augmented and procedural-memory stores — exists elsewhere; the novelty is the conjunction held in one local-first indelible memory multi-graph.

Following the determinism-first discipline of the substrate paper, we state and, where they hold, formalize five by-construction properties: **deterministic plan compilation** (the execution plan is a pure read of immutable node order, golden-test-pinned); **bounded loop execution** for *capped* loops composed with the depth-3 recursion bound (a property of the plan *contract* that the shipped walker now also enforces at runtime — loop bodies re-execute with the cap enforced in code, and an uncapped `@loop` is bounded by the walker’s default cap of three, §5/§11; `parallel[]` dispatch remains contract-only); **structure-preserving training** (a token-extraction validator reverts any LLM rewrite that drops a structural token — a DSL token, `@needs`, `only_engrams`, or a goal header — so a known structural token can never be silently *omitted*, a set-presence guarantee, not per-step placement); **cross-skill-edge stability under retrain** (in-place retraining reuses the source id, so the call graph stays resolvable); and the **empty-engram train invariant** (no *automatically-recalled* personal node content enters a skill body at training time — personal grounding occurs only at walk time through recall bindings — so a trained skill is by construction a **portable, source-grounded** artifact: exportable as a signed `.gsk` pack that is *designed to bind a recipient’s* own engrams at walk time). This cross-owner **portability** follows by construction from the empty-engram invariant, but we have not yet evaluated an independent owner/recipient importing and running a pack, so we scope it as future work (§13). The paper’s *measured* result is **borrowability in the intra-cortex sense** established above — one Agempus calling a sibling’s single skill instead of spawning the whole sibling agent. These are guarantees enforced by construction, not deep results; we formalize them because a procedural substrate should be held to them.

As a case study we document **Claude Code reading the Graphnosis repository and the founder’s plans and sessions, identifying the need for Skills, and training a 76-skill corpus** — spanning code/engineering, business/go-to-market, and meta/agent families, later extended to a 108-skill corpus across twelve families — eleven domains plus the chat agent’s own — as a **representational domain-compatibility** probe (compatibility under the same training method — **not** measured cross-user transfer) — that the system then used to continue building itself, with a counterexample in which skills that were dispatched but never *walked* produced no benefit. Driving the real router over one annotated skill — an engine demonstration, not a task-performance result (§10.3) — per-step model routing saves **99.5%** (measurement-time catalog) against an all-Sonnet-4.6 baseline when local models are available (eight of nine steps run free and local under the *measurement-time* catalog; all nine under the shipped catalog, §10.3) — the reproducible corpus-wide figure being E2’s ~99.6% adaptive / ~96.6% cheapest-qualifying cloud-only / 73.3% coverage-max over 105 skills (§10.7) — and **73.3%** in a premium coverage-max cloud-only configuration — that 73.3% is the complement ($1 - 0.2667$) of the Haiku/Sonnet price ratio on the pinned catalog, while the cheapest-qualifying rule reaches ~96.6% cloud-only (corpus-wide cost accounting, §10.7), all cost-only with quality retention measured separately (E6: local quality holds on mechanical steps and collapses on the reasoning/writing steps that dominate the corpus, a 51%/11% capability mix — §10.7); lazy dispatch loads a **computed ~8%** of the full library in the modeled one-to-two-matched-skill session (48,780 tokens to inline the 76-skill measured library vs ~4,100 for a typical session) rather than inlining every skill. Governed retraining measurably restores freshness — per-skill vitality rising 92→98 and 94→99 on a 0–100 scale as superseded nodes are collapsed (a live-cortex re-compile record, §10.5; the E3 read-only sweep corroborates the scoring *mechanism* with a named penalty decomposition, not the up-move itself, §10.7).

1. Introduction

A loop is a mechanism; the knowledge inside the loop is the thing that lasts. The agentic-systems literature has converged on the loop — plan, act, observe, repeat — as the unit of autonomous work, and tooling has followed: prompt files, skill libraries, and workflow engines that script what an agent should do. But the *content* of those loops — the hard-won procedure for shipping a release, triaging a bug, preparing an investor update — is almost always stored as a frozen artifact: a paragraph of instructions, a YAML graph in a workflow engine, a markdown file checked into a repo. Such an artifact has two silent failure modes that mirror the ones the companion paper identifies for declarative memory. The first is **decay by assumption**: a procedure encodes facts (“recall from the **investors** engram”, “the smoke test lives at this path”) that drift out from under it, and nothing notices. The second is **separation from memory**: the procedure lives in one store and the facts it depends on live in another, so the procedure can neither be versioned with them, encrypted with them, nor checked against them.

The thesis we ask the reader to hold onto: a procedure is most durable, and most useful, when it is stored *as a typed-edge subgraph in the same owner-held, indelible graph that holds the agent’s memory* — not as a paragraph, and not as a graph in a separate workflow engine. Co-location is not a convenience; it is what makes a Skill versioned, encrypted, recall-bound, and vitality-tracked for free, because it inherits every one of those properties from the substrate rather than reimplementing them. A Skill is therefore a first-class graph object: a title hub, eight typed goal nodes, and an ordered body of step nodes, joined by five evidence-tagged control-flow classes that encode sequence, bounded loops, conditional branches, context bindings, and sub-skill calls, plus the `skill:goal` attachment binding the goal nodes to the hub. The procedure is not *described by* the subgraph; it *is* the subgraph. And because co-location keeps every *automatically-recalled* private fact at walk time rather than baking it into the procedure, the same move that makes a Skill durable also makes its procedure **portable** by construction: the SOP that ships is grounded in its source, not its owner, so *in principle* it can be exported to another owner’s cortex and run against *their* memory. This cross-owner **portability** follows by construction from the empty-engram invariant — we derive it, rather than demonstrate it with a third-party import — and scope it as future work (§13). But the property this paper *measures* is nearer to home — **borrowability**: one Graphnosis agent borrowing a sibling’s single skill within the same owner’s cortex, rather than spawning the whole sibling agent. The human picture is familiar: borrowing a skill is asking the colleague who already knows — not absorbing their whole expertise or briefing them on everything, but asking the one thing they are expert at, handing over the specifics, and getting the answer back. Absorbing the *whole* colleague — spawning and fully onboarding a fresh AI agent (or sub-agent) for the task — is how agents are built and trained today; a borrowed skill is the leaner alternative, since the colleague here is itself a Graphnosis agent whose one relevant skill can be called without standing up the entire agent. §10.6 puts a number on the difference between Graphnosis agents: rather than standing up a fresh sub-agent for the same sub-task, invoking a sibling’s skill in-cortex avoids the entire spawn warm-up — we **measure** a full sub-agent cold spawn at $\approx 21.3k$ tokens against a ≈ 475 -token in-cortex walk-and-recall for the borrowed skill, a $\approx 39\text{--}45\times$ **measured** separation on this harness (two datapoints — a live single run corroborated by a reproducible N=5 sandbox; $\sim 30\times$ is against the ~ 700 -token modeled typical borrow, §10.7). We report a narrow accounting of this tradeoff, not a law; the architectural claim is the *warm-up avoided*, $O(1)$ on both sides under lazy dispatch (Proposition 5).

This is deliberately **not a separate-store workflow engine** — fairly described, it is a workflow *substrate embedded in the memory graph* — and not a prompt-template library. A separate-store workflow engine owns its own graph format, its own runtime, and its own store, disjoint from the agent’s knowledge; a prompt library is prose that the model must re-read in full every time. The Skill Subgraph is the inverse: the control-flow graph is expressed in the *same* generic node-and-typed-edge primitives the memory substrate already provides (the five `skill:*` classes ride in the optional **evidence** string of ordinary directed edges; there is no new graph engine), it is walked by reading immutable node order rather than chasing edges, and it is loaded *lazily* — an agent reads a one-line index of the library and hydrates only the one or two skills a context actually matches. Retrieval, routing, and composition are operations over an owner-held procedural state, not a bolted-on subsystem.

A single principle organizes the design, inherited from the substrate: **determinism first**. Plan compilation, loop-termination bounds, structure preservation, and in-place versioning are computed deterministically,

with no model in the loop; the optional LLM rewrite that polishes a skill’s prose is a detachable, structure-validated overlay that can never drop an executable token. Which model *runs* a given step, and how much that costs, is governed by a separate, read-only routing pass; the contribution is the deterministic procedural substrate beneath it.

1.1 Contributions

Borrowability — *the headline the rest serve*. The property named in the title is **intra-cortex borrowability**: because sibling Agempi share one owner-held cortex, any one can borrow *exactly the one skill it needs* from a sibling through an `@skill`: call — resolved at training time through the encrypted cross-engram side-table at **zero model-token cost** — instead of spawning and warming up the whole sibling agent. This is the paper’s *measured* result (§10.6; the live single run — ≈ 475 -token in-cortex walk-and-recall vs ≈ 21.3 k for a cold spawn — corroborated by a reproducible N=5 sandbox, §10.7), and it is what the other contributions exist to deliver: co-location is precisely what lets an Agempus know a sibling holds the skill and call it privately, locally, and cheaply. A **distinct cross-owner property** — **portability** — follows by construction from the same guarantees but is scoped here as *future work*: the **empty-engram train invariant** (§4) keeps automatically-recalled personal node content out of the body, so the SOP that ships is source-grounded, not owner-grounded (hand-authored details and sensitive engram names remain the owner’s to review before sharing); the signed **.gsk** pack (AES-256-GCM payload, Ed25519 signature) — a **Graphnosis Skill Kit**, the portable *file* that serializes an Agempus’s skill subgraphs for another owner (the Kit is the shippable file; the *Skill Subgraph* is the live in-cortex concept this paper measures) — carries the full eight-goal structure and recall recipes across machines under integrity-by-signature; and the same `@skill`: resolution is *designed* to let a ported skill bind a *recipient’s* engrams at walk time. We have not yet evaluated an independent owner importing and running a pack — our intuition is that if borrowing works among Agempi within one cortex, a skill engram ported into another cortex should extend it — so that validation remains future work (§13); the empty-engram invariant is precisely what would make a skill safe to share.

We separate the contributions that deliver it into **research contributions** (empirically testable or formally provable claims) and **system and engineering contributions** (the deployable substrate).

Research contributions.

1. **Deterministic plan compilation** (Theorem 1, §5) — the executable plan for a Skill is a pure function of its nodes’ immutable storage order, not a traversal of its edges, so under the system’s canonical serialization it is byte-identical across runs and cannot recurse infinitely on a loop back-edge. Pinned by a golden regression test.
2. **Bounded plan contract for capped loops** (Theorem 2, §5) — a `skill:loop` edge may carry a finite iteration cap (`max=M`) and sub-skill calls are bounded by a recursion depth of three, so *any executor that honors the contract* halts on a finite skill graph whose loops are capped. This is a property of the plan *contract*, and — since the walker now re-executes loop bodies with caps enforced in code (§11) — also a runtime guarantee of the shipped walker for loop-bearing, non-parallel skills; an uncapped `@loop` lies outside the contract bound, though the shipped walker bounds it anyway with a default three-iteration cap (an executor decision, not a contract property).
3. **Structure-preserving training** (Proposition 2, §4) — the optional LLM rewrite is admitted only if a token-extraction validator confirms every DSL token and goal header from the source survives; otherwise the verbatim source is kept. The validated token set includes the per-step routing tag (`@needs`) and the recall binding (`only_engrams`), so a rewrite cannot silently *drop* the tag that selects a step’s model or the binding that selects its engrams (the guarantee is against omission, not a claim of value-level equality).
4. **Cross-skill-edge stability under retrain** (Invariant 3, §6) — in-place retraining reuses the skill’s source id, and because `@skill`: calls resolve by name to that id (at walk time intra-engram, through the side-table cross-engram), the call graph’s resolvability is invariant across retrains; name matching is case- and punctuation-insensitive, so a slug-versus-title-case mismatch cannot fork a duplicate source.
5. **The empty-engram train invariant** (Invariant 1, §4) — no *automatically-recalled* personal node content enters a skill body at training time; personal grounding is deferred to walk-time recall bindings. Hand-authored facts and sensitive engram names are outside the invariant and require owner review

before sharing (§4). A trained skill is therefore a source-grounded SOP, the analog of the substrate paper’s promotion gate between predicted and attested knowledge.

Contributions 1–5 are guarantees enforced *by construction*, not deep results; we formalize them because a procedural substrate should be held to them, and because they must survive in-place retraining, encrypted snapshotting, and cross-session resume. Where the shipped reference walker enforces less than the plan contract specifies — it runs the single-call path and iterates steps linearly, and re-executes capped loops with the cap enforced in code — leaving parallel `parallel[]` sub-skill dispatch as the sole remaining obligation on external executors — we say so explicitly (§5, §11).

System / engineering contributions. Built on the dual-graph substrate of the companion paper: Skills as typed-edge subgraphs that reuse the substrate’s single encryption key, append-only operation log, and snapshot versioning with no new storage type (§2–§3); **per-step capability routing** that selects the cheapest model meeting each step’s declared `@needs` under a privacy→strategy→cost precedence, with sensitive-engram steps hard-locked to local inference (§5, §7); **lazy dispatch** that keeps a large library out of the context window by loading a title-and-trigger index and hydrating only matched skills (§7); **single-authority, self-registering dispatch** in which one owner-held trigger table — the `trigger` → `skill` rows compiled into the trained `skill-dispatch` skill — is the single matching authority driving three consumers from one parse: a background proactive watcher (under explicit anti-spam bounds and a per-skill `[dispatch-safe:]` autonomy gate), implicit in-chat routing, and an export that renders the same table into the host AI client’s *own* configuration (a `CLAUDE.local.md` Skills block, a `.cursorrules` file, a Cursor `.mdc` rule), so a newly trained or amended skill registers itself with whatever agent is driving — writing only to project roots the owner configures, and updating its local registry otherwise (§7); reversible in-place versioning with encrypted per-snapshot history and automatic repair of crash-truncated skills (§6); a per-skill **vitality** model that drives a governed, user-confirmed retrain loop (the vitality computation, the staleness queue, and rollback are deterministic and license-free; the in-app retrain that the owner may then confirm uses the `skill-training` gate) (§6); and a portable, signed `.gsk` distribution format (AES-256-GCM payload + Ed25519 signature) carrying the full goal structure and recall recipes (§13). The deployed evidence is a **76-skill, self-built corpus** spanning code, business/GTM, and meta families, extended to a **108-skill corpus across twelve families** (eleven domains plus the chat agent’s own) as a *representational domain-compatibility* probe (**not** measured cross-user transfer) — compatibility under the same training method (§9, §9.1).

Empirical contributions. We report four results that isolate the procedural layer from the model running it; the first is a measured borrow-vs-spawn result (a live single run corroborated by a reproducible sandbox), the next two are exact accountings over the deployed corpus, and the fourth is a production before/after observation — none a benchmark of model quality.

1. **Borrowability — intra-cortex sibling-skill invocation** (§10.6) — one Agempus borrows a sibling’s single skill through an `@skill:` call instead of spawning the whole sibling agent. The `@skill:` target is resolved at training time via the encrypted cross-engram side-table, so *resolution itself costs no model tokens*; the borrow then pays only the callee’s body (mean 642, median 550 tokens by the system’s characters÷-4 tokenizer). Measured directly (E4, §10.7) — the measured borrow arm is a walk-and-recall, a **callee-side proxy** for an `@skill:` sibling call rather than a full cross-engram borrow walk (Appendix A): a **≈39–45× measured separation, harness-specific**, given by two datapoints — a **live** single run (≈475-token in-cortex walk-and-recall vs a ≈21.3k-token full-client cold spawn, ≈45×), the headline datapoint, corroborated by a **reproducible sandbox** re-measurement (5 cold spawns ≈25.0k tokens vs a deterministic 637-token borrow, 39.3×); the separately-quoted ~30× is against the ~700-token *modeled* typical borrow, not a measured datapoint. The architectural claim (Proposition 5) is the **constant warm-up avoided**: a borrow pays a bounded per-body cost while a spawn pays an unconditional warm-up co-location makes unnecessary — $O(1)$ in library size for a borrow, against the $O(N)$ cost of an agent that *eagerly inlines* its whole library (and, even against a fairly lazy-dispatched spawn, the constant warm-up W). Borrowing’s **resolution and walk are private and local by construction** — an in-process side-table lookup and walk within one owner-held, encrypted cortex, crossing no network and invoking no external trust for resolution and compilation; content locality of any recalled facts is conditional on local routing (§5).
2. **Per-step routing savings** (§10.3) — driving the real router over an annotated nine-step skill, adaptive

local+cloud routing runs **99.5%** below an all-Sonnet-4.6 baseline (eight of nine steps free and local) — a **measurement-time catalog** figure; the reproducible headline is the E2 corpus-wide cost accounting (~99.6% adaptive, ~96.6% cheapest-qualifying cloud-only, 73.3% coverage-max over 105 skills, §10.7). A premium *coverage-max* cloud-only policy saves **73.3%** by meeting each step’s @needs with Haiku — but that 73.3% is the complement ($1 - 0.2667$) of the Haiku/Sonnet price ratio on the pinned catalog, whereas the cheapest-qualifying rule (Definition 2) reaches ~96.6% cloud-only (corpus-wide, §10.7). These are cost figures only; per-capability quality retention is measured separately (§10.7). The 73.3% coverage-max row is invariant to the per-step token assumption; the _{96.6%}/99.6% adaptive rows are invariant only to uniform rescaling of that assumption. Only the absolute dollar figure scales.

3. **Lazy dispatch vs library inlining** (§10.2) — a **91.5–92.6% reduction**: a dispatched session loads roughly **8%** of the library, where inlining all 76 skill bodies costs 48,780 tokens by the system’s own characters-÷-4 heuristic, against ~3,600–4,100 tokens for the trigger index (~3,034 tokens) plus the one or two matched skills. We do not claim the mechanism as new — lazy loading is a known pattern; the contribution is the exact token accounting of it over this deployed corpus. Proposition 3 (§10.2) gives the structural reason the ratio is durable: the matched-body term depends only on the number of matched skills, not on library size, so — as long as a session keeps matching only a few skills — a larger library tends to strengthen the case. Being a ratio of two token counts counted the same way on each side, it is also robust to the tokenization heuristic.
4. **Empty-engram invariant, before/after** (§4, Appendix A) — a *production before/after observation* (not a controlled trial): skills trained before the empty-engram contract carried personal-memory dumps baked into their bodies and had to be scrubbed during the per-family migration, while skills trained under it were clean — a before/after observation consistent with the invariant separating portable procedures from privacy-contaminated ones (qualitative: exact contaminated/clean counts were not recorded).

These figures are accountings, not statistical results. The **corpus-wide routing accounting (§10.7, E2) reproduces from the shipped, pinned catalog and fixture**; the **dispatch decomposition (§10.2)** ships no body-token fixture, so its 48,780 / ~3,034 counts are measurement-time accounting over the corpus, not reproducible from anything shipped; the single-skill §10.3 figures reflect the *measurement-time* catalog (see the §10.3 catalog note), and the E4 live telemetry, the E3 read-only vitality records (a read-only sweep that corroborates the scoring *mechanism* behind §10.5’s 92→98 / 94→99 recovery, not the up-move itself), and the E6 quality record are reported live-cortex records with stated harness limits, not headless reruns. Two measurements are robust to the measurement heuristic for distinct reasons — the coverage-max routing percentage (73.3%) is invariant to the per-step token assumption (the adaptive rows are invariant only to uniform rescaling), and the dispatch ratio is robust to the characters-÷-4 tokenization — in both cases because each side of the comparison is counted the same way. Every recomputable saving follows from the *structure* (lazy dispatch, per-step routing) and the router, not from the license — walking, dispatch, routing-preview, import, text-format export, and rollback are license-free — so the **corpus-wide routing accounting (E2) is independently recomputable** against the shipped catalog and fixture, and the dispatch decomposition is recomputable from a caller’s own corpus; only the in-app training compile and signed .gsk pack export are gated (§14).

Methodological contribution. The corpus is **self-built**: an AI coding agent read the live repository and working sessions and trained the skills (several distilled from real debugging, so each SOP encodes what worked, not what docs claimed), and the system then improved itself through a **closed, on-device self-improvement loop** — **retrospective-learning** → **adaptive-skill-creation** → **train_skill** → **dispatch retrain** — in which a new procedure self-registers without data leaving the machine (under local-only routing) — always under owner confirmation; no shipped configuration performs autonomous writes (the L3 unattended executor is read-and-compute, off by default) (§8, §13). We keep a counterexample: skills *dispatched but never walked* produced no benefit, so the gain comes from walking the procedure graph, not from the library merely existing.

1.2 Why one substrate is essential

The case for co-locating procedure and memory is the same case the companion paper makes for owning your memory, extended one layer up. A Skill stored in the agent’s own knowledge graph is encrypted by the same key as the facts it depends on, versioned by the same snapshot machinery, and — critically — *checkable against those facts*: the vitality model can notice when a memory a skill cites has changed and queue the skill for retraining, something impossible when the procedure lives in a separate workflow engine. It can bind to private engrams at walk time (a fundraising SOP that reads a sensitive **investors** engram under a consent gate, routed local-only) while the SOP body itself stays generic and shareable. And it pays nothing extra for indelibility, encryption, or sync — it inherits them. A procedure that lives where memory lives is the smallest change that makes agent know-how durable, private, and self-maintaining at once.

2. The Procedural Layer over an Owner-Held Substrate

Skills are not a new store bolted onto Graphnosis; they are a *convention over the substrate the companion paper describes*. A trained Skill lives in a dedicated engram (template **skill**, shown as “Graphnosis Skills”), but the nodes and edges it is made of are the SDK’s ordinary primitives. The SDK’s **NodeType** enumeration contains no “skill” member, and its **DirectedEdgeType** enumeration contains no “skill” member; a Skill’s structure is carried entirely by each node’s **role** (stored in the node’s **source.section** field) and by the optional **evidence** string on otherwise-generic directed edges. Adding the procedural layer therefore introduced *no new graph engine* — the discipline is that every property the substrate guarantees for memory holds, unchanged, for procedure. Because the substrate permits several typed edges in parallel between the same pair of nodes — multiple **skill:*** relations, or a **skill:*** edge alongside a memory **contains** edge — it is a *multi-graph* in the strict sense (the term the companion paper’s title carries), not a simple graph; a Skill simply adds new edge *families* over the same nodes.

Proposition 1 (Zero-primitive procedural extension). *The skill layer adds no member to the SDK’s **NodeType** enumeration and no member to its **DirectedEdgeType** enumeration: $\text{SkillNodeTypes} \subseteq \text{NodeType}$ and $\text{SkillEdgeTypes} \subseteq \text{DirectedEdgeType}$, where both right-hand sets are the substrate’s pre-existing enums. All skill semantics are carried by (i) each node’s **role** and (ii) the optional **evidence** string on otherwise-generic directed edges; the procedural layer is thus a labeling discipline over the unchanged memory-graph schema — there is no new graph engine. Proof (by inspection). **Node half:** every skill node is an ordinary substrate node carrying a stock **NodeType** (the substrate’s **concept/fact/section-class** members) — the leading **metadata** comment node, the title hub, the inline-goal nodes and the eight structured goal nodes, the ordered body/step nodes, the **recipe** recall steps, and trailing **recalled-memory** nodes all reuse existing **NodeType** values, their skill role held instead in the node’s **source.section** field (§3), so no enum member named **skill** is added to **NodeType**. **Edge half:** each of the five control-flow edge classes maps onto a stock directed-edge type — **skill:seq/skill:loop** → **precedes**, **skill:branch** → **depends-on**, **skill:ctx** → **supports**, **skill:calls** → **contains** (§3) — as does the goal-attachment edge **skill:goal** → **contains** (a stock **DirectedEdgeType**, Definition 1), with the class held only in the **evidence** string, and no enum member named **skill** exists in either substrate enumeration. ■ (Verifiable by inspection of the enums and the §3 mapping table, not a deep result; this is the formal anchor of the “not a workflow engine” defense, §12.)*

Three inheritances matter. **One key.** A single derived data key encrypts the skill nodes, the encrypted per-snapshot history files, the cross-engram call side-table, and the resumable-run records — the same key that encrypts every declarative memory in the cortex; there is no separate skill keystore. **One op-log.** Every skill-node insertion and reordering emits **addNode** / **reorderSource** events through the same append-only, signed operation log as memory writes; the log is an *audit and sync* channel, and recoverable state is reconstructed from the sources and snapshots, not by replaying the log. **One indelibility discipline.** A retrain never hard-deletes: the prior live nodes are captured to an encrypted snapshot, then soft-deleted (**validUntil** set, confidence driven to zero) through the very correction pipeline memory uses for supersessions, and new nodes are inserted under the *same* source id. The worst case for a superseded skill node, exactly as for a

memory, is reduced prominence, never erasure (companion paper, Theorem 1).

One boundary is worth stating plainly, because it separates this paper from its companion. The *substrate* — generic nodes, typed edges, op-log, encryption — is the SDK (@nehloo/graphnosis); the entire *skill layer* — training, walking, routing, vitality, dispatch, packaging — is the application sidecar (apps/desktop-sidecar). The companion paper is about the former; this paper is about the latter, built on top of it.

3. The Skill as a Typed-Edge Subgraph

Definition 1 (Skill subgraph). A trained Skill is a title hub node, a set of typed goal nodes partitioned over eight categories — *Trigger*, *Prerequisites*, *Requires*, *Produces*, *Success*, *Out of scope*, *On failure*, *On completion* — an ordered body of step nodes, and an edge set drawn from five evidence-tagged control-flow classes — **skill:seq**, **skill:loop**, **skill:branch**, **skill:ctx**, **skill:calls** — over those nodes, with the goal nodes attached to the hub by **skill:goal** contains-edges. The procedure *is* this subgraph, by construction rather than by interpretation.

