OpenClawBrain: Learned Graph Traversal for Retrieval Routing

Jonathan Gu jonathangu@gmail.com

February 2026

Version: 11.2.1

1 Abstract

OpenClawBrain is a learned retrieval graph for LLM agents where workspace chunks are nodes and document transitions are weighted edges. OpenClawBrain augments vector-seeded retrieval with trajectory-level learning: it updates edge confidence from full-path outcomes using corrected policy gradients. It provides value on two axes: (1) retrieval accuracy through graph-structured traversal over co-occurrence edges between related chunks, and (2) context efficiency through trajectory-level learning. In simulation, active context drops from 30 nodes on the first deployment query to 2.7 nodes on average in the last 10 queries (91% reduction). In production, on-demand routing supplies 3–13KB per query from 52–66KB baseline startup context. The results in this paper are constrained to 15 deterministic simulations, 236 local tests, and the reproducible benchmark harness.

2 1. The Problem

Repeated workspace navigation can leak repeated context and amplify cost. Most cost comes from one pattern: each query reloading too much workspace context. A common failure mode is over-retrieval: each query re-injects a large fraction of workspace context, increasing latency and token cost. This hurts retrieval quality as well as budget. *By "the graph is the prompt," we mean that traversal over the learned edge-weighted graph selects the subset of context for the expensive read stage.*

Objective. Given a query q and graph G, the traversal policy π_W selects a candidate subset of nodes $F \subseteq V$ for expensive reads. We minimize expected context tokens with success-constrained control:

$$\min_W \ \mathbb{E}_{q\sim\mathcal{D}}[\text{tokens}(F)] \quad \text{s.t.} \quad \mathbb{E}_{q\sim\mathcal{D}}[\tilde{z}(q)] \geq \rho,\ |F| \leq B.$$

Let $\tilde{z} \in \{0,1\}$ denote success and define the signed outcome $z = 2\tilde{z}-1 \in \{-1,+1\}$. In practice, this objective is implemented by signed-outcome updates with baseline control. In a deploy flow, *check CI → inspect manifest → rollback → verify*, similarity retrieval groups all four steps as related, not as ordered. It also fails on conflict: when both "run tests before deploy" and "skip tests for hotfix" are close in vector space, RAG can still surface the wrong instruction. OpenClawBrain augments seeding (vector search or heuristics) with learned traversal; it is not a standalone retrieval method without an entry mechanism.

3 2. The Graph

OpenClawBrain stores memory as a directed graph. Nodes hold chunks; edges represent possible next reads.

```
from dataclasses import dataclass, field
from typing import Any
@dataclass
class Node:
    id: str
    content: str
    summary: str = ''
    metadata: dict[str, Any] = field(default_factory=dict)
@dataclass
class Edge:
    source: str
    target: str
    weight: float = 0.5
    kind: str = 'sibling'
    metadata: dict[str, Any] = field(default_factory=dict)
```

Queries first get seed nodes by embedding search. Let $\mathcal{S}(q) \subseteq V$ be those seeds, and let F be the fired set for the current episode. A router then follows outgoing edges for a bounded number of hops. An expensive model reads only fired nodes. Traversal terminates when any of three conditions is met: the hop ceiling, the fired-node budget, or the context-character budget. Edge damping attenuates within-episode reuse; budgets provide hard guarantees. Edge tiers decide cost

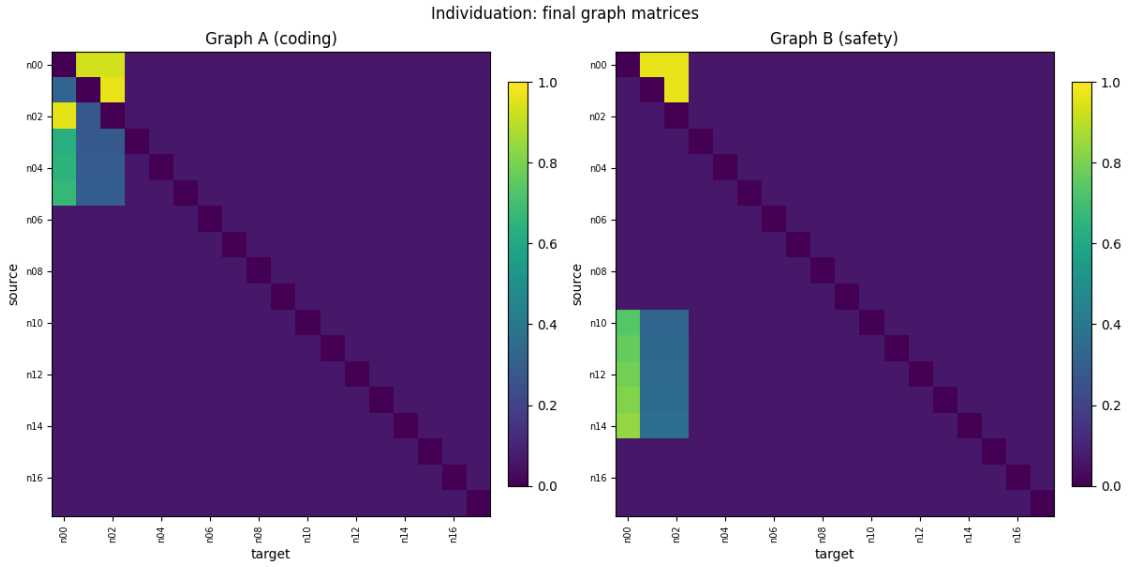

Figure 1: Different workloads produce individualized confidence patterns.

and automaticity.

$$\text{If } w \leq \theta_{\text{inh}} = -0.01, \text{ then inhibitory overrides other tiers.}$$

$$\text{Otherwise } w \geq 0.6 \Rightarrow \text{reflex}, \; 0.2 \leq w < 0.6 \Rightarrow \text{habitual}, \text{ else dormant.}$$

Dormant includes all non-actionable edges not covered by the precedence above.

Tier updates move edges among discrete regimes.

Tier	Weight Range	Behavior	Cost
Reflex	≥ 0.6	Auto-follow	Near zero
Habitual	$0.2 \leq w < 0.6$	LLM chooses follow/skip	Low
Inhibitory	≤ -0.01	Suppresses targets	Near zero
Dormant	< 0.2	Hidden unless seeded	Zero

Table 1: Edge-tier behavior and cost.