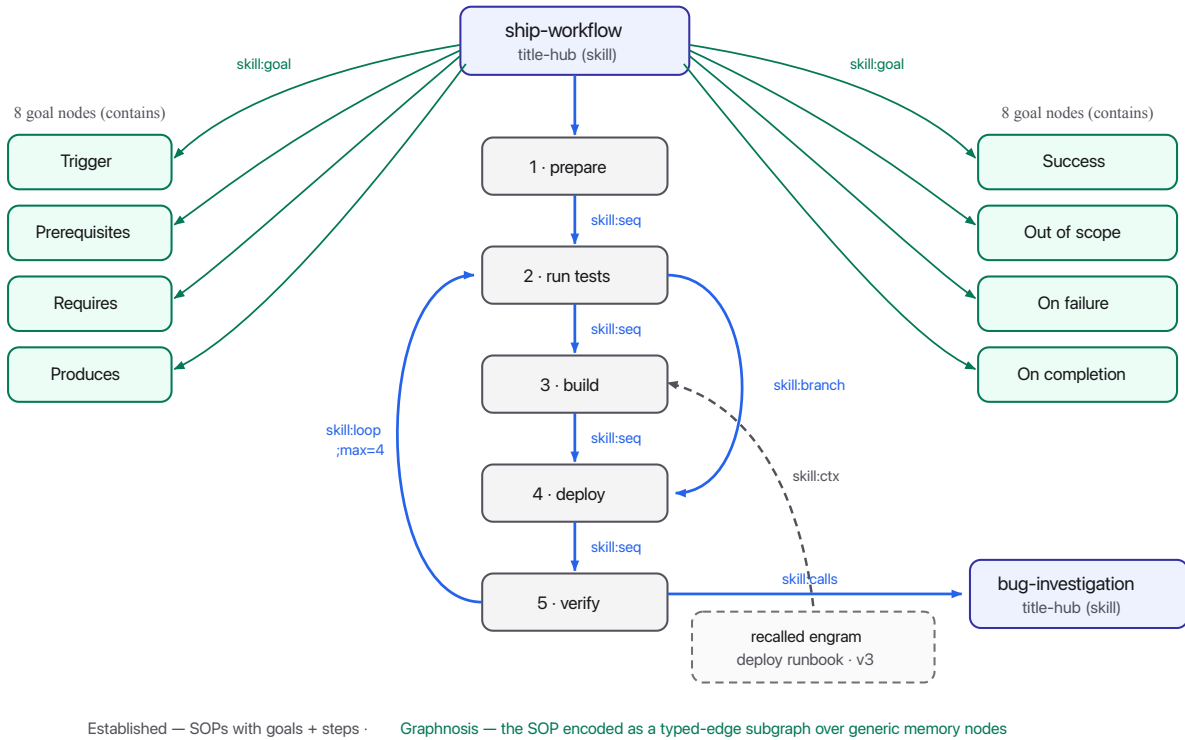


Figure 1. Anatomy of a Skill as a subgraph. The trained skill **ship-workflow** is one subgraph on the memory substrate: an indigo title-hub, eight emerald goal nodes joined by **skill:goal** (contains) edges, and a blue ordered step chain (**skill:seq**), with a bounded **skill:loop** back-edge (**;max=4**), a **skill:branch** forward skip, a dashed recalled-context node bound by **skill:ctx**, and a **skill:calls** edge to a second skill (§3).

Eight goal categories. A Skill states its contract in eight typed goal nodes — *Trigger*, *Prerequisites*, *Requires*, *Produces*, *Success*, *Out of scope*, *On failure*, *On completion* — parsed from line prefixes at training time and lifted out of the step body into a meta-header hung off the title hub by **contains** edges (evidence **skill:goal**, weight 0.85). The categories are not decoration. *Requires* and *Produces* type the skill’s inputs and outputs, which is what lets calls bind arguments and capture returns (§5); *Trigger* carries a **[dispatch-safe: yes|partial|no]** tag declaring whether the skill is safe to auto-propose — now read by

the autonomy gate that caps proactive dispatch (§7, §13); *Success* carries a [verify: tool|state|human] completion oracle that tells an executor how to know it is done; and *On failure* names the recovery skill to run if a step throws. Goals deliberately hang off the title hub rather than joining the step chain, so recalling any single goal lets a traversal reach the title and, from there, the whole procedure.

Node roles and order. Each node carries a `role: metadata, title`, one of eight `goal-*` roles, `body`, `recipe` (a body step that is a recall recipe), or `recalled-memory`. Training inserts them in a fixed, presentational order — `metadata` comment, `title`, inline goals, the eight structured goals, body steps, then any trailing `recalled-memory` — and the structure is recoverable from node order and role alone, independent of edge state.

Five edge classes carry control flow as structure. Each maps onto a stock SDK directed-edge type, with the skill semantics in the `evidence` string and a characteristic weight:

Class	SDK edge	Weight	Meaning
<code>skill:seq</code>	<code>precedes</code>	0.90	do step n , then $n+1$ — the canonical chain
<code>skill:loop</code>	<code>precedes</code>	0.70	a bounded back-edge to an earlier step, carrying <code>;max=N</code>
<code>skill:branch</code>	<code>depends-on</code>	0.75	a conditional forward skip
<code>skill:ctx</code>	<code>supports</code>	0.60	a recalled-memory node bound to the step it informs
<code>skill:calls</code>	<code>contains</code>	0.95	a sub-skill invocation, carrying <code>;args=...;capture=...;onFailure=...;parallel=...</code>

Edge derivation is **idempotent**: each linker first deletes every edge bearing its own evidence tag that touches the source, then rebuilds from the current nodes — so the subgraph is a pure function of node text and order, stable across any insert, reorder, or retrain.

Lemma 1 (Edge-derivation idempotence). *For each skill-edge class $c \in \{\text{seq}, \text{loop}, \text{branch}, \text{ctx}, \text{calls}\}$, the linker L_c first deletes every edge whose evidence tag is c (bare or* suffixed, e.g. `skill:loop;max=N`) incident to a source, then re-emits edges purely from that source’s current node text and storage order. Model L_c as a map on graph state $(N, E) \mapsto (N, (E \setminus E_c^{\text{own}}) \cup E_c(N))$, where E_c^{own} is the incident edge set tagged c and $E_c(N)$ is the freshly derived class- c edge set — a pure function of node text and order alone. So stated, L_c is idempotent: $L_c(L_c(N, E)) = L_c(N, E)$, since the delete-own-tag step makes the second application reproduce the identical delete-then-rebuild. The control-flow overlay (all five classes together) is therefore a deterministic function of (node text, node order), and — since the goal `contains` edges of Definition 1 are themselves derived deterministically from role and line-prefix at compile time (see proof) — so is the full skill subgraph.* *Proof.* The delete-all-own-tag step makes L_c insensitive to edges it previously wrote, so re-applying it reproduces the same delete-then-rebuild and the identical edge set — a fixpoint; the rebuild reads only node text and order, so E_c depends on no prior edge state. Hence the control-flow overlay (the five c classes) is a deterministic function of (node text, node order); the goal `contains` edges of Definition 1 are outside this quantifier but are themselves derived deterministically from each node’s role and line-prefix at compile time, so the whole skill subgraph — control-flow overlay plus goal edges — is a deterministic function of (node text, node order) alone. ■ (Enforced by construction — and now regression-tested after a real bug the review surfaced: the linker’s delete step matched only the bare* `skill:loop` tag, missing the capped `skill:loop;max=N` form, so capped-loop edges accumulated across re-derivations and the fixpoint failed (silently weakening Theorem 1 and Invariant 3). The deletion predicate now matches bare and suffixed forms; `skill-loop-idempotence` (14/14, verified red before the fix,*

§10.4) asserts that changing `@loop: 2 max=4` to `max=5` and re-deriving leaves exactly one loop edge. This lemma is the load-bearing reason Theorem 1 and Invariant 3 hold.)*

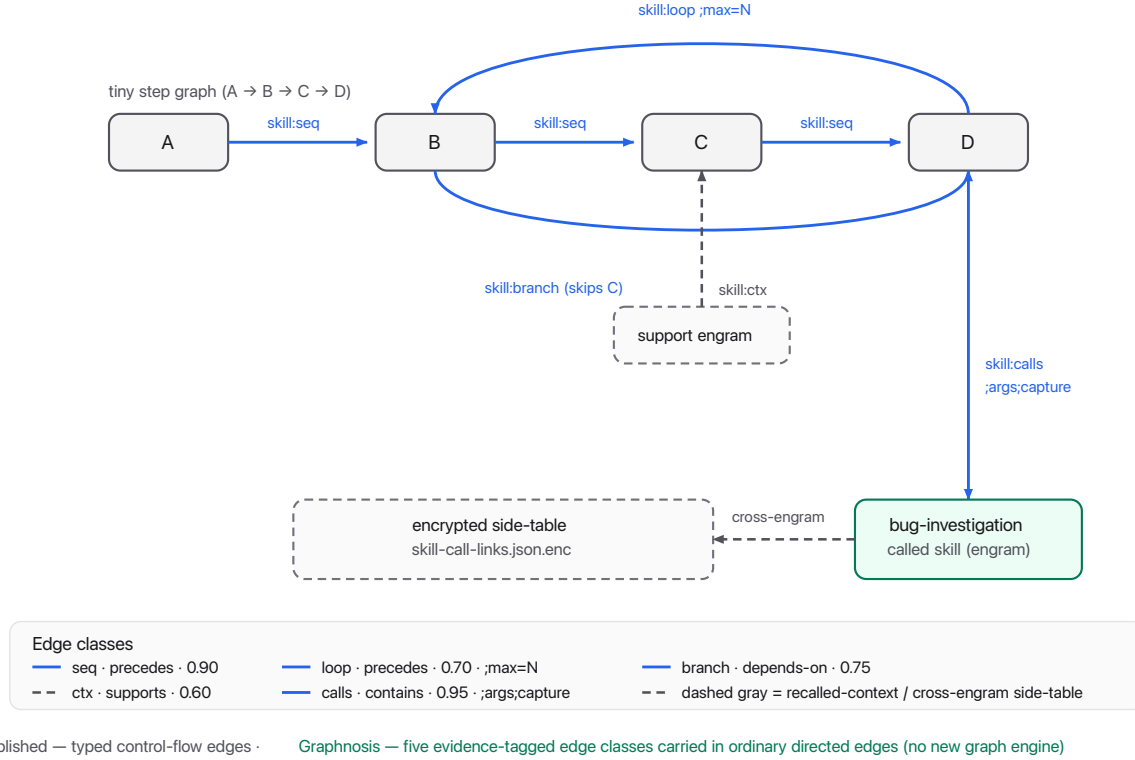


Figure 2. The five skill edge classes on a small step graph, each a stock SDK directed edge carrying skill semantics in its **evidence** string: `skill:seq` (precedes, 0.90), `skill:loop` (precedes, 0.70, ;max=N), `skill:branch` (depends-on, 0.75), `skill:ctx` (supports, 0.60), and `skill:calls` (contains, 0.95). A cross-engram call cannot be a graph edge and is routed through the encrypted `skill-call-links.json.enc` side-table (§3).

A small step DSL. Control flow and bindings are authored inline in step text and parsed at training time: `@loop: N max=M` (and the wiki form `[[loop: ...]]`) produces a `skill:loop` edge carrying its cap; `@branch: N` produces a `skill:branch` edge; `@skill: name(arg=$var) -> $capture` produces a `skill:calls` edge with structured argument and capture metadata; and `only_engrams=[...]` on a recall step binds it to specific engrams. Loop and branch references are detected multilingually (the trainer ships keyword pattern sets for several languages).

One distinction matters for the “control flow as graph structure” claim. `@branch` and `@loop` *do* become graph edges; `@needs`, the per-step model-routing tag, does **not** — it is parsed from the step’s text at walk time (§5) and never materialized as an edge. Routing metadata is therefore graph-*resident* (it lives in a node’s text) but not graph-*topological*. We state this rather than overclaim that routing is encoded in edge structure.

One implementation reality. An SDK directed edge joins two nodes within a single graph, so a `@skill: call` that targets a skill in a *different* engram cannot be a graph edge. Such cross-engram calls are resolved at training time and persisted in an encrypted side-table (`skill-call-links.json.enc`, under the same cortex key), which the walk consults; intra-engram calls remain ordinary `skill:calls` edges. We surface this as a constraint of reusing the substrate’s strictly intra-graph edge model, not as a design we would otherwise have chosen.

4. Training: Deterministic Compilation, Not Memory Fusion

“Training” a Skill is not gradient learning and, in the shipping system, not memory fusion either. It is a deterministic *compile*: authored SOP text is parsed, chunked into the typed nodes of §3, and linked into the subgraph. The naming follows the user’s vocabulary — one trains a code-review skill the way one trains a colleague — but the operation is closer to compilation than to fitting.

Invariant 1 (Empty-engram training). *Let $\text{body}(K)$ be the node set of trained skill K ’s compiled body and $\text{cortex}_{\text{personal}}$ the personal memory nodes. With train-time federated recall gated off ($\text{ENABLE_CORTEX_RECALL_AT_TRAIN} = \text{false}$), recall at training is scoped to an empty engram, so on the **default (no-recipe-binding) path** no automatically-recalled personal node content enters the body — stated at the content level: no auto-recalled node of $\text{cortex}_{\text{personal}}$ has its text occur in any node of $\text{body}(K)$. On that same default path — absent hand-authored personal facts — $\text{body}(K)$ is a pure function of K ’s authored source, and personal grounding occurs only at walk time through recall bindings (§5). A trained skill is therefore portable and source-grounded — shareable as a signed .gsk pack without leaking the cortex it was trained beside. (Enforced by a module constant on the **default** empty-engram path. Scope, after review: this guarantees no automatically-recalled personal node content* enters the body. It does **not** cover (i) the opt-in **bind_recipes** path, which derives **only_engrams** recipes from cortex state — a query-plus-engram-name binding, not frozen content; nor (ii) hand-authored facts or sensitive engram names the author writes into the source, which remain the owner’s to review before sharing. So “pure function of authored source” holds for default training without recipe binding; “no leakage” means no automatic frozen personal node content, not “no authored or metadata leakage.” The procedural analog of the companion paper’s promotion gate; this invariant is what underwrites the cross-owner *portability* scoped as future work in §1.1.)**

This invariant is the procedural analog of the companion paper’s promotion gate between predicted and attested knowledge, and it is what makes a trained skill *portable*: because no personal cortex node is *automatically* woven into the body, a skill can be exported and shared (§13) without leaking the cortex it was trained beside — hand-authored details and sensitive engram names remain the owner’s to review before sharing. It also keeps SOPs legible — a step reads “recall the prior tier decisions,” not a frozen paste of last quarter’s pricing.

Two modes, one structure contract. Training has a free, deterministic path and an optional paid one. In **memory-augmented** mode (the default, no model in the loop) the compiled body is the authored source verbatim, chunked and typed. In the **Pro** path, a local LLM may rewrite the prose for clarity — but with no cortex recall and no fabricated “(from memory...)” attributions, and, crucially, under a structure gate:

Proposition 2 (Structure preservation — no total omission). *Let $\text{tok}(s)$ be the set of structural tokens of a skill source s — every [ANCHOR] line, every DSL token (**@skill:**, **@loop:**, **@branch:**, **@parallel:** and wiki forms), the routing tag **@needs**, the recall binding **only_engrams**, and all eight goal-header prefixes — and let r be an optional LLM rewrite. The saved skill is $\text{save}(s) = r$ if $\text{tok}(s) \subseteq \text{tok}(r)$, else s ; hence $\text{tok}(\text{save}(s)) \supseteq \text{tok}(s)$ always, so training can never silently drop an executable token, a routing tag, a recall binding, or a goal header. Proof. The validator lifts $\text{tok}(s)$ and checks each element survives in r ; on any miss the pipeline reverts to memory-augmented mode, where $\text{save}(s) = s$ and the inclusion is trivial; when admitted, $\text{tok}(s) \subseteq \text{tok}(r)$ by the gate. A model-free suite checks both branches (9/9). ■ (De-rated after review: the validator checks **set-presence** via deduplicated substring match, not per-step placement or count — so this guarantees no structural token is **entirely omitted**, not that each survives on its original step; a rewrite could relocate a token to a different step. Per-step ordered verification is future work. The rewrite is a detachable cosmetic overlay.)*

The gate is a token-extraction check: it lifts every anchor, every **@skill:** / **@loop:** / **@branch:** / **@parallel:** / wiki-form token, the per-step routing tag **@needs**, the recall binding **only_engrams**, and all eight goal-header prefixes from the source, and confirms each survives the rewrite; on any miss it reverts to memory-augmented

mode and records a degradation note. The rewrite is therefore a detachable cosmetic overlay — exactly the determinism-first posture the substrate paper takes toward its LLM and GNN layers.

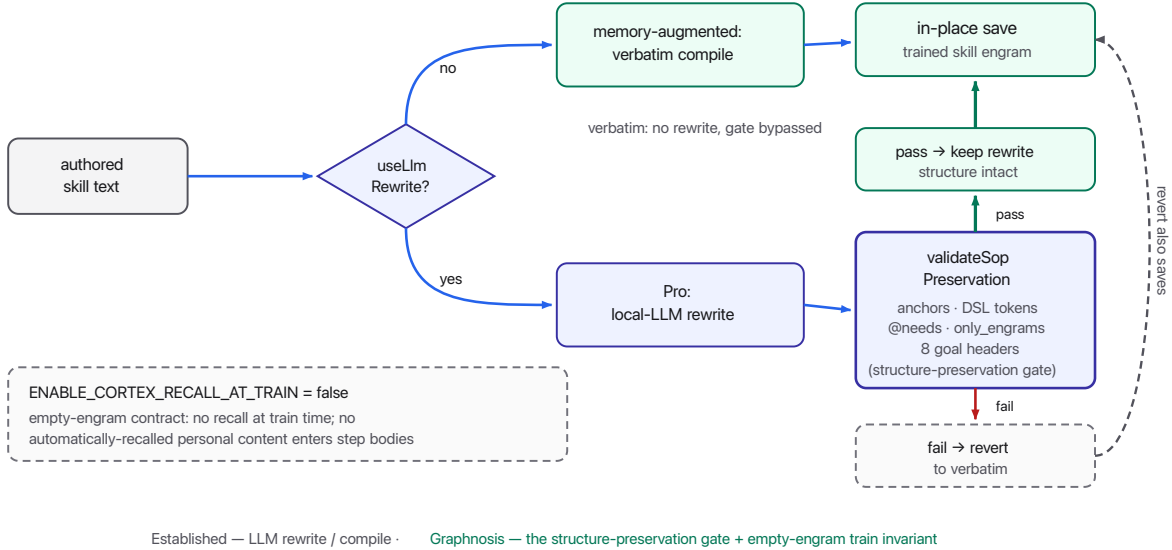


Figure 3. Training as a structure-preserving compile. The optional Pro rewrite is admitted only if `validateSopPreservation` confirms every anchor, DSL token, `@needs`, `only_engrams`, and goal header survives; otherwise the verbatim source is kept. Training pulls no cortex memory (`ENABLE_CORTEX_RECALL_AT_TRAIN = false`) — the empty-engram contract (§4).

What is gated, stated plainly. The training pipeline contains machinery for memory augmentation — a relevance floor, a self-tuning recall breadth, an “influential nodes” ranking — that is *dormant* under the empty-engram invariant above. We describe it as designed-but-gated, not as a live self-tuning system, so as not to claim behavior the shipping configuration does not exhibit. If the gate were opened, recalled paragraphs would be appended as trailing **recalled-memory** nodes (never interleaved into the numbered procedure) under an explicit `_(from ...)_` attribution; today they are not.

Saving is a re-compile. On every train or retrain the source is rebuilt from scratch — metadata comment, title, inline goals, the eight structured goals, body steps, trailing recalled-memory — after which all SOP edges are re-derived by the idempotent linkers of §3. The training pipeline — the deterministic memory-augmented compile of §4 together with the optional Pro rewrite — is gated behind an Ed25519-signed **skill-training** license feature checked at the transport layer; without it a user may still store, import, and walk skills and export them in any text format, but not run a compile in-app or export a signed `.gsk` pack. The gate guards the in-app training surface, not the *value* a trained skill carries: its typed-edge structure, its per-step routing, and the token-cost savings of §10 are properties of the artifact and the router, available to anyone who can walk it. Versioning, snapshotting, and the retrain mechanics of this re-compile are the subject of §6.

5. Execution: Compiling and Walking the Plan

Running a Skill is a two-step affair: compile the subgraph into an execution plan, then walk the plan. The compile is a pure read, and it is where the strongest determinism guarantee lives.

Theorem 1 (Plan-compilation determinism). *Let a Skill K have ordered live body nodes*

$(n_1, \dots, n_m)$ in storage order (soft-deleted nodes filtered) and edge overlay $E(K)$. The compiler Π producing K 's execution plan is a pure function of $(\text{nodes}(K), E(K))$: its step list is the index-ordered image $\langle n_i \mapsto \text{step } i \rangle$, never an edge traversal, and loop/branch/call overlays are deterministic lookups keyed by node id. For a fixed graph state $\Pi(K)$ is byte-identical across runs under its canonical serialization, and a **skill:loop** back-edge cannot drive Π into nonterminating recursion. *Proof.* Steps are built by mapping the body-node array to positions (step i is array index i), not by following edges; a back-edge $n_j \rightarrow n_i$ ($j > i$) is recorded only as a bounded annotation on step j and never re-enters the construction (Lemma 1 makes the edge set itself a pure function of node text and order). The serialized plan is asserted byte-equal to a stored fixture (7/7). ■ (De-rated: enforced by construction, golden-test-pinned. Scope: this is the compile; loop re-execution is an executor obligation — Theorem 2.)

Reconstructing the chain from storage order rather than edge traversal is the load-bearing choice: it is what lets **skill:loop** express a real back-edge in the *graph* while the *plan* stays a finite, ordered list with the loop recorded as a bounded annotation on a step.

Two surfaces, one compile. The same compiled walk is rendered two ways. **walk_skill** produces human-readable narrative — CONSTRAINTS and PROCEDURE prose with loop/branch/sub-skill annotations — for explaining a procedure to a person. **walk_skill_structured** produces a machine contract, a **SkillExecutionPlan**: **requires**[] (with inline type hints) and **produces**[], then ordered **steps**[], each optionally carrying **calls** (a sub-skill with **args** and **captureAs**), **parallel** (sibling sub-skills to dispatch concurrently), **maxIterations** (a loop-convergence cap), **branchesTo** / **loopsBackTo** targets, and **supportingContext**, plus top-level **failureHandlers**. An AI executor walks the steps in order, invokes sub-skills with the named arguments, captures their returns under the named variables, and routes exceptions to the matching handler.

Bounded control flow gives termination. Loops and recursion are both capped:

Theorem 2 (Boundedness of the plan contract — enforced by the shipped walker for loops; parallel dispatch still external). *This bounds any executor that honors the plan contract. The shipped reference walker now honors the loop side of that contract in code — it re-executes loop bodies with each cap enforced by its scheduler, bounds uncapped loops with a default cap, and is regression-pinned by a model-free suite (§11) — so for capped-loop, non-parallel skills the bound is also a shipped-system guarantee; parallel[] dispatch remains an obligation on external executors. Let a Skill K have body steps with **skill:loop** back-edges each carrying a finite cap M_ℓ , forward-only **skill:branch** edges, sub-skill calls forming a tree of at most $D = 3$ executed levels (the executor returns early at recursion depth ≥ 3 , so depths 0, 1, 2 execute), and at most one recovery sub-skill plus one retry per throwing step, where **a recovery sub-skill's own execution is a sub-skill call that occupies a depth level** — it descends the same $D = 3$ recursion ladder as any other call and is refused at depth ≥ 3 . Then for any executor honoring the caps the number of step executions is **finite**: each capped back-edge fires at most M_ℓ times, recursion is hard-bounded at $D = 3$ executed levels (a recovery handler that itself throws can invoke a further handler only within that same depth budget, so mutual On failure* handlers cannot recurse without bound), branch edges are forward-only, and failure recovery adds at most one execution per step — a finite unrolling of a finite graph — hence every walk over a finite skill graph whose loops are capped halts. We claim **finiteness only, not a max-per-step bound**: with multiple or nested reachable loops the exact per-step execution count is executor-specific.* *Proof.* Each back-edge runs $\leq M_\ell$ times by hypothesis; failure recovery adds ≤ 1 execution per step (the factor 2) and each recovery run is itself a sub-skill call that consumes one level of the depth budget, so recovery cannot escape the recursion cap; branch edges are forward-only; recursion is hard-capped — at depth ≥ 3 the executor returns early, so depths 0, 1, 2 execute — giving the call tree $D = 3$ finite levels, and summing the per-level bound (including recovery calls, which live on the same ladder) over those D levels gives the finite total. ■ (Scope: the bound covers skills whose loops carry a finite cap — an uncapped @loop lies outside the contract bound, though the shipped walker bounds it too, with a default three-iteration cap. Termination is an obligation on any contract-honoring executor; the shipped reference walker enforces the depth-3*

*cap, single-pass recovery, and loop re-execution with in-code cap enforcement plus a hard per-walk execution backstop; **parallel**[] siblings remain undispatched — §11.)*

Sub-skills compose like functions. A `@skill: name(args) -> $capture` step binds its arguments into the child walk and captures the child’s output under the named variable for downstream steps; recursion is capped at depth three. Failure recovery is equally bounded: when a step throws and an *On failure* handler is present, exactly one recovery sub-skill runs and the step is retried exactly once — a fixed, runaway-free recovery.

Per-step routing. Each step’s `@needs` tag is parsed into a capability set and resolved, by a read-only planner, to the cheapest model whose capabilities cover it, under a `privacy→strategy→cost` precedence; steps that touch an engram *flagged* sensitive are hard-locked to local inference regardless of strategy — the lock is over what the sensitive flag covers, not a property the planner can infer for unflagged content. The planner is deterministic and side-effect-free, so a walk’s cost can be previewed before a single token is spent; we quantify the savings in §7 and §10.

Definition 2 (Per-step routing selection). *For a step s with declared capability set $\text{needs}(s)$ and active model catalog $\mathcal{K}$, the read-only planner selects $\text{model}(s)$ lexicographically under the precedence $\text{privacy} \succ \text{strategy} \succ \text{cost}$, over the feasible set $\mathcal{F}(s) = \{m \in \mathcal{K} : \text{capabilities}(m) \supseteq \text{needs}(s)\}$: if s recalls a sensitive engram, $\mathcal{F}(s)$ is first restricted to local models (Invariant 2); the active strategy then imposes a primary ordering on $\mathcal{F}(s)$ — under a cost-first strategy (**adaptive**, **local-only**) the ordering is $\arg\min_{m \in \mathcal{F}(s)} \text{cost}(m)$, whereas under **always-best** (coverage-max) it is $\arg\max$ over capability coverage — and cost breaks ties only within that strategy ordering (so the cost-argmin governs the cost-first strategies, not **always-best**). Residual ties — including the free-local ties at cost 0 that a cost-first strategy leaves undistinguished — are broken by typical-latency, then by stable catalog order (per `model-router.ts`), so the ordering is total. The selection is deterministic and side-effect-free, so the previewed total $\sum_s \text{price}(\text{model}(s))$ is computable before any token is spent; if some $\mathcal{F}(s) = \emptyset$ the walk is infeasible under that strategy. (Definitional; its evaluation over the nine-step **ghampus-operator** against the static catalog is the §10.3 measurement.)*

Invariant 2 (Privacy hard-lock). *This is a **per-step** guarantee. For every step s whose own recall touches an engram *flagged* sensitive, $\text{model}(s) \in \text{LocalModels}$ independently of the active strategy: the planner restricts $\mathcal{F}(s)$ to local models before the cost minimization of Definition 2, so that step routes local — within the scope of that flag, privacy is not traded for cost. Proof (by construction). The privacy gate is the highest-precedence filter; cost minimization ranges only over the already-restricted feasible set. ■ (Now wired and regression-tested. The planner always supported the lock, but until this revision **no walker passed the per-step tier map**, so it did not engage; the walk paths — **agent-walker**, the autonomous **unattended-executor**, and the IPC preview/execute paths — now resolve each step’s recalled-engram tiers and pass them in. A model-free suite (**skill-privacy-lock**, 17/17, §10.4) asserts a sensitive-recall step routes to a local model even with all cloud providers enabled, with a control case proving the wiring is load-bearing. Scope: the lock is **per-step only** and there is **no taint propagation** — content derived* from a sensitive recall at one step and captured into a variable that flows to a later, cloud-routed step is **not** covered; only steps whose own recall touches the flagged engram are locked. It is also conditional on correct sensitive-flagging — an unresolved engram name is conservatively treated as sensitive (lock-when-unsure); behavior under a mis-flagged engram is out of scope. Illustrated by **fundraising-readiness** — §9.)**

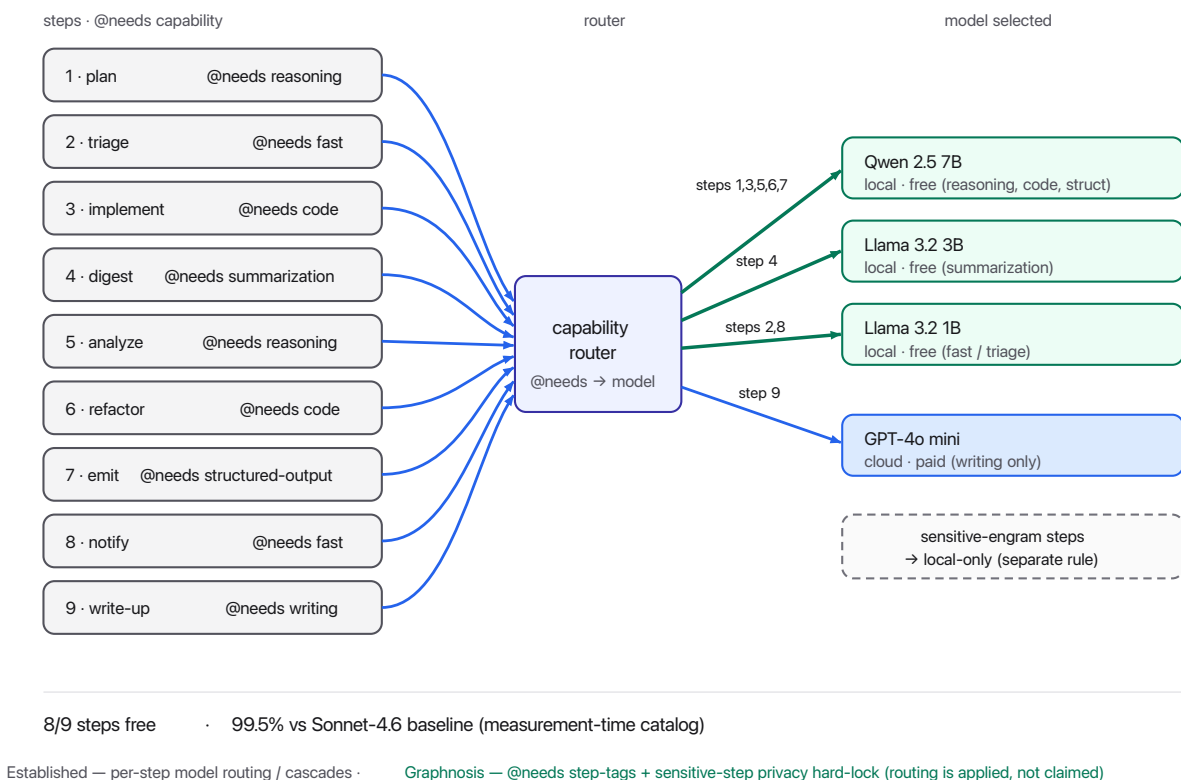


Figure 4. Per-step model routing. Each step’s @needs capability is resolved to the cheapest qualifying model; across ghampus-operator’s nine steps, eight run on free local models and only the **writing** step reaches a paid cloud model — 99.5% below the all-Sonnet-4.6 baseline (\$5, \$10.3) — a **cost** figure under the *measurement-time* catalog (the shipped catalog routes all nine local; see the \$10.3 catalog note); E6 (§10.7) finds local quality holds on mechanical steps and collapses on reasoning/writing. This is the *per-skill* view; the corpus-wide **cost accounting** across 105 skills is Figure 11 (§10.7). The privacy hard-lock — sensitive-engram steps forced to local inference (Invariant 2) — is a separate guarantee, illustrated not by this routing example but by **fundraising-readiness** (§5, §9).

Resumable runs. Because captured variables normally live only for one conversation, a multi-skill walk can persist its state — captured variables and the last completed step — to an encrypted run record and resume in a later session from the next step. Long procedures therefore survive a closed laptop.

What the reference walker does and does not do (stated plainly). The in-application walker executes the `single-calls` path, iterates steps in order, and *re-executes loop bodies*: a `loopsBackTo` edge re-enters its body up to the cap, which the walker’s scheduler enforces in code — the authored `max=M`, else a default cap of three for an uncapped `@loop` — with an early exit when consecutive body outputs reach a fixed point and a hard per-walk execution backstop behind both. Caps are lifetime budgets per edge within one walk (an enclosing loop never resets an inner edge’s budget), so every walk terminates by construction, and each walk reports its per-edge iterations, cap source, and stop reason for audit. Reaching a cap is a normal, reported outcome — the walk falls through and completes — in contrast to global-recursion-budget engines, where exhausting the budget raises an error and fails the run (the design contrast is drawn against LangGraph in §12). The one plan feature the walker still does not perform is `parallel[]` concurrency: dispatching parallel siblings remains an obligation the contract places on external executors. We are explicit about this division so the termination claim above is read correctly — for loops it is enforced by the shipped walker; for parallel dispatch it remains a property of contract-honoring executors only.

6. Lifecycle: Vitality, Versioning, and Governance

A procedure that cannot tell when it has gone stale is a liability dressed as an asset. The Skill Subgraph gives every Skill a freshness score, a reversible version history, and a governed retrain loop, so that aging is visible and correctable rather than silent.

Per-skill vitality. Each Skill carries a 0–100 score that starts at 100 when trained and decays by three bounded penalties:

```
score = 100 - agePenalty - stalenessPenalty - citedDriftPenalty
agePenalty      = min( floor(months_since_trained * 5), 25 )
stalenessPenalty = round( superseded_nodes / total_skill_nodes * 50 )
citedDriftPenalty = round( missing_cited_nodes / total_cited_nodes * 40 )
```

with bands *fresh* ≥ 80 , *aging* 60–79, *retrain* 40–59, *stale* < 40 . The three terms encode the three ways a procedure rots: it ages, its own nodes get superseded by partial edits, and the external memory it cites drifts out from under it. (This per-skill score is distinct from the cortex-wide vitality of the substrate — connectivity 0.40, confidence 0.25, activity 0.20, coherence 0.15 — and the two should not be conflated.)

A measured baseline. Across the 76-skill corpus the model reports a healthy library: every skill *fresh*, mean 97.7, median 98, minimum 92 — i.e., the library was healthy at measurement time (a snapshot; the scores do not by themselves prove the maintenance machinery caused them). The score also moves the right way under intervention: a clean re-compile of the two lowest skills lifted them 92→98 and 94→99, because retraining resets the age penalty and clears the bulk of the superseded-node staleness (a small residual keeps the scores at 98/99, not a perfect 100 — the recompile does not necessarily re-ground every cited node). Notably, the score does *not* reward goal completeness or structural resolution — a deliberately narrow definition of freshness we flag in §11, since it means the goal-completeness work of §9 improves the SOP without registering as a vitality gain.

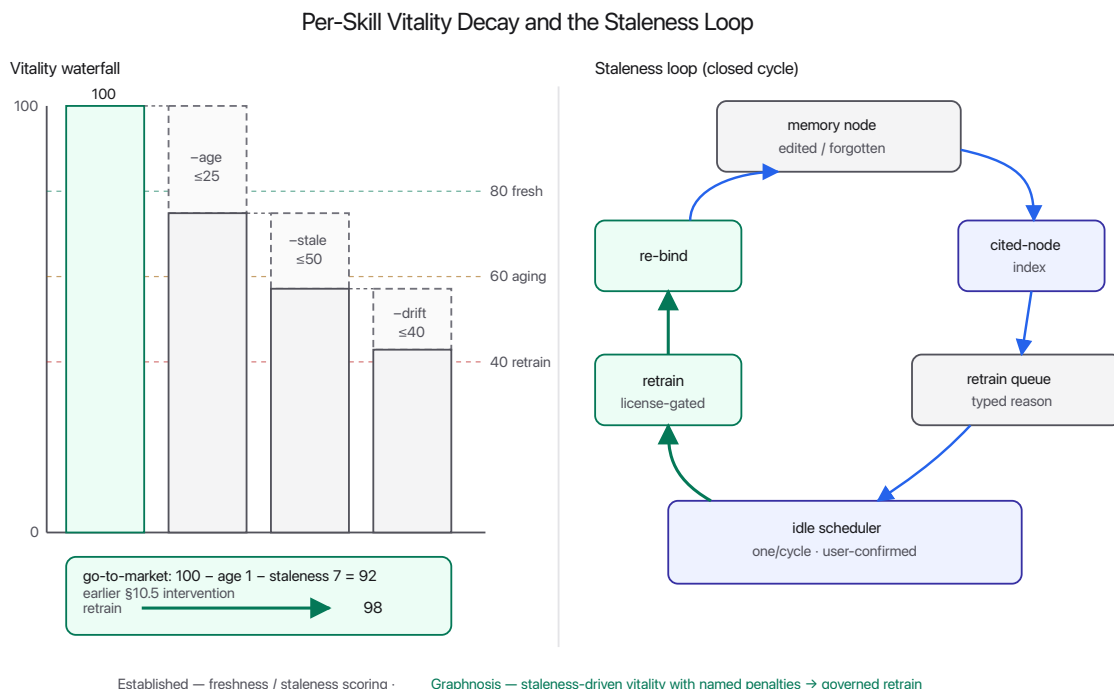


Figure 5. Per-skill vitality and the staleness loop. Vitality falls from 100 by three bounded penalties (age ≤ 25 , staleness ≤ 50 , cited-drift ≤ 40) into fresh/aging/retrain bands; editing a cited memory node enqueues the dependent skill for a governed, user-confirmed retrain. A clean re-compile lifted **go-to-market-planning** from 92 to 98 (§6, §10.5).

In-place, reversible versioning. A retrain reuses the skill’s single source (§4); before mutating, the prior live nodes are written to an encrypted per-snapshot file (one file per snapshot, appended in $O(1)$, retained indefinitely), which is what powers version history and rollback. Rollback first snapshots the *current* state, then replays the chosen snapshot — so a rollback is itself reversible and the lineage is never lost. An earlier model created a fresh source on every retrain, leaving “red island” orphans in the graph; collapsing to one source per skill is what keeps the call graph resolvable (§1.1, contribution 4). A crash between *clear* and *re-insert* would leave a **hollow** skill — a labeled source with no walkable nodes — so reads auto-repair it by replaying the newest non-empty snapshot; the fix that made retrain safe was reordering it to rename the source *after* inserts succeed rather than before.

Invariant 3 (Cross-skill-edge stability under retrain). *An in-place retrain of skill K reuses K ’s source id, so $\text{id}(\text{retrain}(K)) = \text{id}(K)$; because a `@skill: call` resolves its target by name* to a source id — intra-engram at walk time, cross-engram through the encrypted **skill-call-links** side-table — reusing the id keeps every incoming call resolvable without rewriting anything on the caller. Name matching is normalized so $\text{norm}(\text{slug}) = \text{norm}(\text{title-case})$ while genuinely distinct names stay distinct. Consequently, for any caller C with a `@skill: call` to K , the predicate $\text{resolves}(C, K)$ is invariant under any number of retrains of K , and no casing/punctuation variant can fork a duplicate source.* *Proof (sketch).* Retrain snapshots and soft-deletes the prior nodes, then re-inserts under the same source id, renaming only after inserts succeed, so the id a name resolves to is never reallocated (Lemma 1 re-derives the intra-engram edges idempotently); cross-engram links live in the encrypted side-table keyed by caller and name, re-resolved per-caller, leaving no dangling reference. Name normalization is pinned by a model-free suite (6/6). ■ *(Enforced by construction; the formal form of §1.1’s cross-skill-edge stability, contribution 4.)**

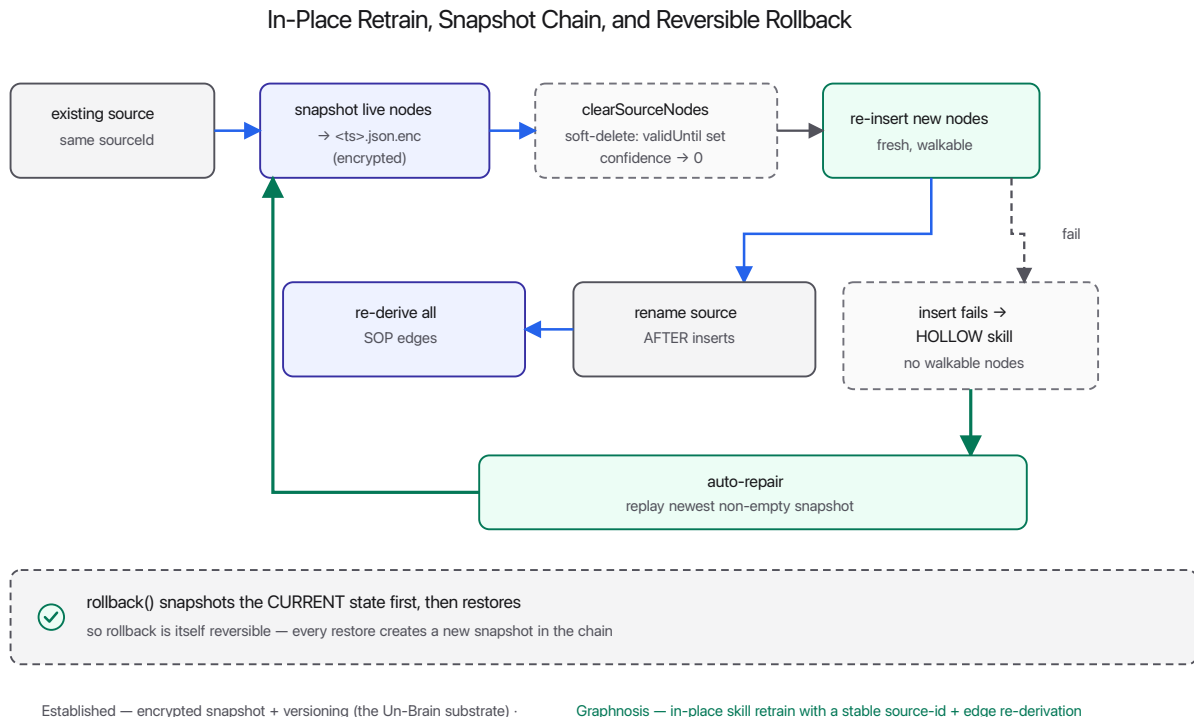


Figure 6. In-place retrain and reversible rollback. A retrain snapshots the live nodes to an encrypted file, soft-deletes them, re-inserts new ones, renames the source *after* inserts succeed, and re-derives all edges; a crash leaves a hollow skill that auto-repairs by replaying the newest non-empty snapshot. Rollback snapshots the current state first, so it is itself reversible (§6).

The closed staleness loop. A Skill records the ids of the memory nodes its recall recipes cite. When the owner edits, forgets, or supersedes one of those nodes, the dependent skill is enqueued for retraining with a

typed reason (*source-edited*, *superseded*, *forgotten*), and a periodic sweep audits cited-node drift directly. An idle background scheduler drains the queue conservatively — at most one retrain per cycle and (by default) surfaced for the owner to confirm rather than applied silently. Drift detection and enqueueing are deterministic and license-free; the retrain that the owner then confirms is the step that consults the **skill-training** gate, so an unlicensed owner still *sees* every staleness signal even though applying the fix in-app requires the license. Freshness is thus maintained the same way the substrate maintains memory: deterministically, auditably, and under the owner’s hand.

7. Dispatch: The Right Skill at the Right Moment

A library of dozens of procedures is only useful if the right one surfaces at the right moment without the rest crowding the context window. Dispatch is the mechanism, and it is also the headline scaling argument.

Lazy loading. The dispatch router never inlines the library. Step one loads only an *index* — a compact one-line **trigger** → **skill** entry per route — at roughly 160 characters per skill in the conservative measurement (a one-line entry carrying the source id; $\approx 3,034$ tokens over the 76 skills, §10.2). Step two classifies the current context against that index’s trigger table. Step three hydrates the full text of only the matched skills — one to three in the scenarios we model (not measured session telemetry). A session that matches two skills loads two skills, not the whole library; we quantify the resulting token reduction in §10.2.

Proactive matching. Beyond explicit invocation, a background watcher proposes skills from context through a three-pass match — due or upcoming obligations, keyword overlap with recently ingested material, and time-based cadence rules — firing only above a score threshold. Anti-spam bounds keep it from nagging: a small per-session cap on new proposals (five), a multi-hour suppression window per (signal, skill) pair (six hours), and exclusion of meta-skills that operate on the skill system itself. A skill *declares* its auto-proposal safety through the `[dispatch-safe: yes|partial|no]` tag on its Trigger goal (§3). The proactive watcher now runs every matched skill through the autonomy gate (§11, §13), which reads this tag and caps a **dispatch-safe: no** or low-confidence match at *suggest*; a meta-skill blacklist additionally keeps self-referential skills out of the proactive channel.

One trigger table, several consumers. The same trigger table is the single matching authority for the proactive watcher, for implicit routing inside a chat, and for *export*: the table is rendered into the host AI client’s own configuration — a `CLAUDE.local.md` block, a `.cursorrules` file, Cursor `.mdc` rules — so the skills register themselves with whatever agent is driving. The library, in other words, advertises itself to its executor rather than waiting to be asked.

8. Case Study: The System That Built Itself

The corpus in this paper was not authored as a demo. It was trained, by an AI coding agent (Claude Code) reading the Graphnosis repository together with the founder’s plans and prior sessions, to make the ongoing work of building Graphnosis cheaper and more reliable — and it was then used to do exactly that. We present this as a *motivating case study*, reconstructed after the fact from commit history and session transcripts rather than from instrumented telemetry; its figures are accordingly **excluded from the paper’s empirical claims** (§10) and should be read as a guiding anecdote, not as measured evidence. We present its failures alongside its wins for the same reason.

The motivating baseline. An early build push run *without* skills — many parallel sub-agents coordinating from a monolithic instruction file — exhibited the failure mode the kit was built to remove: workers stuck on stale assumptions, repeated rework loops, and a large, diffuse token bill spread over days. (These figures are reconstructed from commit history and session transcripts, not instrumented telemetry, and we label them as estimates throughout.)

Two derivation waves. The corpus accreted in two dated waves, visible directly in the skills’ training timestamps. (The methodology itself shipped earlier, in May 2026 — the trained, goal-structured, cortex-

bound `train_skill` compile on 2026-05-27 and the finite-step `walk` layer with the full eight-goal contract on 2026-05-31.) A first wave (mid-June, lower recall-breadth settings) crystallized roughly fifty foundational skills (an approximate, anecdotal wave size, excluded from the empirical claims) — several distilled from real debugging sessions, so the SOP encodes what actually worked rather than what the documentation claimed. A second wave (a week later, higher recall-breadth) retrained and repaired the first and added skills born from concrete incidents in shipping and compliance work. The result is the 76-skill library measured here.

The self-modifying loop. What makes the story more than “someone wrote good prompts” is that the corpus improves itself, on-device, through skills that operate on the skill system: a mistake triggers a `retrospective-learning` walk, which proposes a new or amended procedure via `adaptive-skill-creation`, which compiles it with `train_skill`, which retrains `skill-dispatch` so the new skill self-registers into the router (§7). Indexed work trains the skills; the skills then govern the next session’s work — a closed learning loop that, under local-only routing, never leaves the machine.

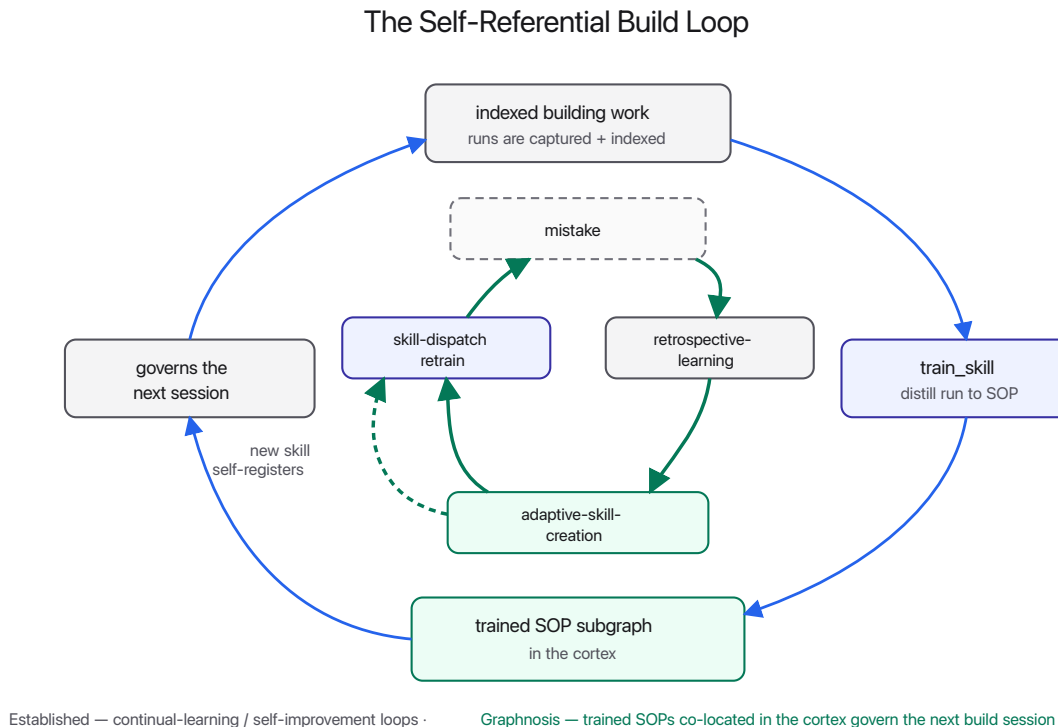


Figure 7. The self-referential build loop. Indexed work trains skills that govern the next session; an inner loop turns each mistake into an amended procedure (`retrospective-learning` → `adaptive-skill-creation` → dispatch retrain) that self-registers — a closed learning loop that, under local-only routing, stays on-device (§8).

Why it saves tokens. The savings mechanism is structural, not rhetorical. Lazy dispatch (§7) keeps the library out of the prompt and per-step routing (§5, §10) keeps cheap work on cheap models — both deterministic-accounting reductions quantified in §10 — while the empty-engram contract (§4) keeps personal-memory bloat out of the skill bodies, a structural property whose effect shows up as a before/after observation (Appendix A) rather than a token figure. The case-study token bands here, by contrast, are reconstructed estimates and carry no such accounting.

The counterexample. The gains are contingent on actually *walking* the dispatched skills. A later firefight in which skills were *instructed but not walked* at the orchestrating level regressed to the no-skills baseline — multiple rework loops, a net-negative verdict in the founder’s own retrospective. We keep this in the paper because it sharpens the claim: the benefit comes from executing the procedure graph, not from a skill library merely existing in the cortex.

WITH vs WITHOUT Skills: The Build Ledger

case study — token bill per build state (lower is better)

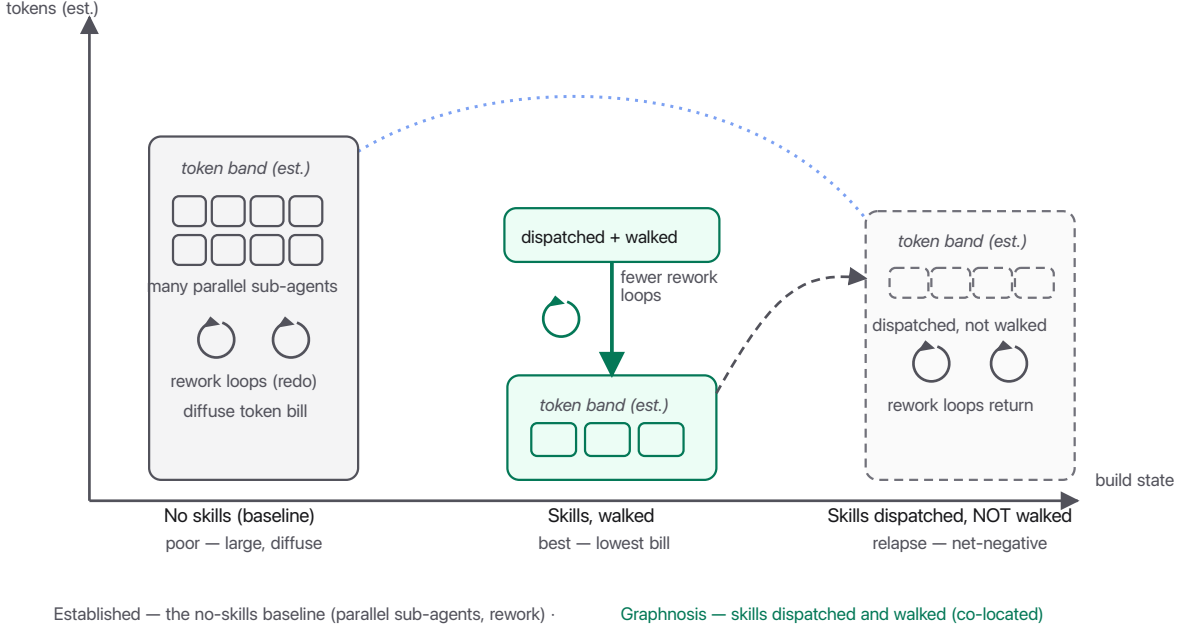


Figure 8. The build ledger (token bands reconstructed from commit history and session transcripts; estimates, not instrumented telemetry, and excluded from the paper’s empirical claims — see §10). Building without skills incurred parallel-worker sprawl and rework loops; skills *walked* gave the lowest cost; skills *dispatched but not walked* relapsed to net-negative — the benefit comes from executing the procedure, not from its mere presence (§8). The bands illustrate the qualitative ordering this case study motivates, not a measured magnitude.

9. The Trained Corpus: Generality Across Domains and Completion Oracles

If Skills were only good for code, “procedure as subgraph” would be a narrow trick. The corpus shows otherwise: the same substrate carries engineering SOPs, business/go-to-market SOPs, and self-referential meta-skills, distinguished cleanly by their completion oracle.

Three families. Twenty-six **code/engineering** skills close on a machine check — [verify: tool] or [verify: state] (run the smoke test; confirm the build artifact is fresh). Twenty-five **business/GTM** skills close on human judgment — [verify: human] — and open by recalling from a private, topic-scoped engram. Twenty-four **meta/agent** skills operate on the system itself (dispatch, retrain queues, retrospectives, sub-agent delegation). That is seventy-five in the three core families as catalogued today; the corpus measured in §10 counted seventy-six, the difference being a single skill — **research-publishing-workflow** — recategorized into the Research family during the per-family migration, so the grand total is unchanged (Appendix A). The split is not cosmetic: the oracle tells an executor *how to know it is done*, and it differs by kind of work.

Eleven domains, one method. The three families above are the self-built corpus measured in §10. As a representational domain-compatibility probe — whether the substrate is *domain*-agnostic and not merely *task*-agnostic — we then trained eight further families by the identical empty-engram method — **research** (literature-review, citation-audit, experiment-design, research-publishing), **creative**

(composition, release, arrangement-review, diagram-layout), **nonprofit/board** (board-meeting-prep, grant-proposal-drafting, fundraising-campaign-plan), **heritage/genealogy** (source-triage, lineage-verification, archive-ingestion), **event organizing** (event-concept-and-budget, event-public-launch-campaign, sponsor-and-vendor-negotiation), **film production** (script-development, production-greenlight-package, teaser-launch-campaign), **acting** (role-breakdown-and-prep, self-tape-audition-workflow, public-persona-and-promo), and **reviewing** (manuscript-review-protocol, publication-readiness-audit, peer-review-response) — extending the library from 76 to **101 skills across eleven domains** (the eight new families add 26 skills — Research and Creative four each, the other six three apiece — atop the 75-skill core, after **research-publishing-workflow** was recategorized into Research). A twelfth family followed — the **Ghampus Skills**, the chat agent’s own operating procedures (§9.1) — bringing the corpus to **108 skills across twelve families**. Nothing about the method changed: each new skill is the same typed-edge subgraph with the same eight goal categories, trained with cortex recall disabled, binding private facts only at walk time. The new domains close predominantly on **[verify: human]** — they are judgment work — and so join the human-oracle family on the oracle axis even as they widen the domain axis. The dispatch and vitality measurements (§10.2, §10.5) report the 76-skill state at the time of measurement; a larger library tends to *strengthen* the lazy-dispatch ratio, since a session still hydrates only the one-to-three skills it matches regardless of how many it does not. Two axes of generality are thus distinct and both represented in the corpus: it is *domain-agnostic* (eleven domains by one method) and *oracle-agnostic* (procedures close on a tool, state, or human completion oracle), with the completion oracle — not the domain — the primitive that classifies kinds of procedural work.

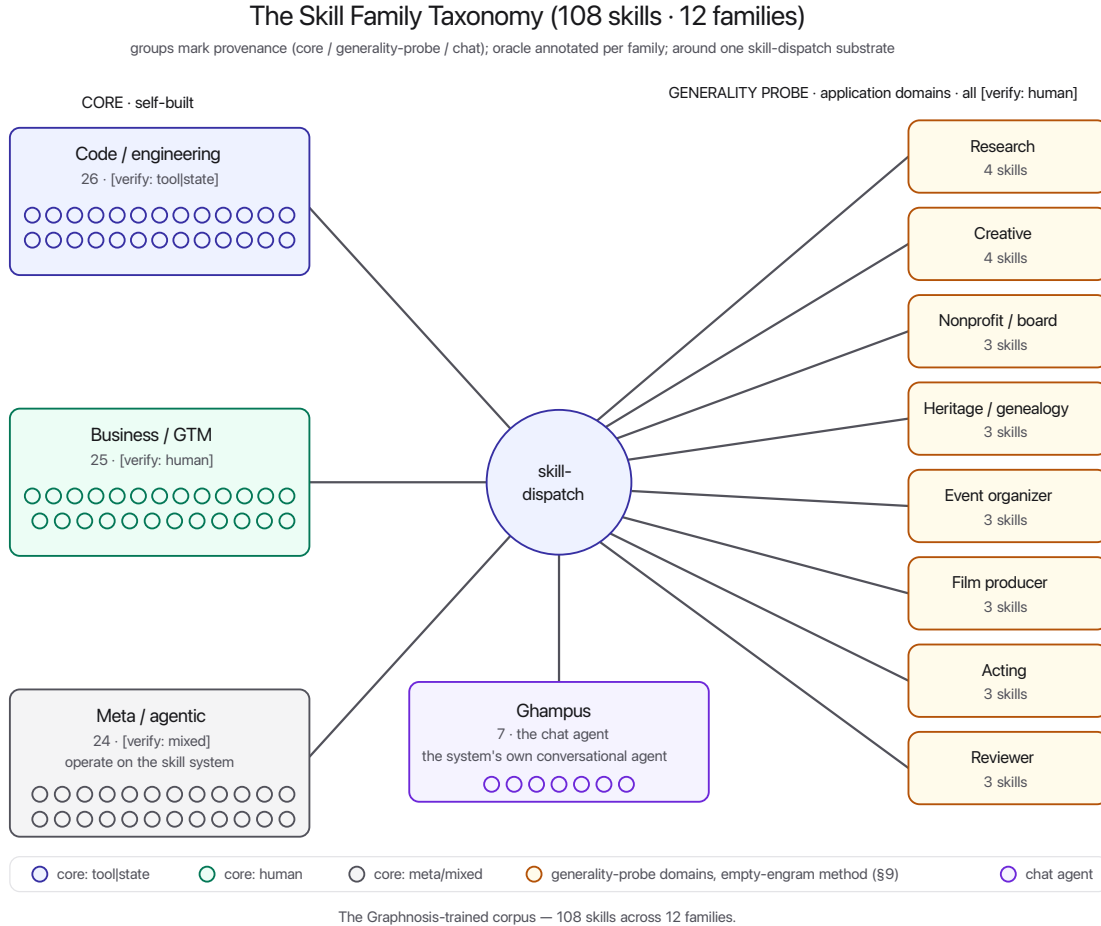


Figure S1. The corpus across completion-oracle families — code/engineering (26, [verify: tool|state]), business/GTM (25, [verify: human]), and meta/agentic (24) in the self-built core (76 as measured in §10; **research-publishing-workflow** was later recategorized into Research, leaving 75 in the three core families)

— joined at a **skill-dispatch** hub, with eight further application domains (research, creative, nonprofit, heritage, event organizing, film production, acting, reviewing) and the chat-agent family (Ghampus) trained by the same method, extending the library to 108 skills across twelve families (§9, §9.1). Two orthogonal axes structure the corpus: domain (the families) and completion oracle (annotated per family); the colour groups mark provenance (core-split / probe / chat-agent).

Privacy rides at the binding, not the body. A business skill keeps its SOP generic and its memory private by binding to engrams only at walk time. **fundraising-readiness**, for instance, recalls from a sensitive **investors** engram — which fires a consent gate and forces local-only routing — while the procedure text itself contains no figures and is freely shareable in principle, after the owner’s review of the authored text (§13). This is the §4 invariant paying off in practice: the same skill is portable *and* discreet.

A cross-skill call graph. Skills compose. **skill-dispatch** sits at the hub; a code cluster forms around a **bug-investigation runtime-diagnosis** loop and a **ship-workflow→generated-artifact-freshness-check** chain, and a business cluster around **go-to-market-planning enterprise-sales-prep→enterprise-gtm-compliance-ang** the two joined at the top by dispatch. The call edges are the **skill:calls** relations of §3, with **bug-investigation** recurring as a common *On failure* handler across the code family.

9.1 The Engram Is the Agent: Agempi on a Shared cortex

The taxonomy above invites a reframe that the rest of this paper has been building toward without yet naming it. A trained skill is not a passive document the runtime consults; it is one procedure inside a larger, durable, owner-held object — the per-family skill engram that is the core of a *domain agent*. We name that object an **Agempus** (AGENT + hippoCAMPUS; plural **Agempi**) — a **Graphnosis agent**, the term we use for it in running prose — and define it precisely, in the primitives already established: an Agempus is the engram (a family of skills together with their bound memory, dispatch index, and call graph), not any single skill.

Definition 3 (Agempus). An Agempus is a skill engram: the per-family bundle of (i) its **procedures** — the typed-edge step subgraphs of §3, with their **skill:seq/skill:loop/skill:branch** control flow; (ii) the **private memory they bind at walk time** through recall recipes (§4, §5), never frozen into the body; (iii) the **dispatch index** that surfaces it on a matching context (§7); and (iv) the **cross-skill call graph** that lets it invoke siblings (§3, the **skill:calls** relations and the encrypted cross-gram side-table). These four are co-located in one engram and held by the owner. An Agempus is therefore the *persistent, model-agnostic core* of a domain agent: the model is rented and swappable — the per-step router (§5) picks a different one for each step and a different one again next quarter — while the Agempus is owned and durable.

In the neuroscience lens of §15, an Agempus is the brain’s task-control loop made durable and owned: the Agempi **share one cortex**, and each **has a hippocampus** (its recall), **a basal ganglia** (its skill set — each skill a *striosome*, a discrete procedure compartment), and **rents a prefrontal cortex** (the per-step model, §5). The name keeps the hippocampal root because what is distinctive is *owning the memory*, not selecting the action — every agent selects actions; only this one carries its own cortex.

This is the inversion that distinguishes the design from a prompt library or a *separate-store* workflow engine (§12): the agent’s identity does not live in a model checkpoint or a system prompt, it lives in an engram on the substrate. What the owner keeps when the frontier model changes is exactly the Agempus.

The relationships, named. Three terms already in use lock together under this frame. The **cortex** is the one owner-held memory graph (the substrate of the companion paper) that every skill and every fact shares. The **Ghampus** is the chat Agempus — itself an agent whose domain is conversation, with its own family of operating skills (the Ghampus Skills, §9). Its routing is driven by **ghampus-operator** — catalogued in the meta/agent family because it operates the system itself (§9), and the §10.3 measurement example — distinct from the Ghampus Skills family it dispatches over. It fields the user and *dispatches*. The **Agempi** are the domain specialists it dispatches to: **bug-investigation**, **go-to-market-planning**, **fundraising-readiness**, **lineage-verification**, one per family of work (§9). Ghampus is the front of house; the Agempi are the roster behind it; all of them read and bind the single cortex beneath. The

Graphnosis Autonomous Praxis is the autonomous pipeline that employs and trains the Agempi — the same training compile of §4 driven by the vitality and governed-retrain loop of §6, so the roster stays current without the owner re-authoring it by hand. A cross-engram `@skill: call` (§3) is, read this way, one domain agent borrowing a skill from another — the roster collaborating, not a monolith branching — and because the callee shares the cortex, that borrow avoids the sub-agent warm-up entirely: the **live** single run **measured** ≈ 475 tokens for the in-cortex walk-and-recall against $\approx 21.3\text{k}$ for a full cold spawn ($\approx 45\times$), corroborated by a reproducible $N=5$ sandbox (§10.6, §10.7), because the callee need not re-learn the context.

The caveat, stated exactly — no overclaim. An Agempus is the agent’s *core*, not a running loop. It does not act on its own; it acts when a runtime *walks* it (§5), and how far that walk proceeds without a human is governed by the L0–L3 autonomy axis of §13 — a default the engram sets, overridable per skill, and in every case capped by the skill’s authored `[dispatch-safe:]` safety. A `dispatch-safe: yes` Agempus with a high-confidence trigger and uncontradicted recalled memory is *permitted* to walk end-to-end at L3 — “end-to-end at L3” naming the contract/permission ceiling, not a demonstrated write-capable autonomous run, since today’s reference walker (§11) runs read-and-compute only and does not dispatch `parallel[]` siblings (capped loops it re-executes with the cap enforced in code; an unattended run admits a loop only with an authored cap, §11); anything less resolves down to preview-and-confirm or a passive suggestion. An *unattended* executor that walks an auto-eligible Agempus with no human in the loop is now implemented (§11, §13) but **off by default**: it runs only under an explicit owner opt-in, gated by the same per-skill authored safety, with an encrypted per-action audit and per-action undo. Left off — the default — an Agempus acts only when a runtime walks it. Calling an engram an agent is a claim about its *structure and ownership*, not a claim that it loops autonomously unless the owner turns that on.

Why the reframe earns its keep. Three threads of this paper resolve once the engram is read as the agent. First, the corpus (§9) is not a flat library but a **roster of specialists** — each family an Agempus with its own completion oracle, the whole connected through the `skill-dispatch` hub and the cross-skill call graph. Second, **distribution** (§13) becomes portable agency: a signed `.gsk` pack is not a bundle of prompts but a *signed SOP pack designed for future cross-owner portability* — its procedures, dispatch index, and recall recipes travel together, designed to bind a recipient’s own cortex at walk time (cross-owner import unevaluated, §13). Third, it sharpens the **unit-of-reuse** argument of §13: when the model is rented and the Agempus is owned, the unit of competitive advantage stops being the prompt and becomes the private, governed, ever-improving Agempus you hold — a domain agent whose core outlives every model it is walked by.

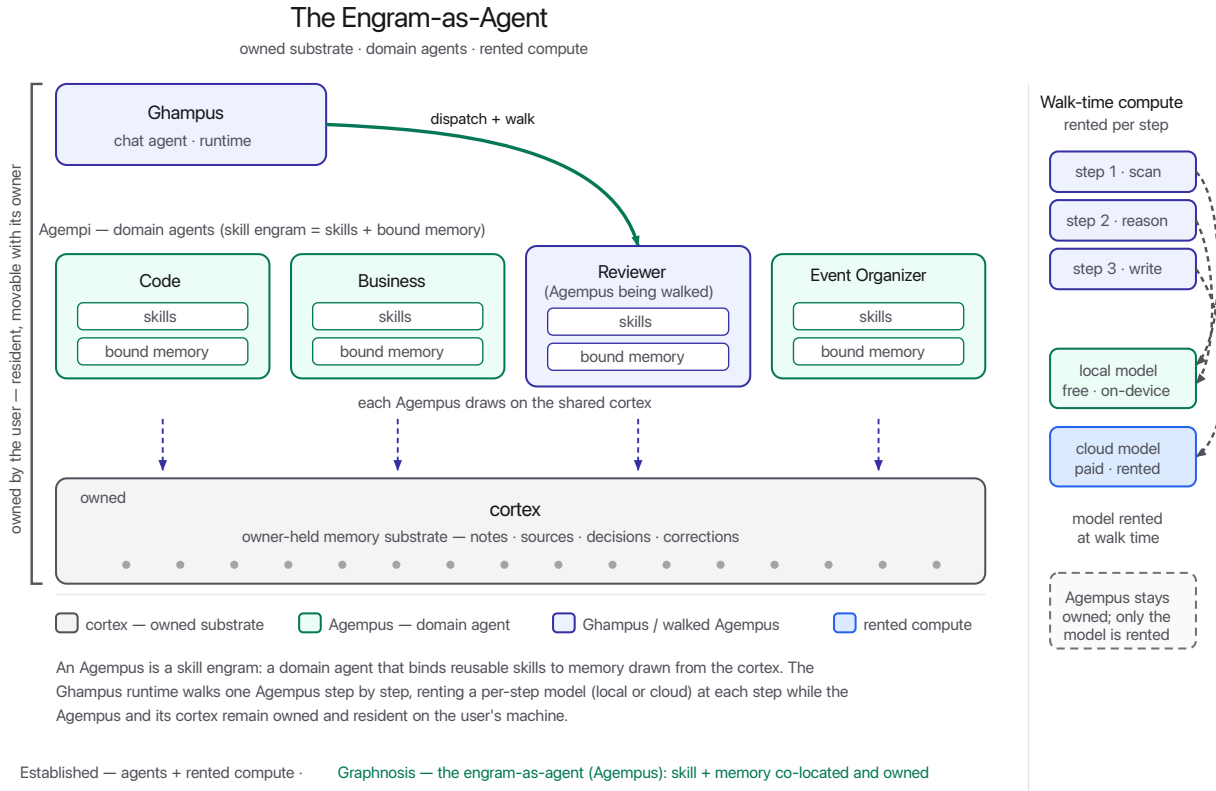


Figure S2. The engram-as-agent. An owner-held cortex (memory substrate) feeds several Agempi — domain agents that each bind reusable skills to cortex-drawn memory — while the Ghampus runtime dispatches and walks one Agempus step by step, renting a per-step model (local or cloud) at walk time as the Agempus itself stays owned and resident.

The procedures read like hard-won craft, because they are. *ship-workflow* insists: *always show git status and git diff --stat before staging*. The changelog rule reasons from consequences: *describe the change from the user's perspective — the git log is public*. *bug-investigation* encodes a bounded retry directly in its DSL: *@loop: 5 max=4 if the smoke test still fails*. These are not generic best practices; they are this project's accumulated procedure, compiled into a form an agent can walk.

One caveat on the corpus. Per-step routing (§5) is a real engine capability, but the *original* 76 skills largely under-exercised it: most carried no **@needs** annotations and defaulted to a general capability, so the dramatic routing savings of \$10.3 were a property the engine made *available* more than one the measured library fully exploited. That gap is now closed corpus-wide: the shipped fixture shows the retrofit reached the *original* corpus too — 71 of the original 76 were also **@needs**-retrofitted, so 100 of 108 skills carry the status **retrained** (the eight remainder already-tagged or excluded). Across the thirty-two new family skills (the eight new domains plus Ghampus; the migrated **research-publishing-workflow** excluded), twenty-nine were given explicit per-step **@needs** in this pass (status **retrained**) and the three Film Producer skills already carried them (**skipped-already-tagged**), so all thirty-two now route per step. This corpus-wide retrofit — not the router-side inference pass — is what the E2 routing numbers (§10.7) are computed over; a router-side inference pass (§11) is a **fallback** that assigns a best-fit capability to any *future* step that still declares none, so an untagged step is right-sized rather than collapsing onto the smallest model, but it is not the mechanism behind the measured E2 figures. The goal-completeness picture also improved with the maintenance pass of this work: five thin skills were completed to the full eight-category contract, and the new families were authored complete — even though, per §6, vitality does not score completeness.

10. Evaluation

Following the substrate paper’s discipline, we hold the answer/agent model fixed and vary one feature of the procedural layer at a time, and we separate exact measurements from labeled estimates.

10.1 Setup

There is no public “SOP execution” benchmark, so — as the companion paper does for contradiction triage — we define measurements that isolate the procedural layer rather than the model running it: a token-cost decomposition of dispatch, a per-step routing-savings accounting driven by the real router, and a deterministic, model-free correctness check of the defining primitive. Where a figure depends on the user’s encrypted run history we say so and substitute a reproducible proxy.

10.2 Lazy dispatch vs library inlining (token decomposition)

The dispatch claim (§7) is that a session loads a fraction of the library. Measuring all 76 skills with the system’s own tokenization heuristic (characters $\div$ 4): inlining every skill body costs **48,780 tokens**; a lean dispatch index (a one-line **trigger** $\rightarrow$ **skill** entry per skill — conservatively counted here including the source id) costs $\approx$ **3,034 tokens**; the median skill body is 550 tokens (mean 642; the heaviest, **skill-dispatch** itself, is 1,751 — and that body is the truncated/corrupted one excluded from the E2 routing subset (§10.7), so 1,751 is a *floor* on its true length and 48,780 is correspondingly a floor on the inline total — a direction-safe understatement for the reduction ratio). A typical session that hydrates one or two matched skills therefore costs $\approx$ **3,600–4,100 tokens** — a **91.5–92.6% reduction**, i.e. a dispatched session loads roughly **8%** of the library. The figure is conservative (the index measure overcounts by including the source id, which the live trigger table omits) and, being a ratio of two character- $\div$ -4 counts, robust to the coarseness of the heuristic.

Proposition 3 (Lazy-dispatch footprint bound). *Let a library of N skills have per-skill index cost idx_i (one line) and body cost body_j . A session matching k skills (typically $1 \leq k \leq 3$) loads $F(N, k) = \sum_{i=1}^N \text{idx}_i + \sum_{j \in \mathcal{M}} \text{body}_j$ against full inlining $\sum_{j=1}^N \text{body}_j$. The matched-body term depends only on k , not on N ; hence as N grows at fixed k , inlining grows $\sim$ linearly in N while F grows only at the far cheaper index rate, so, provided added skills’ index-to-body ratio stays below the current footprint ratio and the added skills go unmatched (matched k fixed), the reduction $1 - F / \sum_j \text{body}_j$ is non-decreasing in N — a larger library then tends to strengthen the ratio. *Proof.* F ’s body term is constant in N , so F grows only at the index rate while inlining grows at the body rate. Formally, for additions that go *unmatched* (leaving k and the matched-body term fixed), the reduction $1 - F / \sum_j \text{body}_j$ is non-decreasing in N whenever each added skill’s index-to-body ratio stays at or below the current footprint ratio $F / \sum_j \text{body}_j$; an addition that *joins* the matched set adds its body to F and can lower the reduction, so the monotonicity is stated for unmatched growth — an empirical scaling argument over this corpus, not a closed-form guarantee. ■ (Empirical for $N = 76$: inlining = 48,780, index $\approx$ 3,034, a $k \in \{1, 2\}$ session $\approx$ 3,600–4,100 tokens — a 91.5–92.6% reduction, $\approx$ 8% of the library; the monotonicity-in- N holds whenever an index line costs far less than a body.)*

Lazy Dispatch vs Library Inlining

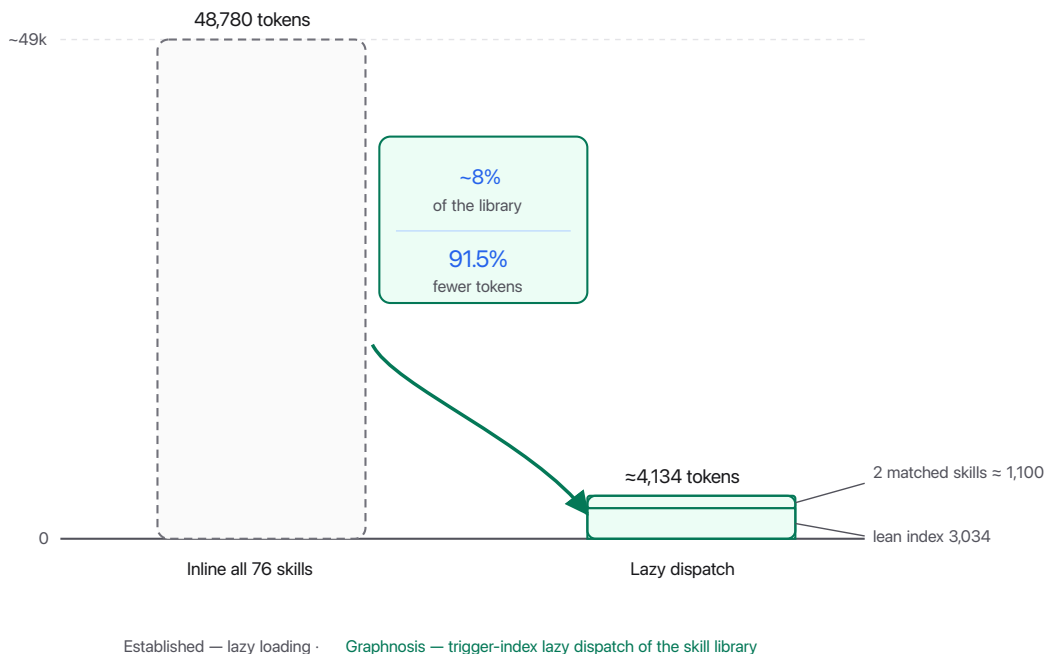


Figure 9. Lazy dispatch versus library inlining. Inlining all 76 skill bodies costs 48,780 tokens; a dispatched session loads a lean **trigger** → **skill** index ($\approx 3,034$ tokens) plus only the one or two matched skills ($\approx 3,600$ – $4,100$ tokens total) — roughly 8% of the library, a 91.5–92.6% reduction (§10.2).

10.3 Per-step routing savings (measurement-time accounting)

Driving the **real router** (`planSkillWalk` over the static model catalog) across an annotated nine-step skill (`ghampus-operator`), with a per-step assumption of 1,500 input / 600 output tokens and the system’s own Sonnet-4.6 counterfactual baseline (\$3 / \$15 per 1M tokens, \$0.0135 per step):

Configuration	Routing outcome	Walk cost vs \$0.1215 baseline	Saved
Adaptive, local + cloud	8/9 steps → free local (Qwen 2.5 7B, Llama 3.2 1B/3B); 1 step (writing) → GPT-4o mini	\$0.0006	99.5%
Always-best, cloud only	all 9 → Claude Haiku 4.5	\$0.0324	73.3%
Local-only	8/9 free; writing step has no qualifying model	<i>infeasible</i>	—

Catalog note (required for reproduction). These three rows reflect the **measurement-time** model catalog, in which no local model declared the **writing** capability — hence the one paid **writing** step and the *infeasible* local-only row. The catalog **shipped and pinned with this paper** (`route-b-artifacts/lib/model-registry.ts`, sha256 in `CHECKSUMS.txt`) has **since added local writing** support, so re-running `planSkillWalk` over the frozen catalog routes **all nine steps local** ($\approx 100\%$ saved, local-only *feasible*). We keep the 99.5% figure as the measurement-time result and flag that it is **not** reproducible from the shipped catalog without restoring the earlier catalog state; the corpus-wide

cost accounting of §10.7 (E2), by contrast, *is* run against the shipped catalog. Figure 4 shows the measurement-time routing. (The 73.3% coverage-max saving below is a property of the *pinned* catalog — Haiku 4.5 pinned \$0.80/\$4.00, Sonnet 4.6 \$3/\$15, price ratio 0.2667. At current retail (Haiku 4.5 \$1/\$5) the same identity yields 66.7%; the identity structure — saving = 1 − price-ratio — is the claim, not the constant.)

Two results are worth drawing out. First, the *premium* **always-best** policy’s **73.3%** is a **pricing identity, not a corpus property**: **always-best** routes every step to Haiku (coverage-max), and the saving is the complement of the Haiku/Sonnet price ratio on the pinned catalog, $1 - 0.2667 = 73.3\%$ on both token axes regardless of the skill. The cheaper-by-design *cheapest-qualifying* rule (Definition 2, the adaptive default) instead reaches Gemini Flash / GPT-4o-mini and saves far more — ~96.6% cloud-only corpus-wide (§10.7). Second, the 73.3% coverage-max percentage is **invariant to the per-step token assumption**, so that figure is robust (the adaptive _{96.6%}99.6% rows are invariant only to *uniform* rescaling of the assumption, since they mix models at different price points); only the absolute dollar scales. This is the real selection function over the real catalog — but a *cost* figure only; output quality by capability is the separate study of §10.7.

10.4 The defining primitive, checked deterministically

Several of the formal claims of §1.1 are pinned by a small, model-free regression suite — the procedural analog of the companion paper’s deterministic contradiction confusion matrix; the rest (e.g. Theorem 2 termination, Invariant 1 empty-engram, and the autonomy gating of §13) are by-construction, constant-gated, or tested in their own modules rather than in this suite:

- **Plan-compilation determinism** (Theorem 1, §5) — a golden test serializes a compiled plan and asserts byte-equality against a stored fixture (7/7 assertions passing).
- **Structure preservation** (Proposition 2, §4) — the token-extraction validator is checked to (a) pass a prose-only rewrite that keeps every token and (b) reject rewrites that drop a **@needs** routing tag or an **only_engrams** binding (9/9 assertions passing).
- **Cross-skill-edge stability** (§6) — the in-place name match is checked to treat slug and title-case forms as one skill while keeping genuinely distinct names apart (6/6 assertions passing).
- **Privacy hard-lock** (Invariant 2, §5) — a step recalling a sensitive engram is asserted to route to a local model even with all cloud providers enabled, plus a control case proving the tier-map wiring is load-bearing (17/17 assertions passing).
- **Edge-derivation idempotence** (Lemma 1, §3) — re-deriving a capped loop’s edges (**@loop: 2 max=4 → max=5**) leaves exactly one loop edge, not stale duplicates; verified red before the fix (**skill-loop-idempotence**, 14/14 assertions passing).

These are existence proofs that the guarantees hold by construction, not accuracy benchmarks; their value is that a regression would surface the moment any of the five properties broke. On availability, stated plainly: **two test sources — the privacy hard-lock (17/17) and edge-derivation idempotence (14/14) — are included for inspection in route-b-artifacts/tests/, but they import sidecar modules and are not standalone-runnable; the other three (plan-compilation determinism, structure preservation, cross-skill-edge stability) are not shipped. All five pass counts are internal sidecar CI records, not standalone artifact checks** a reader can re-run against the published artifact; the sidecar suite runs internally via `pnpm exec tsx tests/run-all.ts`.

Procedural Correctness, Checked Deterministically

SUITE	RESULT
 Plan-compilation determinism golden plan	7 / 7
 Structure preservation incl. @needs & only_engrams	9 / 9
 Cross-skill name match slug / title-case name normalization	6 / 6
 Privacy hard-lock (Invariant 2) sensitive-engram steps forced local	17 / 17
 Edge-derivation idempotence (Lemma 1) re-derivation reaches a fixpoint	14 / 14

model-free checks

a regression surfaces the instant a property breaks

Established — regression / golden testing · Graphnosis — the five by-construction guarantees these suites pin

Figure 10. The defining primitive, checked deterministically. Five model-free regression suites pin the formal claims: plan-compilation determinism (7/7), structure preservation including @needs/only_engrams (9/9), cross-skill name matching (6/6), the privacy hard-lock — a sensitive-recall step forced local even with all cloud providers enabled (17/17), and edge-derivation idempotence for capped loops (14/14). A regression surfaces the instant a property breaks (§10.4).

10.5 Governed retraining moves vitality the right way

The lifecycle claim (§6) is that retraining restores freshness. Against the measured baseline (76 skills *fresh*, mean 97.7, median 98), a clean re-compile of the two lowest-scoring skills raised them **92** → **98** and **94** → **99** by resetting the age penalty and clearing the bulk of the superseded-node staleness (a small residual keeps the scores at 98/99, not a perfect 100), with the prior versions preserved as rollback snapshots. The intervention is small because the corpus was already healthy at measurement time — a snapshot, not proof the maintenance loop caused it; the causal signal is the *move* under retrain (92→98, 94→99), not the baseline level.

10.6 Cross-skill calls vs. sub-agent spawns

Because every Graphnosis agent shares one cortex, it invokes a sibling’s procedure through an in-cortex cross-skill call — @skill: name(args) -> \$capture — which composes like a function: it binds arguments into the child walk, captures the return under a named variable for downstream steps (§3, §5), and resolves the (possibly cross-engram) target through a side-table computed at training time, so the resolution itself costs no model tokens. By the paper’s characters-÷-4 heuristic (§10.2; the same tokenizer the reference

walker applies live), the marginal cost of such a call is essentially the called skill’s body — median 550, mean 642 tokens across the 76-skill corpus — plus the few-token argument binding, on the order of 700 tokens — and E4 (§10.7) **measured** the in-cortex walk-and-recall of this borrow path at **≈ 475 tokens**. The alternative, spawning a fresh sub-agent for the same sub-task, must instead pay a warm-up the shared cortex makes unnecessary: a re-sent system/role prompt, a re-description of the task, and a re-priming of context the caller already holds. Measured directly (E4, §10.7): the **live** headline run pairs a ≈ 475 -token borrow against a **$\approx 21.3\text{k-token}$** full-client cold sub-agent spawn ($\approx 45\times$), corroborated by a reproducible sandbox re-measurement (5 cold spawns $\approx 25.0\text{k}$ vs a 637-token borrow, $39.3\times$) — a **$\approx 39\text{--}45\times$ measured** separation, with the separately-quoted $\sim 30\times$ against the ~ 700 -token *modeled* typical borrow. That entire gap is the spawned agent’s warm-up, avoided when the sibling already shares the cortex. The separation is host-dependent in absolute size ($\approx 21.3\text{k}$ is a *full Claude Code* sub-agent, a harness-specific maximum), but the architectural invariant is the **constant warm-up avoided**. A caution on the accounting: the borrow tokenizer (characters $\div 4$ over walk-plan + recall) and the spawn tokenizer (real `subagent_tokens` over the full session including output) are **different**, so the borrow and spawn sides are *not* counted under the same convention — the same-convention robustness argument applies to the §10.2 dispatch ratio, not to this comparison; the separation’s robustness rests instead on its magnitude far exceeding any plausible tokenizer discrepancy.

A familiar economy of expertise. The everyday analogue is delegation between people who already know each other’s work. When you need something a colleague is expert at, you do not hire and onboard a new specialist for the afternoon; you walk over and ask the person whose skill it already is, handing them only the specifics and getting back the result. You don’t need their *entire* skillset, only the one skill that answers the question; you don’t hand over your whole context or send them digging through everything they know, because the ask is specific and pre-resolved — so the exchange is quick, and the caller draws the single named procedure, not the whole agent. Spawning a sub-agent — the prevailing way to delegate in agent systems today — is the hire-and-onboard path: a fresh worker who must be told who they are, what tools they have, and what the situation is before they can start, the whole colleague stood up and absorbed when only one of their skills was needed. A cross-skill call is the walk-over: the sibling agent already holds the procedure and already shares the cortex, so all that crosses the boundary is the arguments in and the return out. The token delta above is, almost exactly, the cost of onboarding a stranger versus asking a colleague who already knows.

On precedence — an engineering result, not a theorem. We checked the literature before claiming novelty, and we are precise about what is and is not new. Each constituent idea has prior art: function-like composition of tools *inside one agent* (CodeAct; Wang et al., 2024); invoking an agent as a callable with structured input and a returned value (the agent-as-tool pattern in current agent SDKs); and typed-edge skill graphs that compose stored procedures (recent skill-graph work such as SkillDAG and GraSP). What we did **not** find is the specific conjunction — one agent invoking another’s *learned, named* skill as a composable function with explicit argument binding and return capture, resolved through a co-located cross-engram side-table, *together with a reported per-call token comparison against sub-agent spawning*. Recent surveys of agentic skills do not yet taxonomize this as a category distinct from tool calls, handoffs, and skill libraries, which is the gap we sit in. We therefore present it as, to our knowledge, a *first reported accounting of a specific tradeoff*, not as a law: the measured separation is a property of how heavy the spawned agent’s warm-up is, not an invariant — which is exactly why we state the baseline (system prompt, task framing, context priming) so the number can be reproduced and contested. The claim is deliberately narrow and, we think, useful: *given a shared cortex, calling a sibling skill is cheaper than standing up a sibling agent, and here — measured on both sides, though under different tokenizers (§10.6) whose discrepancy is dwarfed by the $\approx 39\text{--}45\times$ gap — is by how much*.

Proposition 5 (Borrow avoids the spawn warm-up; the $O(N)$ gap is an eager-inline anti-pattern). *Borrowing skill t via an in-cortex `@skill` call costs $C_{\text{borrow}} = b_t + \alpha$ — the called body plus a constant argument-binding term — resolved through a training-time side-table at no model-token cost (§3). Two distinct comparisons follow. (1) **The architectural claim (a constant, not an asymptote):** standing up a sibling agent pays an unconditional warm-up W — a re-sent system/role prompt, task framing, context priming — that the shared cortex makes*

*unnecessary; granted the same** lazy dispatch as Graphnosis (Proposition 3), a spawn’s matched-body term is $O(1)$ in library size but it still pays the index term $\sum_i \text{idx}_i = O(N)$ that Proposition 3 attributes to the caller’s own index — so, scoping the comparison to exclude that shared index term on both sides, the genuine separation is the **constant warm-up** W , not an asymptotic one. (2) **An eager-inline anti-pattern baseline (modeled)**: a spawned agent that instead inlines its whole library pays $C_{\text{spawn}} = W + \sum_{j=1}^N b_j = \Omega(W + N\bar{b})$, growing linearly in N — but that linearity is a property of the bad baseline, not of spawning.* *Proof.* (1) C_{borrow} excludes W (context already held) and is a side-table lookup; a lazily-dispatched spawn loads only matched bodies plus the same $O(N)$ index both sides carry, so — index term aside — both sides are $O(1)$ in matched-body cost and differ by W . (2) Eager inlining adds one b_j per skill, a sum linear in N . ■ (Empirical, §10.7: the live headline run **measured** a full-client cold spawn at $\approx 21.3k$ tokens against a ≈ 475 -token in-cortex walk-and-recall — $\approx 45\times$, grounding 3/3 vs 0/3 — corroborated by a sandbox re-measurement (5 spawns $\approx 25.0k$ vs a 637-token borrow, 39.3 $\times$). So the **measured** separation is $\approx 39\text{--}45\times$ (two datapoints); the separately-quoted $\sim 30\times$ is against the ≈ 700 -token modeled* typical borrow ($b_t \approx 642 + \text{binding}$), not a measured datapoint. The warm-up W is host-dependent and measured; the per-skill slope is modeled. Distinct from Proposition 3, which bounds the caller’s footprint; this bounds the callee.)*

10.7 In-session reproduction on the live cortex (Route B)

To move three of this paper’s claims from “by construction” toward “demonstrated,” we ran five experiments (E1–E5) directly against the author’s live cortex (38 engrams, 196 trained-skill versions) over the Graphnosis MCP, plus a sixth quality-retention sanity note (E6, below), each with its success rule fixed before the run; the full protocol and raw outputs are in the companion artifact `route-b-artifacts/route-b-results.md`. **Portability proxy (E1)**: a skill trained source-only (**influential memories**: 0) into a fresh recipient engram carried the recall *recipe* but none of the recipient’s facts in its body, exported to a signed `.gsk` pack whose own footer states verbatim that “personal memories are not included” (the paper’s gloss: no *automatically-recalled* personal memory travels; the footer itself further warns that hand-authored personal or proprietary content may still appear in the trained text and to review before sharing), and at walk time bound the recipient’s own deployment fact (similarity 0.83) — a same-machine **binding proxy** for the cross-owner *portability* scoped as future work (§13), not a headline result: export inspected, retrained into a recipient engram, then bound and walked. (The `.gsk` pack was produced and inspected but **not imported** — no one-call `.gsk` import path existed at the time of the run, so transport was shown via a source-grounded retrain into the recipient engram, not a true import; a quarantine-scoped `.gsk` import has since shipped (§11) but E1 was not re-run through it, and an independent third-party device remains the open case.) **Routing (E2, a cost accounting — no quality measured)**: over the corpus’s real `@needs` tags we priced each step’s routed model against the all-Sonnet-4.6 baseline. Under the realistic **adaptive** policy $\sim 99.6\%$ of cost is saved with every *included* skill routable (105/105); **cloud-only** is **96.6%** under the paper’s cheapest-qualifying rule (Definition 2 — it reaches Gemini Flash / GPT-4o-mini) and **73.3%** under coverage-max — and that 73.3% is the complement ($1 - 0.2667$) of the Haiku/Sonnet price ratio on the pinned catalog, a pricing identity invariant to corpus size, not a distribution. Local routing saves 98.7% on cost but only **90.5% of skills (95/105) run fully local**: **tone-match/cited** are declared by no local model, so 10 skills need cloud for at least one step. These are cost figures only — whether a 4-bit local 7B matches Sonnet’s *output* is unmeasured and is the quality-retention study below. (An earlier pass reported “100% local / 73.3% corpus distribution”; that conflated a constant price ratio with a distribution and drew its 100% from a fixture that had recoded the local-unserveable tags — corrected here.) The step-capability mix over the 809 measured steps is reasoning- and writing-heavy — 51% **reasoning**, 23% **fast**, 11% **writing**, 8% **structured-output**, then **summarization**, **code**, **tone-match** (1.6%), **extraction** — so the capabilities E6 finds local quality *collapses* on (reasoning, writing) are exactly the ones that dominate the corpus, which raises rather than lowers the unmeasured-quality risk of local routing. Stated exactly, so the adaptive figure is not over-read: the **$\sim 99.6\%$ adaptive saving is achieved precisely by routing those judgment steps (reasoning $\sim 51\%$, writing $\sim 11\%$) to free local models — the very steps E6 finds local quality collapses on** — so realizing that saving *without* quality loss is **not** established. Keeping the judgment steps on cloud, where quality holds, would *lower* the saving toward the $\sim 96.6\%$ cloud-

only figure; the 99.6%/96.6% gap is thus the price of the quality risk, not a free lunch. **Vitality (E3):** a read-only sweep returned sampled vitality scores of 96–100 with an attributable penalty decomposition (a 20-day-old skill at 96 = age −3, superseded-node −1), generalizing §10.5’s single recovery to a mechanism with named terms. **Borrowability (E4) — the paper’s headline measured result:** the **live** headline datapoint is a single run — two real full-client cold sub-agent spawns cost $\approx 21.3\text{k}$ tokens each (21,406 and 21,186), grounding 0/3 of the task’s facts (memory-ablated by instruction; the 0/3 is by construction, see E5), against ≈ 475 tokens (measured) for the in-cortex walk-and-recall that grounded 3/3 — a $\approx 45\times$ live separation, entirely the spawned agent’s warm-up, avoided when the sibling already shares the cortex. “Single-run” scopes to this live arm: the spawn figure is from **two single-shot** live-cortex spawns, not a repeated $N>1$ series. A **reproducible-record sandbox re-measurement corroborates** it (it does not replace the live demo): the **borrow side is reproducible headless** (`route-b-artifacts/borrow-cost.ts` — deterministic body+binding accounting over 12 real skills / 118 steps, reproducing “a borrow $\approx$ one body, $O(1)$ ”), and `route-b-artifacts/borrow-spawn-runs.json` records **5 cold sub-agent spawns** at mean **25.0k tokens, sd 2**, grounded 0/3, against a **single deterministic** 637-token synthetic borrow, grounded 3/3 — a descriptive **39.3 $\times$** on that harness. The sandbox borrow’s recall searched all engrams (36 others, no matches), so it was PII-free *by outcome* — recall surfaced only the 3 sandbox facts — not PII-free by construction; it is a single deterministic walk with no CI on the ratio (only the spawn arm is replicated). Across the two datapoints the **measured** separation is $\approx 39\text{--}45\times$; $\approx 21.3\text{k}$ is a *full Claude Code* sub-agent — a harness-specific maximum, not an architectural constant — so the absolute multiple is host-dependent, and the invariant is the constant warm-up avoided. The architecturally warranted claim is a **constant warm-up separation**: borrowing skips a re-sent system prompt, task framing, and context priming the shared cortex already holds. The per-skill $O(N)$ term (Figure 12) appears *only if* the spawned agent eagerly inlines its whole library — denied the lazy dispatch of Proposition 3 — so the per-skill slope and its curve are **modeled, not measured**; the only measured quantities are the warm-up separation and one ~ 642 -token mean body (median 550). **Co-location ablation (E5):** on a fact-grounded task with the success rule frozen in advance, only the arm whose procedure was **co-located** with owner-held memory completed it (3/3 facts grounded); two **memory-ablated** arms (a prose-SOP arm and a no-memory arm, *instructed to use no memory tools*) each grounded 0/3 and correctly refused to fabricate — the 0/3 follows by construction from the ablation. This is a within-design co-location ablation that isolates the effect of co-locating procedure with owner-held memory; it is complete as an ablation and makes no comparative claim about any other system.

Quality retention — a routing-design sanity note (E6). Cost says nothing about output quality, so we ran a small internal check: 60 tasks (10 per capability) each completed by Sonnet-4.6 (baseline), Haiku-4.5 (the cheap-cloud routed model), and Qwen2.5-14B (local — *stronger* than the catalog’s declared 7B, so an upper bound on local quality), scored by a blind, position-**counterbalanced** Opus judge — candidate order rotated per task, not RNG-randomized (objective capabilities checked against references, code executed). Retention = the cheap model passes the check or ties-or-beats Sonnet ($K=10$ per capability — the cells below are **directional, not point estimates**; a prior stochastic run scored 77/42 overall):

Capability	Haiku (cloud)	Qwen-14B (local)
code	100%	100%
structured-output	60%	80%
extraction	80%	70%
reasoning	70%	0%
summarization	70%	20%
writing	70%	0%
overall	75%	45%

In this pilot the split is stark: cheap **cloud** (Haiku) retained 75% overall — at least 60% in every capability — while the **local** 14B holds only on **mechanical** capabilities (code 100%, structured-output 80%, extraction 70%) and **collapses on judgment** ones (reasoning 0%, writing 0%, summarization 20%) — and the

catalog’s actual local model is a smaller 7B, a lower floor still. Read directionally — from **10 examples per capability** under a single LLM judge — the cheap models held on mechanical steps and fell on judgment ones: the warranted reading is that *cost accounting identifies cheap candidates and any quality loss is capability-dependent*, not that local routing is validated as “safe” anywhere. That is *consistent with* per-step @needs routing (cheap/local for code and extraction, a strong model for reasoning and writing) being the right primitive rather than a single cost knob — but we state it as a **directional warning from a pilot, not an established routing law**. The threats are real: K=10 per capability gives wide uncertainty (a prior stochastic run scored 77/42 against the 75/45 here); the judge is a single LLM — blind and reference-checked, but not human-calibrated, with no inter-rater measure, and an Opus judge may favor Anthropic-style prose; Qwen-14B is *stronger* than the catalog’s declared 7B; the cheapest-qualifying cloud pick (Gemini Flash) was proxied by Haiku and left untested; the 60-task battery was itself authored by models in the same harness (a correlated-design threat); and no end-to-end routed skill *walk* was quality-scored — these are per-capability completions, not full skills. Two record-integrity issues we disclose rather than re-run. First, the local-arm relay **materialized in 7 of the 60 Qwen records** — extraction-2, reasoning-7/8/9, and writing-0/1/2 (GEN_FAILED / relay-status / scaffold-contaminated outputs) — all of which were scored as Qwen failures; excluding those seven leaves the collapse **direction unchanged** (local still falls on reasoning and writing). Second, the judge was **not blind on 3 records** — writing-0, extraction-2, and reasoning-9 leaked the model or local-runtime name (Qwen2.5-14B on writing-0/extraction-2; Ollama on reasoning-9) into the candidate text — and all three were scored failures on content grounds, not on the leaked identity. A human-rated study across more local models and quantizations is possible future work, but no claim in this paper depends on it.

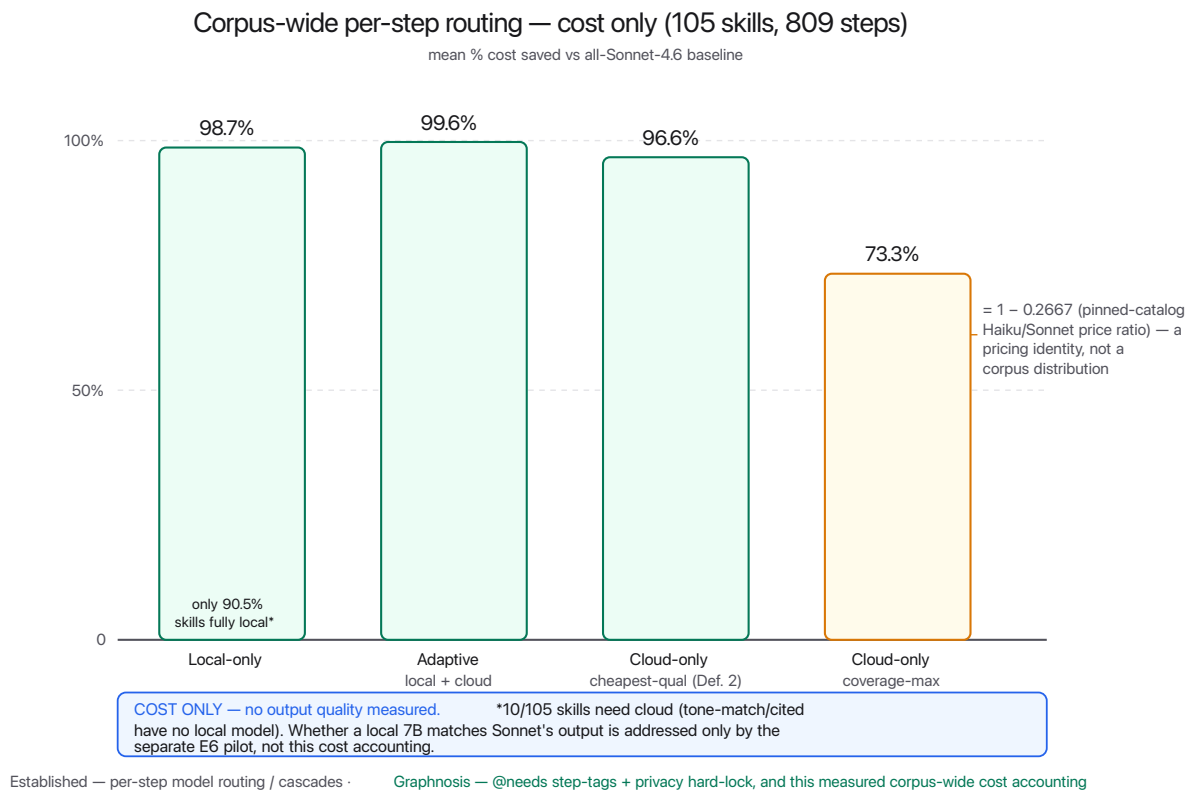


Figure 11. Corpus-wide per-step routing — a **cost accounting only** (105 of the 108-skill corpus, 809 steps; 3 excluded — two already-tagged with empty step bodies, and the body-truncated **skill-dispatch** flagged for repair), mean % saved vs the all-Sonnet-4.6 baseline. Under the realistic adaptive (local+cloud) policy ~99.6% is saved with every *included* skill routable (105/105); cloud-only is **96.6%** under the paper’s cheapest-qualifying rule (Definition 2, reaching Gemini Flash) and **73.3%** under coverage-max — and that 73.3% is the complement ($1 - 0.2667$) of the Haiku/Sonnet price ratio on the pinned catalog, a pricing

identity invariant to corpus size, not a distribution. Local-only saves 98.7% on cost but only **90.5% of skills (95/105) run fully local**, because `tone-match/cited` are served by no local model. No output quality is measured here; whether a local 7B matches Sonnet’s *output* is the separate E6 quality-retention pilot (§10.7). (Per-step view of one skill: Figure 4.)

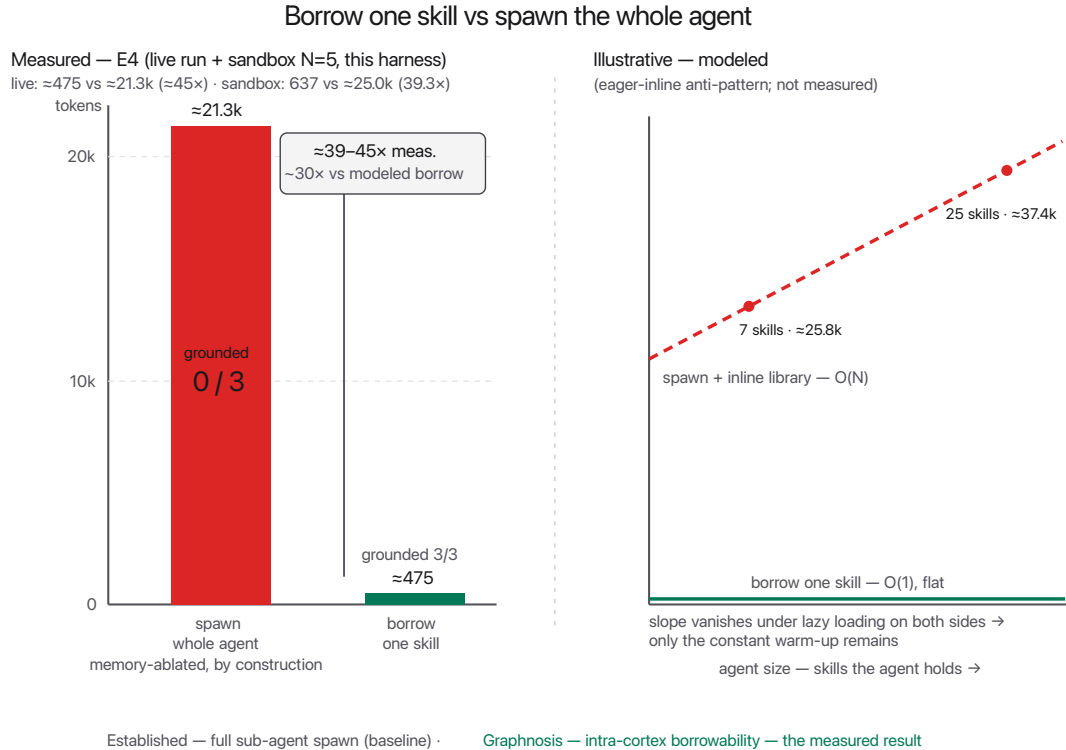


Figure 12. Borrow one skill vs spawn the whole agent (Proposition 5, §10.6). **Foreground — measured (two datapoints):** a **live** single run — a **≈475-token** in-cortex walk-and-recall (borrow, grounded 3/3 against a **≈21.3k-token** *full-client* cold spawn (≈45×; grounded 0/3, memory-ablated by instruction, so 0/3 by construction) — corroborated by a **reproducible sandbox** re-measurement — 5 cold spawns at ≈25.0k tokens against a deterministic 637-token borrow (39.3×) — a **≈39–45× measured separation, harness-specific** (≈21.3k is a harness-specific maximum, not an architectural constant); the separately-quoted ~30× is against the ~700-token *modeled* typical borrow. **Background — illustrative, modeled:** if a spawned agent eagerly inlines its whole library (denied the lazy dispatch of Proposition 3) its cost grows **O(N)** in library size — the rising dashed curve — while a borrow stays flat at one body (mean ≈642, median 550). With lazy loading on both sides the separation reduces to the constant warm-up that borrowing skips.

11. Discussion and Limitations

We scope the claims tightly; several of the boundaries below are flagged inline above and collected here.

Scope of evidence. This paper is a design, an artifact, and a deterministic-accounting existence-proof: it demonstrates that executable SOPs can be co-located with owner-held memory and then walked, routed, versioned, exported, and measured by exact accounting over the running system. It is not yet an outcome study. It does not provide task-outcome evidence, external-user or independent-corpus validation, or head-to-head baselines, so its claims of cost-benefit and generality are established *by construction and by accounting* — counts and deterministic ledgers over the artifact — rather than by controlled task outcomes, while **borrowability is credited to the measured E4 telemetry** (the live ≈475-vs-≈21.3k-token borrow-vs-spawn separation, §10.7), not to accounting alone. Every quantitative *accounting* here is therefore of

the system’s structure and contracts, not a statistical result over instrumented runs or user populations; the exceptions are the E4 borrow-vs-spawn measurement, the E5 co-location ablation, and the E6 quality-retention sanity note (§10.7), which we flag explicitly — E4 as a live single-run measurement corroborated by a sandbox, E5 and E6 as small internal demonstrations, not powered studies. This paper is deliberately scoped as a design-and-deterministic-accounting contribution and does not attempt a controlled outcome comparison; a task-outcome evaluation with independent users and external systems is a complementary, different kind of study — out of scope here and not a dependency of any result reported.

Designed-but-gated capabilities. The train-time memory-augmentation apparatus — relevance floor, self-tuning recall breadth, influential-node ranking (§4) — is dormant under the empty-engram invariant. We present it as future/experimental, not as live self-tuning. (The `[dispatch-safe:]` tag (§3, §7), reserved in earlier drafts, is now read both by the proactive watcher’s exclusion filter and by the autonomy gate of §13 — see *Addressed and planned*.)

Reference walker vs plan contract. The shipping in-app walker runs the single-call path, iterates steps linearly, and **now re-executes capped skill:loop back-edges in code** — each cap enforced by its scheduler, uncapped loops bounded by a default three-iteration cap, with a fixed-point early-exit (identical consecutive body outputs stop the loop), a hard per-walk execution backstop, and per-edge audit reports; the loop side is pinned by a model-free regression suite (`agent-walker-loops.test.ts`, 8 sections / 23 assertions, an internal CI record like the §10.4 suites). So for capped-loop, non-parallel skills the termination theorem (§5) is now a shipped-system guarantee, not only a contract property. What remains an obligation on an external executor is **parallel[] concurrency**: the plan declares parallel sibling groups but the walker does not dispatch them.

Capability tags ahead of content. Per-step `@needs` routing is real and measurable (§10.3) and was under-exercised by the *original* corpus (§9); a corpus-wide retrofit has since tagged 100 of 108 skills (71 of the original 76 among them), and those retrofitted tags — not the router-side inference fallback — are what the E2 routing accounting (§10.7) is computed over. The single-skill routing headline (§10.3) is what the engine *makes available*; the router-side inference pass is a fallback for future untagged steps, not the mechanism behind the E2 numbers. Relatedly, until the hardening described in §1.1/§4, the structure validator did not guard `@needs` or `only_engrams`, so a Pro rewrite could in principle have dropped a routing tag; the gap is now closed and regression-tested.

Measurement caveats. The routing savings (§10.3) and the dispatch token decomposition (§10.2) are both measured, but the underlying tokenization is a coarse characters-÷-4 heuristic, and the index figure deliberately overcounts (it includes the source id the live trigger table omits), so the reported reduction is conservative. Only the case-study token figures (§8) are reconstructed from history rather than instrumented.

Narrow vitality. Per-skill vitality scores age, supersession, and cited-node drift only; it does not reward goal completeness or structural resolution (§6), so genuine quality improvements (§9) can leave the number unmoved.

Data-quality debt. At least one shipped skill carries non-SOP content baked into its body (a retrieval/inference dump and duplicated steps); we excluded such content from measurement and flag it as a migration target rather than retraining over it blindly.

Pack confidentiality, gates, and safe import. Both pack formats — the `.gsk` skill pack and the `.gez` engram pack — encrypt their payload with a fixed, obfuscation-grade key embedded in the open source, so the cipher delivers integrity-by-signature, not confidentiality: the Ed25519 signature is the real guarantee, and at-rest confidentiality rests on the cortex’s own encryption, not the pack cipher. We therefore describe packs as *signed*, not *confidential*. This also bounds what a feature gate can claim. Because the codebase is open, a license check on pack import or export is a product boundary, not a security control — it cannot keep a confidential pack from anyone willing to recompile, and should never be presented as protection. Confidentiality for a pack that must survive on its own — an air-gapped transfer on physical media — has to come from a key the recipient holds (passphrase- or recipient-public-key encryption), not from a gate; with that in place, recompiling past the gate still yields a pack the attacker cannot read. Safety of *ingress* is a separate axis from confidentiality: an imported pack is untrusted content entering the cortex,

so the right controls are the substrate’s owner-adjudicated ones — land an import in a quarantined scope rather than merging it live, never auto-recall or auto-dispatch imported nodes, and treat an imported skill as not **dispatch-safe** until the owner promotes it, so the autonomy gate of §13 keeps it from running on its own. These are the contradiction-handling primitives applied to ingress: surface for the owner, never act silently. Recipient-controlled pack encryption and a quarantine-import path are now implemented and regression-tested: a pack can be sealed to a passphrase (Argon2id) or to a recipient’s public key (X25519) *over* the unchanged Ed25519 signature, so it is confidential as well as verifiable; and an imported pack lands in a quarantined scope — excluded from recall, dispatch, cross-skill resolution, and listing, and forced non-**dispatch-safe** — until the owner promotes it. The import gate has been dropped so trying a shared pack costs nothing; in-app authoring — the train/retrain compile — and signed **.gsk** pack export stay gated, while import, walking, and text-format export are free (§14).

Addressed and planned. Several of the gaps above have concrete fixes in hand for the next release: the structure-preservation validator now guards **@needs** and **only_engrams**; in-place retraining is name-normalized against the duplicate-fork defect; **dispatch-safe: no** is enforced in the proactive watcher; per-step **@needs** is *inferred* when a step declares none, so the router right-sizes the whole corpus rather than defaulting every un-annotated step to the smallest model; the savings ledger is exposed for inspection and now counts the answer tokens it avoids, not only the input; a **skill_lint** pass flags the body-corruption class noted above; and a capable local model is marked **writing-able** so the writing step that made a fully-local walk *infeasible at measurement time* (§10.3) can now run on-device. Three further directions, posed as open problems in earlier drafts, now have working, regression-tested cores. **Recall-recipe binding** is implemented as an opt-in train mode: rather than freezing personal nodes into the body, it emits **recall(...)** *recipes* gated by a relevance floor, so a skill personalizes cleanly at walk time without abandoning the empty-gram default — clean, fresh personalization without the empty-gram restriction *or* accumulated junk. A **contradiction self-heal policy** closes the gap between on-demand contradiction detection and regular maintenance: a pure, deterministic module decides when a health-sweep is due and which detected pairs are genuinely new — not already resolved or suppressed — and a walk-time guard surfaces contradicted memory so a recall step never feeds it downstream. And a **skills autonomy engine** (§13) answers “autonomous but still safe”: a pure L0–L3 gating decision in which the user’s chosen autonomy level is *capped by the skill’s authored safety*. In each case the *decision logic* ships under a model-free suite — the determinism checks of §10.4 plus each engine’s own tests — and the host wiring now exists: a contradiction-health scheduler runs the sweep on a cadence and surfaces newly-detected conflicts as owner-review cards; the walker consumes the contradiction guard; and the proactive watcher runs every matched skill through the autonomy gate, so an unsafe or low-confidence match cannot exceed *suggest*. That last frontier is now closed in the working tree, conservatively. A true *unattended* L3 executor walks an auto-eligible skill end-to-end with no human in the loop, behind an explicit owner opt-in that is **off by default**. It admits a run only when every interlock passes — kill-switch on, opt-in on, effective level re-resolved to L3 at execution time, recalled memory re-checked live for contradictions, the plan walker-executable (parallel dispatch is refused; a loop is admitted only when its cap is *authored* — an uncapped **@loop** is refused unattended even though an attended walk bounds it with the walker’s default cap, because an L3 run with no human watching requires the bound to be authored into the skill), and the side effects reversible — and it writes an encrypted, append-only per-action audit that a dedicated review surface renders with per-action undo. The surrounding guarantees it waited on are the ones now in place: quarantined ingress (above), the per-action audit, and reversible-only execution. Because today’s reference walker emits prompts rather than performing writes, the effects an unattended run can take are read-and-compute; a write-capable walk is the next extension.

12. Related Work

As with the companion paper, the contribution is a **synthesis, not any single first**: every ingredient has prior art, and we claim only their conjunction in one local-first, indelible memory graph. We concede the priors directly.

- **Standard operating procedures and runbooks** are an old idea; we borrow the vocabulary deliberately. What is new is compiling an SOP into a typed-edge subgraph rather than storing it as a

document.

- **Prompt-template skill libraries** (including agent “skills” shipped as markdown with frontmatter) capture reusable behavior, but as prose the executor must re-read in full, with no control-flow structure, no per-step routing, and no freshness model.
- **Agent workflow engines** (LangGraph-style DAGs and their kin) express control flow as a graph — but in a *separate* runtime and store, disjoint from the agent’s knowledge, so the workflow is neither versioned, encrypted, nor checkable against the facts it depends on.
- **Tool / skill catalogs** route a call to a capability; they do not encode a multi-step procedure with goals, loops, and recovery.
- **Model routing and cascades** select a model per request to trade cost against quality; we apply the same idea *per step of a procedure*, under a privacy-first precedence, but routing alone is not a procedural memory.
- **Graph-structured retrieval** (GraphRAG and lighter/lazy variants) builds a graph over a corpus for retrieval; it is a snapshot with no owner-adjudicated freshness model and no notion of an executable procedure stored *as* graph structure.
- **Retrieval-augmented generation / memory** (vector-store RAG and retrieval-augmented memory) fetches text and injects it into the prompt at query time; the fetched chunks are *read*, not walked, and sit in a store separate from any executable procedure — there is no co-located, owner-held procedure to run, which is the contrast the abstract draws.
- **LLM-compiled wikis** (Karpathy’s widely-shared *LLM-Wiki* pattern — an idea file, not a paper — and its Obsidian / Claude-Code descendants) shift the maintenance burden to an agent that compiles sources into interlinked plain-markdown pages: *compile-not-retrieve*, the same premise this paper’s substrate rests on. But the compiled artifact is declarative memory in flat files: no executable procedure, no control flow, no per-step routing; retrieval is the model re-reading an index and pages (not a typed traversal, and the files are unencrypted), and freshness is a periodic LLM “lint” rather than a deterministic staleness score. It is the popular articulation of the idea — and, since June 30, 2026, the vendor-articulated one: LangChain’s *Wiki Memory* post (Chase 2026 — References) names the same pattern, “an agent-maintained data structure that represents source knowledge in an agent-friendly way,” with files as the substrate and an agent as both compressor and maintainer, and concedes the scope limit itself (“Wikis are not all of memory”), while discussing no conflict handling, no staleness model, and no executable procedure. The procedural skill layer is what the pattern, in either articulation, does not attempt.
- **Coding-agent skill & lifecycle tooling** (Google’s *Agents CLI* for the Agent Development Kit, and kin) injects bundled “skills” into a coding assistant to teach it a framework’s scaffold / eval / deploy patterns — build-time tooling scoped to a vendor stack (ADK + Google Cloud). Those skills are re-injected instruction packs that drive a *build-and-ship* lifecycle, not owned procedures co-located with the agent’s memory; they carry no control flow, per-step routing, recall binding, or vitality of their own. This is the broader “skills as the unit for agents” wave — also Karpathy’s *agentic engineering* (spec design, eval loops, security oversight) — which the Skill Subgraph sits one layer beneath, as the owned, durable, model-agnostic memory-and-procedure substrate such lifecycles run on.
- **Function-like skill composition** — executable code-actions that compose tools inside one agent (CodeAct), agent-as-tool invocation with structured input and a returned value, and typed skill-graphs (SkillDAG, GraSP) — composes procedures, but within a single agent or as a separate graph; none reports one *owned* skill calling another across a shared memory substrate, nor the per-call token comparison against sub-agent spawning we give in §10.6.
- **Procedural-memory research** and classical truth-maintenance/belief-revision inform the lifecycle and the surfacing posture inherited from the substrate.

The conjunction we claim is the set held together: **SOPs as typed-edge subgraphs + per-step capability routing + function-like sub-skill calls + bounded loops + walk-time recall bindings + a vitality-driven retrain loop, all in the same indelible, encrypted, owner-held graph that stores declarative memory.** Stated defensively: this is not “just a workflow engine” (the graph is the memory substrate, not a separate runtime), not “just a prompt library” (control flow is structure, loaded lazily, not prose re-read in full), and not “just routing” (routing is one step-level operation over a persistent procedural

state).

We follow the substrate paper’s standard: a source-audited capability matrix across the eight synthesis axes, naming systems generously and citing each (URLs and access dates in the References). Ratings: Yes clearly yes, partial, No clearly no. **These are author-rated positioning judgments, not an independently operationalized benchmark.** Every competitor cell is read from each system’s public documentation and code (not from our own execution of those systems against a common task suite), and the axes are the ones this work was designed around; an author of a different system would likely draw the axes differently. The matrix should therefore be read as a structured *positioning argument* — where the Skill Subgraph’s conjunction sits relative to prior art — and not as an evidentiary or comparative-performance claim.

System	Ctrl-flow	Per-step route	Sub-skill calls	Bounded loops	Recall bindings	Vitality	Local	Co-located
LangGraph	Yes		Yes	Yes	Yes	No	Yes	No
Microsoft GraphRAG	No	Yes	No	No	No	No	Yes	No
LightRAG / Lazy-GraphRAG	No	Yes	No		No		Yes/?	No
Model routing / cascades		Yes	No		No	No		No
Mem0	No	No	No	No	No		Yes	
Zep / Graphiti	No	No	No	No	No		Yes	No
MemAgent	No	No	No	No	No	No		No
Letta / MemGPT	No	No	No	No			Yes	
A-MEM	No	No	No	No	No	No	Yes	No
LLM-Wiki (Karpathy)	No	No	No	No	No		Yes	No
Anthropic Agent Skills				No	No	No	Yes	
Google Agents CLI (ADK)	No	No	No	No	No	No		No
Graphnosis Skill Sub-graph	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Read plainly, the matrix is a positioning map, not a ranking: on most individual axes a prior system matches or exceeds the Skill Subgraph, and we claim no single axis as a first. **LangGraph** is a strong procedural comparator — structured control flow, sub-skill composition with argument binding and return capture, a bounded recursion limit, and execution-time bindings to a persistence layer — but its workflow is *code*, separate from the agent’s memory, and it has no freshness model (No on co-location and vitality). Within the shared Yes on bounded loops the mechanisms differ (a documentation-based

design contrast, not a benchmark): LangGraph bounds a run with one *global* recursion budget authored in code (`GRAPH_RECURSION_LIMIT`, default 25 supersteps — References), and exceeding it raises `GraphRecursionError` — the run fails; the Skill Subgraph’s cap is *per loop edge*, authored in the SOP text itself (`@loop: N max=M`), and reaching it is a normal, reported outcome — the walk falls through and completes, carrying per-edge iterations, cap source, and stop reason in the walk result, with the stopping decision (cap or deterministic fixed-point convergence) made by code, never by the model. LangGraph’s cyclic graphs remain more *expressive* — arbitrary conditional cycles, code predicates — where the Skill Subgraph deliberately narrows to capped back-edges for provable termination. The authored-versus-defaulted cap distinction is load-bearing for §11: the unattended gate keys on whether a loop’s bound was *authored* into the skill, a property a global runtime knob does not expose. The **routing/cascade** family (RouteLLM, FrugalGPT) and the **graph-RAG** systems own per-step or per-task model selection, which we match but do not claim. The **memory** systems do declarative-*fact* freshness — Mem0’s LLM-adjudicated updates, Graphiti’s bi-temporal invalidation — but neither stores an executable procedure, so neither refreshes one. The recent **memory-agent** crop is the nearest namesake but a different layer: **MemAgent** is an RL long-context compression technique — a fixed memory panel overwritten per chunk, with no stored procedure at all; **A-MEM** evolves Zettelkasten-style declarative notes; and **Letta (MemGPT)** comes closest, co-locating on-demand *prose* “skills” in a git-versioned memory store and consolidating in sleep-time passes (hence on co-location and vitality) — yet those skills are markdown reference guides, not typed-edge procedures with control flow, per-step routing, and bounded loops. They remember better; they do not store and *walk* an executable procedure. **Anthropic Agent Skills** is the closest packaged “skill” format — prose steps, a skill-level model override, forked-subagent composition; our design differs in encoding control flow as typed edges, binding recall at walk time, and tracking vitality — whether those differences matter for a given task is not something this paper measures. The **LLM-Wiki** pattern (Karpathy; articulated at vendor scale by LangChain’s *Wiki Memory* post, June 2026 — References) and its Obsidian descendants are a knowledge-*compilation* comparator rather than a procedural one: they compile sources into flat-markdown pages an agent maintains — sharing the *compile-not-retrieve* premise — but hold no executable procedure (No across the procedural axes) and refresh by a periodic LLM lint (), not a staleness-driven vitality score; the contrast there is a knowledge-base contrast (developed in the companion paper) more than a procedural one.

What this design combines that we did not find combined elsewhere is the *conjunction* — in particular **(f) a staleness-driven vitality model and (h) co-location of an executable procedure with declarative memory in one store** — offered as a positioning claim about where the design sits in the space, not a performance ranking against other systems. Letta/MemGPT comes closest — partial on both, by versioning prose skills beside memory and consolidating in sleep-time passes — yet none pairs a true staleness-driven refresh with executable-procedure co-location. That pair — held together with the other six inside a local-first, indelible, encrypted graph — is the contribution as we have positioned it. (Audit caveats: every competitor rating is inferred from public documentation and code, not from our own execution of those systems, so a No generally means *not documented* rather than *proven absent* — features may exist undocumented, or have shipped since our access dates. In particular, LangGraph’s No on vitality/co-location rests partly on absence of documented features — that row was re-checked (2026-07-03) against LangChain’s 2026 platform additions: the LangSmith Context Hub (announced May 2026) versions and tags skills-as-prose-files in a central hub, and the trace-driven Observability→Engine→Context Hub improvement loop (June 2026) is the nearest managed analogue of the retrain loop, but both amount to change-tracking plus LLM-proposed edits in a store separate from agent memory, not a staleness-driven vitality score co-located with it, so both cells stand; Anthropic Agent Skills, Letta/MemGPT, and the memory systems are rated from their docs alone; and LazyGraphRAG is blog-described and unreleased, so its locality is unknown. The Skill Subgraph’s own **bounded-loops** rating is **Yes** as of 2026-07-03: earlier drafts carried because loop *re-execution* was an obligation on contract-honoring executors that the reference walker did not itself perform; the walker now re-executes loop bodies with the cap enforced in code — authored `max=M`, a default cap of three for uncapped loops, fixed-point early exit, and a hard per-walk backstop — under a model-free regression suite, with `parallel[]` dispatch remaining the contract-only feature (§5, §11).)

One SOP, Three Encodings

the same procedure, three substrates

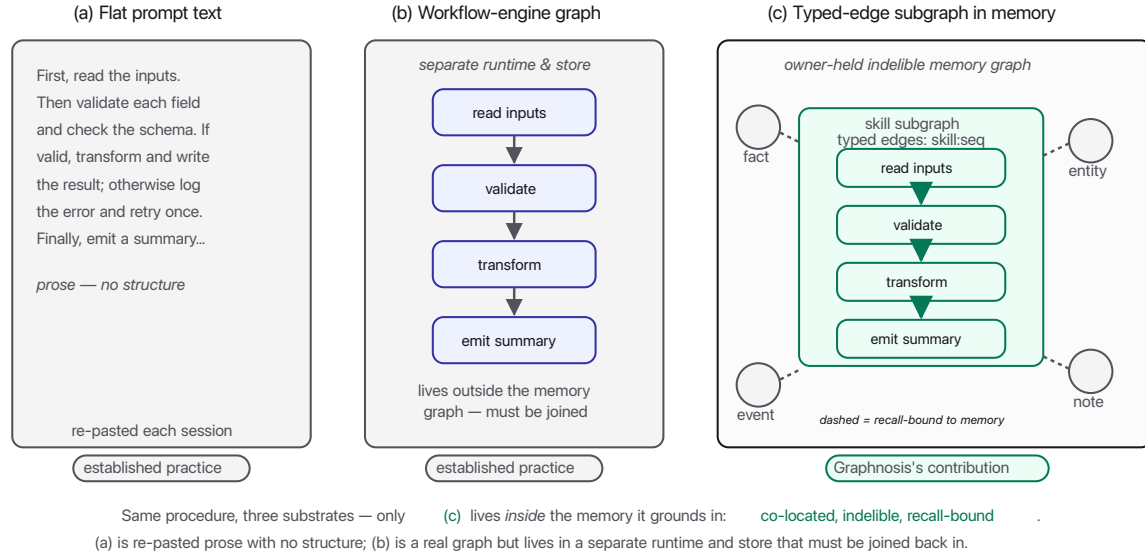


Figure S3. One SOP, three encodings. The same procedure as (a) flat prompt text, (b) a graph in a separate workflow engine, and (c) a typed-edge subgraph *inside* the owner-held indelible memory graph. Only (c) is co-located with declarative memory, indelible, and recall-bound — the synthesis this paper claims (§12).

13. Imagine the Best Set of Skills Possible

The corpus in this paper is one founder’s library; the more useful question for the reader is what an *ideal* library looks like and how to build one. The mechanisms above suggest concrete answers.

The expanded possibility (a design conjecture, not a result). What generalizes here is the *method*, not a demonstrated outcome: the design is, by construction, domain-agnostic — anything repeatable, with goals and steps, compiles into a subgraph — so the same training procedure *should* let a team grow an owner-held, self-improving SOP library for whatever work it actually does: engineering and go-to-market here, but in principle research, creative production, operations, compliance, fundraising, or community-building. We claim this generality as a property of the *method*, not as transfer we have demonstrated for external builders: the evidence is a single owner’s corpus trained by one practitioner, and cross-builder, cross-domain adoption remains unmeasured (§9). Within that corpus, each procedure trained can make later matched sessions cheaper and more reliable (a new skill adds index cost and pays off only when it is later matched and walked), and each domain added composes with the others through the shared call graph and the single memory beneath them. Read this way, the paper is less a system description than a playbook: start a small library in your own field, let indexed work train it, and let the trained skills run the next round of work. The unit of competitive advantage stops being the prompt and becomes the private, governed, ever-improving procedure graph you own.

For the developer. Make `session-start` and `skill-dispatch` the entry hub, so every session begins by loading the lean index and hydrating only what the work needs. Give every code skill a machine oracle — `[verify: tool]` or `[verify: state]` — so an agent knows, without asking, when a step is truly done. And install the **self-improvement triad** — `retrospective-learning` → `adaptive-skill-creation` → `dispatch retrain` — so the library compounds: each mistake becomes a procedure, each procedure self-registers, and the next session is run by a slightly better agent than the last.

For the founder. Bind business/GTM skills to private engrams at walk time — fundraising, pricing, go-to-market, enterprise sales — so strategy *memory* stays on-device under a consent gate while the *procedure* stays generic and shareable. A founder’s edge is in the facts; the SOP can be given away, and — under the local-only routing strategy — the facts never leave the laptop (the per-step privacy lock is per-step, with no taint propagation; Invariant 2).

The design principles, distilled. The eight tags the corpus already uses are a portable design language: *empty-engram training* for portable SOPs; `[verify:]` oracles for halting; `[dispatch-safe:]` gating for safe autonomy; bounded `@loop` and the depth cap for runaway-free composition; `@needs` for cost control; `only_engrams` for privacy. Build skills that declare all eight goal categories, prefer machine oracles where the work admits them, and keep private facts at the binding rather than in the body.

Autonomy, bounded. The same tags make safe autonomous *execution* tractable — the natural endpoint of “imagine the best library”: skills that don’t merely list their steps but run them. The governing principle is that autonomy is capped by authored safety, never the reverse. An autonomy level — L0 manual → L1 suggest → L2 preview-then-run → L3 autonomous — sets how far the system *may* go, and that level is resolved *per skill*: a skill may set its own level, falling back to the engram’s default and then a global default when it sets none, and the authored `[dispatch-safe:]` tag is a hard cap over whatever level resolves. So the effective level is the minimum of the authored cap and the per-skill level (or, absent one, the engram or global default); the skill’s meta/router status, the presence of an unresolved contradiction in recalled memory, and the trigger-match confidence then set how far it *actually* goes. A `dispatch-safe: yes` skill with a high-confidence trigger and uncontradicted memory is *permitted* to run end-to-end at L3 — the contract ceiling, not a demonstrated write-capable run: the unattended executor that exercises L3 today takes only read-and-compute effects, because the reference walker emits prompts rather than performing writes and does not dispatch `parallel[]` siblings (capped loops it re-executes with the cap enforced in code; an unattended run admits a loop only with an authored cap, §11); anything less resolves down to preview-and-confirm or a passive suggestion, and a router skill never auto-runs at all. This is the answer to the determinism worry: determinism lives in the substrate, the compiled plan, and these guardrails — not in the model’s word-by-word creativity — so a skill can be both autonomous and dependable. A working core of this engine ships and is regression-tested (§11), and the once-deferred background executor and its review-and-undo UI are now implemented behind an owner opt-in that is off by default (§11).

Invariant 4 (Authored safety caps autonomy). *Over the ordered levels $L_0 < L_1 < L_2 < L_3$, a skill’s effective level is $L_{\text{eff}} = \min(\text{cap}_{\text{authored}}, L_{\text{resolved}})$, where L_{resolved} is the per-skill level, else the engram default, else the global default, and $\text{cap}_{\text{authored}}$ is the ceiling set by the `[dispatch-safe:]` tag (`no`→*suggest*, `partial`→*preview*, `yes`→*auto*), with a meta/router skill forced to suggest and an unresolved contradiction in recalled memory forced to preview; a trigger-match confidence below the floor (0.6) clamps the exercised action to at most suggest. Autonomy is thus capped by authored safety, never the reverse, and a router skill never auto-runs. Proof (by construction). The ceiling and the confidence floor enter only as min operations over the ordered level set, so each can lower but never raise L_{eff} ; the decision is pure and model-free, regression-pinned (§11). ■ (The unattended L3 executor is off by default and admits a run only when a conjunctive gate — kill-switch, opt-in, $L_{\text{eff}} = L_3$, live contradiction re-check, walker-executable plan (parallel refused; loops only with an authored cap), reversible effects, rate limit — all holds, §11.)*

Distribution. A finished library travels as a signed .gsk pack (AES-256-GCM payload, Ed25519 signature — the signature provides integrity; optional passphrase- or recipient-key encryption adds confidentiality, §11) carrying the full eight-goal structure and recall recipes; a curated engineering subset of the founder’s own library is published as a public pack at github.com/nehloo/graphnosis-trained-skills (the full 76/108-skill corpus measured here is not published). The unit of reuse is no longer a prompt — it is an owner-held, self-improving procedure graph. Within one cortex it is **borrowable**: a sibling Graphnosis agent borrows a skill at walk time through an `@skill:` call (§10.6, the measured result). Across owners it is *intended* to be portable; independent import-and-walk remains unevaluated (the E1 binding proxy in §10.7 is the nearest evidence): another owner could import the whole library as a signed pack *designed to bind to their cortex* — the empty-engram invariant (§4) keeps *automatically-recalled* private facts out of the body, though it does

not sanitize details an author hand-writes into the SOP text or sensitive engram *names* in recall recipes, which remain the owner’s to review before sharing. Indexed work trains the skills; the skills then do the next work.

13.1 Write the paper as an audit of what you built

A methodological note, beyond Graphnosis, for anyone shipping a new capability. The order here was build-first: the substrate, the skill kit, the corpus, and the autonomy engine were envisioned and implemented before this paper existed. What the paper added was not the building but a *reckoning* with it — and that turned out to matter more than expected. The discipline a write-up demands — that every claim be defensible, every number reproducible, every mechanism stated exactly — is an instrument of inspection when you turn it on a system that is already standing. Writing forced rocks to be lifted that the building had left in place: an assertion taken on faith had to be verified, a figure that didn’t quite add up had to be reconciled, a failure mode that had been quietly routed around had to be named. Several fixes in this work began as sentences that would not survive being written down — the dispatch-but-don’t-walk gap (§8), and the chunk-splitting and goal-completeness defects caught while cataloguing the corpus for §9, each surfaced by the act of accounting for the system rather than by using it. The lesson is not “write while you build”; it is that writing a rigorous account *of* a finished system is one of the surest ways to find where it is still wrong — and to fix it before a reader does. The manuscript paid for itself in defects surfaced, not only in claims made.

13.2 Train Your Own Skills, on Your Own Memories, with Any Client

Why these domains, and why yours are different. The families in this corpus span eleven domains plus the chat agent’s own — engineering, go-to-market, meta/agentic, research, creative, nonprofit, heritage, event organizing, film, acting, reviewing, and the chat agent itself (twelve families in all) — and are not a synthetic benchmark. They are one practitioner’s actual working domains, and the procedures in them are the ones that earned their place by being used. That matters for two reasons. First, it keeps the test grounded: a skill like **ship-workflow** or **fundraising-readiness** is real, hard-won procedure from a working life, not an invented task chosen to flatter the system, which is the fair unit to measure a method against. A corpus authored by its owner is the right kind of evidence for a system meant to be held by its owner — the same argument the companion paper makes for owning your memory, applied to procedure. Second, the specific mix is beside the point, and that is the point: your domains are not these domains, and the method does not depend on which they are. The empty-engram contract (§4) and the eight goal categories (§3) are domain-agnostic by construction — each new family in §9 was trained by the identical method with cortex recall disabled, binding private facts only at walk time. Map it onto whatever your work actually is. The reflexivity is not lost on us: a *reviewing* skill in this very corpus could, in principle, walk its procedure over a paper like this one.

The method this paper measures was not built for one founder, and it does not require the founder’s tools to reproduce. The substrate is reached over the Model Context Protocol (§14), so any client that speaks MCP — a local model under Ollama or LM Studio, a desktop assistant, or a cloud agent such as Claude Code, Cursor, or an IDE plug-in — can train and walk a skill against the same owner-held graph. What follows is the reader’s own loop, stated as a procedure rather than a pitch, because the finding worth carrying away is that the *structure* is portable and the *structure* is what the evaluation credits with the savings (§10.2, §10.3).

The only precondition is that your work contains procedures that repeat. A developer has a release ritual; a researcher has a literature-review and citation-audit pass; a musician has a mixdown-and-release checklist; an event organizer has a run-of-show; a genealogist has a source-triage and lineage-verification method; a board member has meeting prep and grant drafting; a teacher has a lesson-and-assessment cycle; a clinician has an intake protocol. Each is *repeatable, with goals and steps* — which is the only thing required to compile it into a subgraph (§3, §9). The further families we trained by the identical method (§9) are the evidence that nothing in the procedure is code-specific; we have not tested every field named above, and offer them as instances of the same precondition, not as measured results.

The loop, concretely.

1. **Keep your memories in your own cortex.** Before any skill, the facts the work depends on — your style guide, your venues, your repositories, your archive conventions, your prior decisions — live as declarative memory in the local graph (the substrate of the companion paper). They stay on your machine, encrypted under one key, indelible (§2). This is the ownership the rest of the loop rests on: a step will *reach* these facts at walk time, but training never copies them into the skill.
2. **Write the procedure as plain numbered steps plus the eight goal categories.** State the SOP the way you would brief a careful colleague: a numbered body, then the contract in the eight typed goals of §3 — *Trigger* (when this fires), *Prerequisites* and *Requires* (what must be true, and what inputs it needs), *Produces* (its outputs), *Success* with a `[verify: tool|state|human]` oracle (how you will know it is done — a passing test, a fresh artifact, or a human sign-off), *Out of scope*, *On failure* (the recovery step), and *On completion*. Most non-code work closes on `[verify: human]`, which is a legitimate oracle, not a weaker one (§9). Authoring all eight is what later lets skills bind arguments and compose (§5).
3. **Train it empty-engram, so the SOP stays generic and shareable.** Compile with cortex recall disabled — the empty-engram contract (§4). No personal cortex node is *automatically* woven into the body; the step reads “recall the prior tier decisions,” not last quarter’s actual numbers. Private grounding is deferred to walk time through recall bindings: a step carries `only_engrams=[...]` naming exactly which engrams it may read (§5). This is the property §9 demonstrates with `fundraising-readiness` — the procedure is portable and discreet at once. A teacher can share a grading SOP that exposes no student record; a clinician can share an intake protocol with no patient in it; a lawyer can share a conflicts check that names no client. The facts bind on your machine, only when the skill runs, only from the engrams you named, and — when an engram is sensitive — behind a consent gate with local-only routing (§5, §9).
4. **Let the AI walk it next time, routing each step to the cheapest capable model.** Tag steps with `@needs` (e.g. `extraction`, `reasoning`, `writing`) and the read-only planner resolves each to the cheapest model whose capabilities cover it, under a `privacy→strategy→cost` precedence, with sensitive-engram steps hard-locked to local inference (§5). On the annotated nine-step skill we measured, eight of nine steps ran free and local and only the `writing` step reached a paid cloud model (measurement-time catalog; the shipped catalog routes all nine local, §10.3) — 99.5% below an all-Sonnet-4.6 baseline when local models are available (a measurement-time-catalog figure; the reproducible headline is the E2 corpus-wide cost accounting, §10.7); cloud-only saves ~96.6% under the cheapest-qualifying rule and 73.3% under the premium coverage-max policy (that 73.3% is the complement $(1 - 0.2667)$ of the pinned-catalog Haiku/Sonnet price ratio) — all cost-only figures (§10.3, §10.7). Dispatch keeps the rest of your library out of the prompt: a session loads a one-line index and hydrates only the one to three skills it matches (§7, §10.2) — for a one-to-two-skill session, roughly 8% of the library, a 91.5–92.6% reduction; a three-skill match is a slightly smaller ~90.4% reduction. Because the reduction is a ratio of two token counts, it is robust to the characters-÷-4 tokenization (§10.2); the *mechanism* is available at your scale, though the exact ratio must be recomputed from your corpus, matched-skill count, annotations, and model catalog.

What this gives you, stated plainly. You keep ownership: the procedures and the facts are yours, in your store, exportable as a signed `.gsk` pack (§13) and deletable on your terms. You keep privacy: local-first by default, with the strongest routing guarantees on exactly the steps that touch sensitive memory (§5). And the improvement compounds — each procedure you train makes the next session cheaper to run, and the self-improvement triad (§13) lets a mistake become an amended procedure that self-registers. The savings are not a property of our corpus that you must take on faith; they follow from the *structure* — empty-engram bodies, lazy dispatch, per-step routing — so the same accounting *method* can be applied to another library on any MCP client; whether comparable savings appear depends on that corpus, matched-skill count, annotations, and catalog.

13.3 Next: Skills and Memory in the Owner’s Own Language

The trained skill corpus measured here was authored in English against an English codebase — but the declarative memory it grounds in was not monolingual. The author’s native language is Romanian, and this cortex already held Romanian memories alongside English; cross-lingual recall was therefore exercised informally throughout the build — observed to work in everyday use, simply never isolated into a measurement or written up. The substrate and the skill layer are, by design, language-agnostic at the layers that matter — multilingual embeddings carry recall, entity extraction is not English-tied, and the trainer already ships loop- and branch-pattern sets for several languages (§3). The next study in this line asks whether that design holds end to end: a deterministic, local-first, co-located cortex in which *both* declarative memories and trained skills — written in any language, or in a mixture of languages within a single SOP — still resolve, recall, and walk correctly. The questions are concrete and empirical: does cross-language recall ground a Romanian skill step against a Hebrew memory; does dispatch match a trigger authored in one language to a session conducted in another; does the typed-edge skill structure stay language-neutral while only its prose varies? The architecture is built to make this *possible* — the goal categories, DSL tokens, and routing tags are structural, not lexical — and the bilingual cortex above is early, informal evidence that the memory half already holds. But everyday use is not measurement, and the skill half is still English-authored. A controlled multilingual study — skills written in non-English or in a mixture of languages, with cross-language grounding and dispatch measured rather than observed — is a separate line of work, not a dependency of the results here.

14. Implementation and Availability

The skill layer is reached over the Model Context Protocol by the same clients that reach Graphnosis memory. The procedural tool surface is twelve calls: `list_skills` and `get_skill` (discovery); `train_skill` (compile/retrain) and `export_skill` / `delete_skill`; `walk_skill` (narrative) and `walk_skill_structured` (the execution-plan contract); `save_skill_run` / `resume_skill_run` (cross-session resume); and `skill_history`, `rollback_skill`, `skill_vitality` (lifecycle). Read tools are deterministic. Note that this surface exposes no one-call `.gsk skill`-pack import: `engram .gez` pack import ships and lands in a quarantined scope (§11), but importing a signed `.gsk` skill pack in a single call is not among these twelve tools — which is why E1’s transport was shown via a source-grounded retrain into the recipient engram rather than a true pack import (§10.7). Walking, text-format export, and `rollback_skill` are available license-free and preserve snapshot history; `train_skill` (compile/retrain) and signed `.gsk` pack export are the surfaces that consult the license gate of §4, and both training and rollback preserve the encrypted snapshot lineage either way.

The implementation is the application sidecar, with a clear file map: `skill-trainer.ts` (compilation, the SOP edge linkers, snapshots, vitality, hollow-skill repair); `skill-sop-rewrite.ts` (the structure-preservation validator); `model-router.ts` and `model-registry.ts` (per-step capability routing and the model catalog); `agent-walker.ts` (the reference walker and routing-savings accounting); `skill-runs.ts` (resumable runs); `skill-call-links.ts` (the cross-gram call side-table); `proactive-watcher.ts` / `proactive-dispatch-match.ts` (dispatch); `savings-tracker.ts` (the cost ledger); and `gsk-format.ts` (signed packs). The generic node/edge/op-log/encryption substrate beneath all of this is the SDK, `@nehloo/graphnosis`, documented in the companion paper.

What the license gates, precisely. The method this paper describes does not, in its core, sit behind a paywall. The eight-category skill structure, the typed-edge subgraph encoding, lazy dispatch, per-step capability routing, and walking a trained skill with any local or cloud model are available without a paid token — including the token-cost reductions those mechanisms produce (§10.2, §10.3), since the savings are a property of the structure and the router, not of the license. The deterministic, memory-augmented compile of §4 — the default path, which runs with no model in the loop — is what the `skill-training` license gates, together with export of the signed `.gsk` distribution pack: the gate is checked at the transport layer on both the train/retrain surface and the `.gsk-export` surface. What it buys is the convenience of *running that compile in-app*, and the optional Pro LLM *rewrite* that polishes a skill’s prose during compilation (§4); that rewrite is a detachable overlay which, under the structure-preservation gate, can never silently *drop* an

executable token, routing tag, recall binding, or goal header (§4, Proposition 2 — set-presence, not per-step placement). Storing skills, importing signed `.gsk` packs, exporting a skill in any text format (`raw`, `claude-md`, `cursorrules`, `system-prompt`, `openai`), walking already-trained skills, and previewing routing cost all work license-free; only packaging a skill into a *signed .gsk pack* joins the in-app compile behind the gate. We state this precisely because an earlier framing in this paper read as if the *core procedural value* were paywalled; it is not — what is gated is the in-app training surface and signed-pack export, while the structure, the routing savings, and every read-and-walk operation are open.

Licensing. This paper’s text and figures are released **CC-BY-4.0**. The SDK is **Apache-2.0**. The skill-training pipeline — and export of the signed `.gsk` pack — is a paid feature gated by an Ed25519-signed license token; storing skills, importing packs, exporting skills in any text format, and walking already-trained skills do not require it. A curated engineering subset of the corpus discussed here is published as a signed `.gsk` pack at github.com/nehloo/graphnosis-trained-skills; the full 76/108-skill corpus is not published.

15. The Un-Brain’s Procedural Cortex: Correcting the Glitches of Biological Task Control

The companion paper named its substrate the *Un-Brain* because it deliberately inverts the failure modes of biological declarative memory: it is indelible where the brain decays, deterministic where the brain confabulates, and owner-adjudicated where the brain silently overwrites. The procedural layer of this paper extends that inversion one level up — from *what is remembered* to *how stored knowledge is turned into goal-directed action*. The mapping below is offered as a structured **analogy**, not a model of cognition; we mark throughout which claims are established neuroscience and which are correspondences we draw to the architecture. Graphnosis does not replicate the brain, and we make no claim that it is brain-like in mechanism. It is useful precisely where it *diverges* from biology.

15.1 How the brain selects and sequences action

Four interacting systems carry the load of converting stored knowledge into ordered, goal-directed behavior. The following are well-established findings, stated as such; the analogy comes only in §15.2.

- **Prefrontal cortex (goal maintenance, task-set, cognitive control).** Miller and Cohen’s integrative theory holds that the prefrontal cortex maintains an active representation of goals and the rules (“task-set”) that bias processing toward goal-relevant pathways — a substrate of cognitive control [1]. This representation is part of what keeps a multi-step intention coherent against distraction.
- **Basal ganglia (action selection, procedural “how,” chunking, gating).** Graybiel’s work proposed that the basal ganglia compile repeated action sequences into **chunks** — units that can be expressed as a single program — and contribute to gating which action is released at a given moment, a neural basis of procedural habit [2]. The “how” of a well-learned routine is associated with this circuitry, sequenced and increasingly automatic. The striatum is itself compartmentalized into **striosome** (patch) and matrix modules [10] — a modular substructure the analogy returns to in §15.2.
- **Hippocampus (rapid binding, episodic retrieval, pattern completion).** Complementary-learning-systems theory holds that the hippocampus rapidly binds the elements of an experience into a retrievable trace and, on a partial cue, can complete the pattern to reinstate the whole — the retrieval operation that supplies episodic context to an ongoing task [3].
- **Engram cells (sparse, distributed memory traces).** Josselyn and Tonegawa’s engram research showed that a specific memory can be carried by a sparse, distributed ensemble of cells whose reactivation is, in the paradigms tested, both necessary and sufficient to drive memory expression [4] — evidence for a physical, addressable unit of a stored memory.

In broad strokes, a goal held in prefrontal cortex recruits hippocampal retrieval of relevant traces and a basal-ganglia procedure to act on them. This is the machinery against which the Skill Subgraph can be read — as a lens, not an equivalence.

15.2 The correspondence, drawn precisely

Each pairing below is **analogy**. We name the mechanism in this paper and the neural system it rhymes with, and we do not claim implementational similarity or shared representation.

- **Plan compilation $\approx$ prefrontal task-set.** Compiling a Skill’s subgraph into an execution plan (§5) fixes the goal contract — the eight goal categories of §3 — and the ordered intention for the run, in loose analogy to prefrontal cortex holding a goal and its rules active. The compiled plan is the fixed task-set the walk is biased toward.
- **The typed-edge subgraph + deterministic walk $\approx$ basal-ganglia chunking and sequencing.** A trained Skill stores its procedure as a chunk of ordered, typed steps (`skill:seq`, `skill:loop`, `skill:branch`; §3) that is retrieved and run as a unit (§5) — the “how,” compiled once and replayed, by analogy to the basal ganglia compiling an action sequence into a habit. Each such skill is, in this lens, a **striosome** — a discrete, modular procedure compartment of the Agempus’s basal ganglia, the way striosomes are the modular compartments of the striatum; the rest of the paper keeps the engineering term *skill*.
- **Recall bindings at walk time $\approx$ hippocampal retrieval of relevant traces.** A recall step grounds a generic SOP in private memory only when the procedure runs (§4, §5), reinstating the specific facts a step needs — analogous to the hippocampal supply of episodic context to an unfolding action, with the substrate’s engram nodes standing in for the retrieved traces.
- **Per-step routing + guardrails $\approx$ cognitive control.** The read-only routing pass and the autonomy gating of §5/§13 decide, per step, which resource runs it and how far it may go — a control signal that biases each step toward its goal-appropriate execution, by analogy to the prefrontal control function Miller and Cohen describe [1].

Read together, these pairings locate the **Agempus** (§9.1) not in any one structure but in the *loop* among the three: a goal held as a compiled plan (prefrontal), the procedure selected and sequenced to serve it (basal ganglia), grounded by recall of the traces it needs (hippocampus). Biology runs this as the cortico–basal-ganglia–hippocampal loop; an Agempus is that loop made durable and owner-held — its procedures, their recall bindings, its dispatch index, and its call graph co-located in one engram, with the model *rented in per step* (§5) rather than fused into the circuit. This is the precise sense in which the basal ganglia is the *action-selector* but not, by itself, the agent: dispatch and the deterministic walk (§5, §7) are the selection-and-sequencing function, yet the agent is the whole loop, which is why the engram — procedures bound to the memory that grounds them — and not any single component is the unit we name the agent. It is also why the name keeps the hippocampal root: an agent defined by *owning its memory* is named for the structure that binds memory in, even though action selection sits next door. And because the Agempi **share one cortex**, an Agempus reaches a sibling’s striosome in the same memory rather than standing up a fresh brain — the anatomical reading of the cross-skill-call saving (§10.6).

15.3 Where the Un-Brain departs from the brain

The point of the analogy is the divergence. Biological task control is powerful but unreliable in four ways the Graphnosis procedural cortex is built *not* to share. Below, the left side is an established failure mode of biological memory; the right side is a design property of this paper. The contrast is the argument; we are not claiming the brain *ought* to behave like the system.

- **Decay vs. indelibility.** Memory traces weaken and procedures fade without rehearsal. A trained Skill does not decay silently: its nodes are indelible under the substrate’s correction discipline (§2, §6), and its *vitality* score (§6) makes aging **visible and measurable** rather than letting it pass unnoticed. Where biology forgets quietly, the Un-Brain surfaces staleness and queues a governed retrain.
- **Confabulation / false memory vs. determinism.** Human recall is reconstructive: retrieving a memory can render it labile and re-store it altered, and suggestion can implant details that were never experienced — reconsolidation and misinformation effects are well documented [5, 6]. The Graphnosis cortex does not reconstruct on read. A walk is a deterministic compile from immutable node order (§5, Theorem 1); training is structure-preserving (§4, Proposition 2 — set-presence); and a rewrite that would *drop* an executable token is reverted to the verbatim source. The procedure that runs is the

procedure that was stored.

- **Interference and fatigue vs. stable, bounded execution.** Biological control degrades under competing task-sets and depletes with effort. The walk is unaffected by load: capped loops and a depth-3 recursion cap bound termination (§5) — enforced in code by the shipped walker’s loop scheduler (§11) — the compiled plan is byte-identical across runs under its canonical serialization, and execution cost is previewable before a token is spent (§5, §10.3). There is no fatigue term and no cross-task interference.
- **Silent overwrite vs. owner adjudication.** Reconsolidation can update a trace with no signal that anything changed [5]. The Un-Brain never silently rewrites: a retrain snapshots the prior version, soft-deletes rather than erases, and surfaces the change for the owner to confirm (§6); contradictions are *surfaced*, not auto-resolved, inheriting the substrate’s owner-adjudicated posture. Where the brain can edit a memory outside awareness, Graphnosis edits only in front of the owner, reversibly.

Stated plainly: the four properties that make the procedural layer trustworthy — indelibility, determinism, bounded termination, owner adjudication — are exactly the four points at which it **declines to imitate biology**. The brain is the existence proof that stored knowledge can be compiled into fluent action; the Un-Brain is an attempt to keep that capability while removing the parts that make biological memory unreliable for an owner who needs it to be exact. The analogy is a lens for understanding the design, not a theory of cognition, and not a claim that the system works the way a brain does.

15.4 The Un-Ganglia: procedural failure modes the design rules out

§15.3 contrasted the Un-Brain with the failure modes of declarative *memory*. The same exercise runs one level up, on action control itself. If the basal ganglia are the brain’s action-selector and habit-sequencer (§15.1), they are also a catalogue of well-studied failure modes — and each, read purely as an *analogy*, has a structural counterpart in how the Skill Subgraph’s control flow is designed. **This table is an expository analogy and carries no experimental content.** The mapping is illustrative only: the engineering guarantees in the right-hand column constrain the *software* control flow, and we make no claim that they bear on, explain, or “rule out” the clinical or biological disorders named on the left — those are properties of neural tissue, not of a step graph. We name the disorders because they label well-understood categories of procedural failure, not to assert any correspondence beyond the analogy.

		The <i>software</i> design property the analogy points to (no claim about the disorder itself)
Brain glitch (named only as an analogy)	What goes wrong, in the brain	
OCD / compulsion [7]	A cortico–striato–thalamo–cortical loop re-fires and cannot reach a stop state	A <i>capped</i> , contract-honoring walk runs under a composed termination bound — per-loop iteration caps composed with the depth-3 recursion limit (§5) — so it cannot loop without end — a bound the shipped walker enforces in code, capping even an uncapped <code>@loop</code> at its default; only <code>parallel[]</code> dispatch lies outside the shipped path (§5, §11). The corresponding <i>software</i> failure mode is bounded by construction.
Habit capture / addiction [8]	Over-trained striatal habits drive behavior stimulus-first after the goal has moved on (model-free overriding goal-directed control)	A skill fires only when dispatch matches its trigger to the <i>live</i> goal (§7), runs under the per-step autonomy ceiling (§5, §13), and loses <i>vitality</i> and drops out of dispatch when stale (§6). Dispatch keys selection to the <i>live</i> goal, so a stale or off-goal skill is down-weighted and drops out rather than driving behaviour on its own.

Brain glitch (named only as an analogy)	What goes wrong, in the brain	The <i>software</i> design property the analogy points to (no claim about the disorder itself)
Perseveration	Striatal/prefrontal dysfunction locks behavior into a task-set the situation has outgrown	Dispatch re-selects every turn against the current goal (§3, §7) — change the goal, change the skill; there is no set to get stuck in.
Parkinson’s — under-gating [9]	Dopaminergic under-activity impairs <i>initiation</i> of an intended action (akinesia)	Selection is a deterministic read of the trigger index (§7, §10.4) — a matched, capability-feasible goal yields a runnable plan with no stochastic initiation step, though a match can still fail to fire when prerequisites or qualifying models are missing (§5).
Huntington’s — over-release [9]	Striatal degeneration releases <i>involuntary, unselected</i> movement (chorea)	The autonomy engine forbids any step from running above its authored dispatch-safe ceiling (§13) — bounding <i>unselected escalation</i> (it does not eliminate model or tool errors within a selected step).

The shape of the argument is the one the companion paper made for memory and §15.3 made for storage, and it is an *architectural analogy* throughout: the procedural cortex is most useful, on this reading, precisely where it **declines to reproduce biology**. The brain is the existence proof that stored procedures can be selected and run fluently; the failure modes above are, by analogy, the price biology pays for doing it with dopamine-modulated, hard-to-inspect, hard-to-edit circuitry. The Skill Subgraph keeps the capability — compile a procedure once, select it on cue, run it as a unit — while, in its software design, bounding its loops, gating its actions, surfacing its staleness, and editing only in front of the owner. To be explicit: it does not *treat* and makes no claim to bear on these clinical glitches; the parallel is that the corresponding *software* failure mode is designed out of the step graph. This is the sense in which the control flow is **Un-Ganglia**, the term the title carries: it keeps the basal ganglia’s selection-and-sequencing while ruling out, by construction, the *corresponding software* failure modes that make the biological version unreliable — the procedural counterpart to what *Un-Brain* named for memory.

One disorder is deliberately *absent* from the table above, because it is not an action-selection failure at all. **Alzheimer’s** and the declarative dementias are neurodegeneration of *memory*, not of the basal ganglia’s action-selection — so they belong first to the companion paper’s domain and to §15.3’s decay axis. They earn a row of their own all the same, set apart, because Alzheimer’s marks the seam where the *Un-Brain* argument and this paper’s *Un-Ganglia* layer meet:

Cross-boundary case (analogy only)	Biological dissociation	Where <i>Un-Brain</i> and <i>Un-Ganglia</i> meet
Alzheimer’s / declarative dementia [11]	Progressive declarative-memory loss can coexist with comparatively <i>spared</i> overlearned procedural skills [11] — a memory/procedure <i>dissociation</i> , not a claim that Alzheimer’s is a basal-ganglia disorder	Un-Ganglia holds procedures as separate, bounded, inspectable Skill subgraphs and makes them not merely resilient but indelible (§5, §6, §13); and because every skill grounds in the same substrate, the shared cortex keeps the <i>declarative</i> facts it binds at walk time indelible and owner-adjudicated too (§15.3; §2, §6). The brain’s dissociation shows procedure outlasting memory; the architecture inherits that ordering and then <i>removes the asymmetry</i> — neither half is left to decay.

So the dissociation cuts in the architecture’s favor twice over. Read one way, it is the brain’s own evidence that the procedural system is the more robust of the two — the observation this paper’s title leans on. Read the other, it is exactly why a design that makes *both* halves indelible does more than biology manages: where Alzheimer’s spares procedure but loses memory, the Un-Ganglia over the Un-Brain spares both. The analogy is not that Graphnosis models dementia; it is that it declines to let either declarative memory or procedural skill decay quietly.

References (this section)

- [1] Miller, E. K., & Cohen, J. D. (2001). *An integrative theory of prefrontal cortex function*. Annual Review of Neuroscience, 24, 167–202.
- [2] Graybiel, A. M. (2008). *Habits, rituals, and the evaluative brain*. Annual Review of Neuroscience, 31, 359–387. See also Graybiel, A. M. (1998). *The basal ganglia and chunking of action repertoires*. Neurobiology of Learning and Memory, 70(1–2), 119–136.
- [3] McClelland, J. L., McNaughton, B. L., & O’Reilly, R. C. (1995). *Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory*. Psychological Review, 102(3), 419–457.
- [4] Josselyn, S. A., & Tonegawa, S. (2020). *Memory engrams: Recalling the past and imagining the future*. Science, 367(6473), eaaw4325.
- [5] Nader, K., Schafe, G. E., & LeDoux, J. E. (2000). *Fear memories require protein synthesis in the amygdala for reconsolidation after retrieval*. Nature, 406(6797), 722–726.
- [6] Loftus, E. F. (2005). *Planting misinformation in the human mind: A 30-year investigation of the malleability of memory*. Learning & Memory, 12(4), 361–366.
- [7] Graybiel, A. M., & Rauch, S. L. (2000). *Toward a neurobiology of obsessive-compulsive disorder*. Neuron, 28(2), 343–347.
- [8] Everitt, B. J., & Robbins, T. W. (2005). *Neural systems of reinforcement for drug addiction: from actions to habits to compulsion*. Nature Neuroscience, 8(11), 1481–1489.
- [9] Albin, R. L., Young, A. B., & Penney, J. B. (1989). *The functional anatomy of basal ganglia disorders*. Trends in Neurosciences, 12(10), 366–375.
- [10] Crittenden, J. R., & Graybiel, A. M. (2011). *Basal ganglia disorders associated with imbalances in the striatal striosome and matrix compartments*. Frontiers in Neuroanatomy, 5, 59.
- [11] Eslinger, P. J., & Damasio, A. R. (1986). *Preserved motor learning in Alzheimer’s disease: implications for anatomy and behavior*. Journal of Neuroscience, 6(10), 3006–3009.

16. Why This Lasts, and the Doors It Opens

We have separated, throughout, what is demonstrated here from what is not. That discipline matters most in a forward-looking section, so we keep it: below, the *durable bet* is an argument from mechanisms already in the paper, and everything past it is labeled conjecture, not roadmap.

The durable bet: the unit of reuse moves down a layer. The prevailing unit of reuse in agentic systems is the prompt, and just beneath it, the model. Both churn. Prompts are re-tuned for each model generation; models are deprecated, repriced, and replaced on a cadence measured in months. A procedure compiled into a typed-edge subgraph is a different kind of object: owner-held, versioned, model-agnostic, and co-located with the memory it grounds in. Control flow is structure, not phrasing (§3); which model *runs* a given step is a separate, read-only routing decision resolved at walk time against whatever catalog exists then (§5, §7). So when a model is retired or a cheaper one appears, the procedure graph does not change — only the routing table’s answer does. Under **cost-first** routing (the adaptive default), routed cost

is non-increasing as capable models are added to the catalog (Proposition 4) — a bound about cost, not quality. (The coverage-max policy behind §10.3’s 73.3% does *not* enjoy this monotonicity — it can select a pricier, more-capable model — and that 73.3% is the complement ($1 - 0.2667$) of the Haiku/Sonnet price ratio on the pinned catalog, not a catalog-monotonic result.) A structured procedure with per-step routing therefore *survives* model turnover and *benefits* from it, where a model-specific prompt merely depreciates. That is the bet: durability comes from putting the know-how in a layer the model churn does not touch.

Proposition 4 (Catalog monotonicity of routed cost — cost-first policies only). *(Stated here, after Proposition 5 (§10.6, where the borrow accounting lives), but numbered before it; the propositions are numbered in logical, not textual, order.) Let $R(S, \mathcal{K})$ be the routed cost of walking skill S under a **cost-first** strategy (**adaptive** or **local-only**), in which each step selects $\arg \min_{m \in \mathcal{F}(s)} \text{cost}(m)$ over its feasible set. If $\mathcal{K} \subseteq \mathcal{K}'$ then $R(S, \mathcal{K}') \leq R(S, \mathcal{K})$ for every S . Proof. For each step the feasible set grows under $\mathcal{K}'$, and a minimum over a superset is no larger; summing over steps preserves the inequality. ■ (Scope, corrected after review: this holds **only** for cost-first selection. It does **not** hold for the **always-best** (coverage-max) strategy, which ranks by capability coverage before cost — adding a more capable but pricier model can be selected and **raise** routed cost. So §10.3/§10.7’s **always-best** 73.3% is not an instance of this proposition; it is the complement ($1 - 0.2667$) of the Haiku/Sonnet price ratio on the pinned catalog. Monotonicity applies to the adaptive default; it bounds cost, not quality.)*

Future systems: bounded procedures for embodied and agentic AI. The same properties that make a procedure durable make it auditable, which matters more as agents act in the world rather than only in chat. An embodied or long-running agent that walks an owner-authored, oracle-checked, bounded procedure is constrained by construction in three concrete ways already present in the substrate: termination is bounded by the iteration cap composed with the depth-3 recursion bound (§5), so a contract-honoring walk cannot run away; each step’s completion is decided by a [verify: tool|state|human] oracle (§3) rather than by the model’s own say-so; and [dispatch-safe:] gating, under the autonomy engine’s rule that autonomy is capped by authored safety (§13), keeps a step from auto-running beyond the level its author permitted. Grounding stays local-first and private at the binding, not the body (§4, §9). We do not claim to have built a robot; we claim that the determinism, oracles, and gating demonstrated here are the right primitives for one, and that an agent walking such a graph is inspectable in a way an agent improvising from a prompt is not. Whether these primitives suffice for real embodied control is conjecture we have not tested. We also restate the §11 caveat that survives: the shipped reference walker now re-executes capped loops with the bound enforced in code, but does not yet dispatch parallel siblings — for parallel plans, termination remains a property of contract-honoring executors rather than of the in-app driver.

Doors opened (conjecture, not commitments). Three possibilities follow from the artifacts rather than from ambition. First, the signed .gsk pack (§13) is *designed* as a unit of *exchange*, not just of backup: a procedure could travel between owners with its full eight-goal contract and recall recipes intact, its integrity carried by the Ed25519 signature — so portable skill packs could become a shared currency of know-how the way libraries became one for code. We note plainly that the pack’s payload cipher is obfuscation-grade and the real guarantee is the signature (§11), so an unsealed pack that travels is *verifiable*, not confidential — though §11’s optional passphrase/recipient-key sealing is now available to add confidentiality on top of the signature. Second, because the empty-engram invariant keeps *automatically-recalled* personal nodes out of the body (hand-authored details and engram names remaining the owner’s to review), a pack is shareable without leaking the cortex it was trained beside — which makes cross-domain libraries, and plausibly a commons or market of trainable SOPs, coherent rather than a privacy hazard. Third, the same invariant is what allows on-device personalization without data egress: a generic SOP binds to a private engram only at walk time, under a consent gate and local-only routing, so the procedure can be public while — under that local-only strategy, and with the privacy lock being per-step with no taint propagation (Invariant 2) — the facts never leave the machine. Each of these is enabled by a mechanism shown here; none is demonstrated at the scale of a working market or library, and we make no claim about timeline.

This paper’s contribution to that trajectory, stated modestly. We do not claim to have invented any of the layers this rests on — SOPs, skill libraries, workflow graphs, model routing, and procedural memory all predate it (§12). What this paper contributes is one worked, measured existence proof that the

conjunction holds together in a single owner-held store and earns its keep: a real corpus an agent trained and then used to build itself, with the routing savings, dispatch reduction, and vitality behavior reported under labeled caveats, and the failure case (skills dispatched but not walked, §8) kept in. If the durable bet is right, the lasting value here is less the corpus than the demonstration that procedure-as-co-located-subgraph is buildable, walkable, and cheap — a small, concrete foothold for whoever builds the embodied agents, shared libraries, and on-device personalization the mechanism makes possible. The rest is for later work, and we name it as such.

17. Conclusion

The argument of this paper is one sentence long: **a procedure belongs where memory lives**. When an SOP is compiled into a typed-edge subgraph in the same owner-held, indelible, encrypted graph that stores an agent’s declarative memory, it inherits — rather than reimplements — encryption, an operation log, snapshot versioning, and a freshness model; it can bind to private facts at walk time without baking them in; it can route each step to the cheapest capable model; and it can be kept alive by a governed retrain loop instead of rotting silently. None of the ingredients is new. The synthesis — holding all of them in one local-first indelible store — is.

The evidence is an **existence proof plus exact accounting on a single owner’s single-machine cortex — not a demonstration of transfer across users, machines, teams, or independent owners/recipients**: a 76-skill library that an AI agent trained, by reading a real codebase and its founder’s sessions, and then used to keep building that codebase — spanning engineering, business, and self-referential meta-skills, and extended to 108 skills across twelve families as a representational domain-compatibility probe (not measured cross-user transfer) — with routing savings as a **cost accounting** (the routing-cost run measures no quality; E6 separately pilots quality retention, §10.7) — a single-skill engine demonstration (§10.3) extended to a corpus-wide cost run (§10.7): ~96.6% cloud-only / ~99.6% adaptive under the cheapest-qualifying rule, the 73.3% premium-cloud figure being the complement ($1 - 0.2667$) of the Haiku/Sonnet price ratio on the pinned catalog — a ~92% lazy-dispatch reduction, a measured **~39–45× borrow-vs-spawn separation** (E4: the live single run ≈ 475 vs $\approx 21.3k$ tokens, $\approx 45\times$, corroborated by an N=5 sandbox at $39.3\times$, §10.7), and a vitality model that measurably restores freshness under retraining. Where the system promises more than it yet delivers — dormant memory augmentation, a reference walker that still defers `parallel[]` dispatch to external executors (its loop side is now enforced in code), capability tags ahead of the corpus that uses them — we have said so. *The Un-Brain* gave the owner a memory that refuses to decay or to resolve its own contradictions; the Skill Subgraph gives that memory a procedural layer that surfaces its own staleness and keeps its private bindings owner-held and local under the stated routing policy. This is its procedural companion.

References

Preprint. Related-work sources for §12 were accessed 2026-06-26; the memory-agent comparators (MemAgent, Letta/MemGPT, A-MEM) 2026-06-28.

Sources for §10.6 (cross-skill calls vs. sub-agent spawns), accessed 2026-06-28: Wang et al. (2024), *Executable Code Actions Elicit Better LLM Agents* (CodeAct), arXiv:2402.01030; OpenAI Agents SDK — agent-as-tool orchestration, openai.github.io/openai-agents-python; and recent typed skill-graph composition work, incl. SkillDAG (arXiv:2606.03056) and GraSP (arXiv:2604.17870). No prior source was found that reports a per-call token comparison of named cross-skill invocation versus sub-agent spawning; §10.6 is positioned accordingly.

Companion paper. Lazar, N. (2026). *The Un-Brain: A local-first, indelible knowledge multi-graph for private AI memory with deterministic, owner-adjudicated contradiction handling*. DOI 10.5281/zenodo.20843387.

LangGraph. Graph API, persistence, subgraphs, and recursion-limit documentation — docs.langchain.com/oss/python/langchain (overview, graph-api, use-subgraphs, persistence, GRAPH_RECURSION_LIMIT).

LangChain memory, Context Hub, and Wiki Memory. Chase, H. (2026). *Wiki Memory* — langchain.com/blog/wiki-memory (2026-06-30). Broekhuizen, J. (2026). *How To Give Your Agent Memory* — langchain.com/blog/how-to-give-your-agent-memory (2026-06-24). *How we built Agent Builder’s memory system* — langchain.com/blog/how-we-built-agent-builders-memory-system (2026-02-21). *Everything we shipped at Interrupt* (Context Hub announcement) — langchain.com/blog/interrupt-2026-overview (2026-05-14). All accessed 2026-07-03.

Microsoft GraphRAG. Documentation — microsoft.github.io/graphrag (indexing overview, model selection, CLI). Edge, D., et al. (2024). *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. arXiv:2404.16130.

LightRAG / LazyGraphRAG. Guo, Z., et al. (2024). *LightRAG: Simple and Fast Retrieval-Augmented Generation*. arXiv:2410.05779; github.com/HKUDS/LightRAG. *LazyGraphRAG: Setting a new standard for quality and cost* — Microsoft Research blog (2024).

Model routing / cascades. Ong, I., et al. (2024). *RouteLLM: Learning to Route LLMs with Preference Data*. arXiv:2406.18665; github.com/lm-sys/RouteLLM. Chen, L., Zaharia, M., Zou, J. (2023/2024). *FrugalGPT*. arXiv:2305.05176 (TMLR 2024).

Mem0. Documentation — docs.mem0.ai (memory operations, memory types). *Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory*. arXiv:2504.19413.

Zep / Graphiti. github.com/getzep/graphiti. Rasmussen, P., et al. (2025). *Zep: A Temporal Knowledge Graph Architecture for Agent Memory*. arXiv:2501.13956.

MemAgent. *MemAgent: Reshaping Long-Context LLM with Multi-Conv RL-based Memory Agent*. arXiv:2507.02259; github.com/BytedTsinghua-SIA/MemAgent.

Letta / MemGPT. Packer, C., et al. (2023). *MemGPT: Towards LLMs as Operating Systems*. arXiv:2310.08560; Letta — github.com/letta-ai/letta, docs.letta.com.

A-MEM. *A-MEM: Agentic Memory for LLM Agents*. arXiv:2502.12110; github.com/agiresearch/A-mem.

Anthropic Agent Skills. Agent Skills documentation — platform.claude.com/docs and code.claude.com/docs; *Equipping agents for the real world with Agent Skills* — anthropic.com/engineering.

Acknowledgments

The research question, the design of the procedural layer, the key tradeoffs, and the system architecture — the eight-goal Skill structure, the typed-edge subgraph co-located with owner-held memory, per-step capability routing, lazy dispatch, the vitality-driven retrain loop, and the borrowability and portability model around them — were defined and directed by the author. Implementation was AI-assisted under the author’s design specifications: the Skill Subgraph was built with **Claude Code (Anthropic)**, which also read the repository and the author’s own sessions to **train the skill corpus** documented here. During the preparation of this work the author used Claude (Anthropic) for coding, corpus training, and drafting support; the claims were then stress-tested two ways — **internal multi-agent adversarial review orchestrated with Claude Code** (fan-out finders with adversarial verification of every finding), and **independent, adversarial (“hostile”) review by GPT-5.5 (OpenAI) and Fable 5 (Anthropic) over several iterations** — each pass surfacing inconsistencies and overclaims that were corrected before submission. The author reviewed and edited all content, independently verified every reference, number, and reproducibility artifact, and takes full responsibility for the scientific claims and the final text. No AI system is an author.

Appendix A. The Corpus, Catalogued

Three populations recur in this paper and are defined once here. **P76** is the original self-built core — the population used for the dispatch decomposition (§10.2) and the vitality baseline (§10.5). **P108** is the full catalogued corpus across twelve families — the broader library. **P105** is the E2 routing subset: P108 minus three exclusions (two already-tagged skills with empty step bodies and the body-truncated **skill-dispatch**, flagged for repair, §10.7). The library **catalogued** here is thus **P108 = 108 trained skills across 12 owner-domain families**; the subset **measured for dispatch and vitality** in §10 is **P76**, the self-built core at the time of those measurements; corpus-wide routing (§10.7, E2) is over **P105**; and the routing example of §10.3 is one annotated skill within P76. As of the migration that accompanied this writing, each family is now its own engram rather than a shared store. This reorganization makes one claim of the paper concrete at the storage layer. **A skill engram is not a folder of prompts.** It is the conjunction of four co-located things: the *procedures* (the typed-edge subgraphs of §3), the *private memory they bind at walk time* (recall recipes resolved against the owner’s cortex, §5), the *dispatch index* that decides which procedure surfaces when (§7), and the *cross-skill call graph* that lets them compose like functions (§3, §5). Held together in one owner-held, encrypted, indelible engram, that conjunction is the persistent, model-agnostic **core of a domain agent**: the model that runs a step is rented and swappable (§5, §10.3), but the engram-agent — its procedures, its bindings, its routing, its call graph — is owned and durable.

One caveat, stated precisely so the framing is not over-read: an engram is the agent’s *core*, not a running loop. It does not act on its own. It acts when a runtime *walks* it (§5), and how far that runtime proceeds without a human is governed by the L0–L3 autonomy axis (§13) — capped, never exceeded, by each skill’s authored safety. The engram is the durable substance; autonomy is a separate dial laid over it.

The twelve families

Family	Engram	Count	Completion oracle	Purpose
Code / engineering	Code Skills	26	[verify: tool\ state]	Ship, test, diagnose, and audit the codebase against machine checks
Business / GTM	Business & GTM Skills	25	[verify: human]	Go-to-market, sales, pricing, fundraising — judgment work bound to private engrams at walk time
Meta / agentic	Meta Skills	24	mixed (tool\ state\ human)	Operate on the skill system itself: dispatch, retrain, retrospectives, delegation
Research	Research Skills	4	[verify: human]	Literature, citation, experiment, and publication workflows
Creative	Creative Skills	4	[verify: human]	Music composition, release, arrangement review, and diagram/layout review
Nonprofit / board	Nonprofit Skills	3	[verify: human]	Board prep, grant drafting, fundraising campaign planning
Heritage / genealogy	Heritage Skills	3	[verify: human]	Source triage, lineage verification, archive ingestion

Family	Engram	Count	Completion oracle	Purpose
Event organizer	Event Organizer Skills	3	[verify: human]	Event concept/budget, public launch, sponsor and vendor negotiation
Film producer	Film Producer Skills	3	[verify: human]	Script development, greenlight packaging, teaser launch
Acting	Acting Skills	3	[verify: human]	Role prep, self-tape auditions, public persona and promotion
Reviewer	Reviewer Skills	3	[verify: human]	Manuscript review, publication-readiness audit, peer-review response
Ghampus (chat agent)	Ghampus Skills	7	mixed (human\ state)	The chat agent's own operating skills — turn protocol, recall routing, in-chat dispatch, memory capture, context hygiene, local-model grounding, consent & autonomy

Code / engineering (26) — sidecar-change-verify, recall-before-coding, bug-investigation, vibe-coding-workflow, generated-artifact-freshness-check, binary-output-verification, ux-decision-gate, ux-review-checklist, performance-regression-check, testing-cadence, security-review-cadence, ship-workflow, multi-repo-release-coordination, dependency-update-protocol, it-architecture-review, deployment-platform-ops, docs-maintenance-workflow, overlay-triage, consistency-audit, changelog-management, localization-awareness, app-accessibility-check, feature-showcase-sweep, praxis-configuration, runtime-diagnosis, public-repo-ship-audit.

Business / GTM (25) — feature-impact-assessment, enterprise-compliance-lens, business-strategy-review, enterprise-sales-prep, compliance-requirements-tracking, customer-support-pattern-recall, enterprise-account-crm, user-feedback-ingestion, release-announcement-draft, investor-update-preparation, product-ideation, legal-review-checklist, partnership-opportunity-assessment, integration-roadmap-planning, go-to-market-planning, content-creation-workflow, enterprise-gtm-compliance-angle, automation-opportunity-pricing-strategy-review, community-engagement, website-copy-workflow, product-trajectory-review, campaign-run-management, fundraising-readiness, compliance-evidence-pack-ops.

Meta / agentic (24) — agentic-communication-style, task-todo-management, subagent-management, mcp-tool-selection, prompt-improvement, agent-delegation-decision, session-end, self-driving-session, skill-dispatch, project-context-management, session-start, skill-maintenance-review, retrospective-learning, adaptive-skill-creation, autonomous-decision-authority, proactive-feedback-protocol, decision-challenge-protocol, strategic-conversation-trigger, cortex-gardening, dispatch-export-sync, execution-architect, batch-completion-report, skill-retrain-queue-ops, ghampus-operator.

Research (4) — experiment-design-protocol, citation-audit, literature-review-protocol, research-publishing-workflow.

Creative (4) — creative-arrangement-review, music-release-workflow, composition-workflow, diagram-layout-review.

Nonprofit / board (3) — fundraising-campaign-plan, grant-proposal-drafting, board-meeting-prep.

Heritage / genealogy (3) — archive-ingestion-workflow, lineage-verification, source-triage-protocol.

Event organizer (3) — event-concept-and-budget, event-public-launch-campaign, sponsor-and-vendor-negotiation

Film producer (3) — script-development-protocol, production-greenlight-package, teaser-launch-campaign.

Acting (3) — role-breakdown-and-prep, self-tape-audition-workflow, public-persona-and-promo.

Reviewer (3) — manuscript-review-protocol, publication-readiness-audit, peer-review-response.

Ghampus / chat agent (7) — conversational-turn-protocol, chat-recall-routing, in-chat-skill-dispatch, chat-memory-capture, context-window-hygiene, local-model-grounding, consent-and-autonomy-in-chat.

Corpus-wide properties

The per-step routing result of §10.3 is an **engine demonstration on one annotated nine-step skill** (ghampus-operator); §10.7 extends it to a corpus-wide **cost accounting** (105 skills, no output quality measured) — ~96.6% cloud-only / ~99.6% adaptive under the cheapest-qualifying rule, with the 73.3% premium-cloud figure the complement ($1 - 0.2667$) of the pinned-catalog Haiku/Sonnet price ratio. On the single skill: driving the real router saves **99.5%** (measurement-time catalog) against the all-Sonnet-4.6 baseline when local models are available (eight of nine steps run free and local — the *measurement-time* catalog; under the shipped, pinned catalog all nine route local, §10.3; the reproducible headline is the E2 corpus-wide cost accounting above), and **73.3%** in a premium cloud-only configuration where the router meets the declared capabilities with Haiku rather than Sonnet. Lazy dispatch (§10.2) is accounted over the measured P76 corpus: a session hydrates only the one-to-two skills it matches, loading roughly **8%** of the library — a **91.5%** token reduction against inlining every P76 body — and the ratio tends to strengthen as the library grows from 76 toward 108 (so long as matched skills stay a small fraction), because a matched session’s footprint is independent of how many skills it does *not* load. On freshness, the 76-skill core reported healthy vitality at measurement time (§6, §10.5): every skill in P76 *fresh*, and a clean re-compile of the two lowest moved them 92→98 and 94→99 by resetting the age penalty and clearing most of the superseded-node staleness. (The vitality baseline is P76, not the full P108.)

The per-family migration also produced a **qualitative** before/after for the **empty-engram train invariant** (§4) — exact contaminated/clean counts were not recorded, so this is a production observation, not a measured rate. Skills trained *before* the empty-engram contract was enforced carried cortex-recall dumps — frozen personal-memory pastes — baked directly into their step bodies, and these had to be scrubbed during this migration. Skills trained *under* the contract were clean in the cases inspected: no personal node had entered a body at training time, so there was nothing to scrub. This is a finding, not a claim of merit — the contaminated bodies are exactly the data-quality debt flagged in §11 — but it is the kind of before/after the invariant predicts: the contract is what separated the portable skills from the ones that had absorbed private memory.

The five by-construction properties, restated over the corpus

The five formal claims of §1.1 are the guarantees the catalogued corpus is built to satisfy:

1. **Deterministic plan compilation** — the plan compiler is a pure read of immutable node order, golden-test-pinned, so a skill’s plan is deterministic (byte-identical for fixed graph state) and cannot recurse on a loop back-edge (§5).
2. **Bounded plan contract for capped loops** — for any skill whose loops carry a finite `max=M` cap (composed with the depth-3 recursion bound), a *contract-honoring* executor halts. This bounds the contract, and — with loop re-execution now enforced in code by the shipped walker (§11) — the shipped system too for non-parallel skills; an uncapped `@loop` lies outside the contract bound but is held to the walker’s default cap (§5).
3. **Structure-preserving training** — a token-extraction validator reverts any optional rewrite that drops a DSL token, `@needs` tag, `only_engrams` binding, or goal header, so a known structural token cannot be silently *omitted* — a set-presence guarantee, not per-step placement (§4).
4. **Cross-skill-edge stability under retrain** — in-place retraining reuses each skill’s source id, and because a `@skill:` call resolves by *name* to that id (nothing is re-pointed, and a cross-engram call never becomes a graph edge — it is a side-table lookup, §3), the cross-family call graph stays resolvable across retrains without touching the caller (§6, Invariant 3).

5. **The empty-engram train invariant** — no *automatically-recalled* personal node content enters a skill body at training time; personal grounding is deferred to walk-time recall bindings, which is what makes every skill trained under the empty-engram contract a portable, source-grounded artifact (§4).

Beyond the guarantees, what the corpus demonstrates. The five above are formal claims the corpus is built to *satisfy*; the corpus also *demonstrates* — empirically, in §10 — what those guarantees were for. It exercises cross-family skill borrowing: a sibling `@skill:` call resolving across engrams through the training-time side-table (§3), one Graphnosis agent reaching another’s procedure without the call ever becoming a graph edge — **resolvable by construction** (side-table + Invariant 3), though no single named cross-family borrow walk is separately measured in Route B. And it delivers the measured economics: a sibling borrow avoids a sub-agent’s warm-up — a **constant-factor** saving, $\approx 39\text{--}45\times$ **measured** across two datapoints (the live single run ≈ 475 vs $\approx 21.3\text{k}$ tokens, $\approx 45\times$, corroborated by a reproducible $N=5$ sandbox at $39.3\times$, §10.7) and $\sim 30\times$ against the ~ 700 -token modeled typical borrow; the per-skill $O(N)$ slope is *modeled*, *not measured*, so this is a warm-up gap, not an asymptotic law; per-step routing saves **99.5%** local-paired on one skill (§10.3, measurement-time catalog; the reproducible corpus figure is E2’s $\sim 99.6\%/96.6\%/73.3\%$, §10.7) and, corpus-wide, $\sim 96.6\%$ cloud-only under the cheapest-qualifying rule or **73.3%** under coverage-max (the complement of the pinned-catalog Haiku/Sonnet price ratio, $1 - 0.2667$) — cost only (§10.7); and a dispatched session loads roughly **8%** of the library, a $91.5\text{--}92.6\%$ reduction against inlining every body (§10.2). These are implemented capabilities and token accountings — not task-outcome results, and not formal propositions — what the corpus does once the guarantees hold, not the guarantees themselves.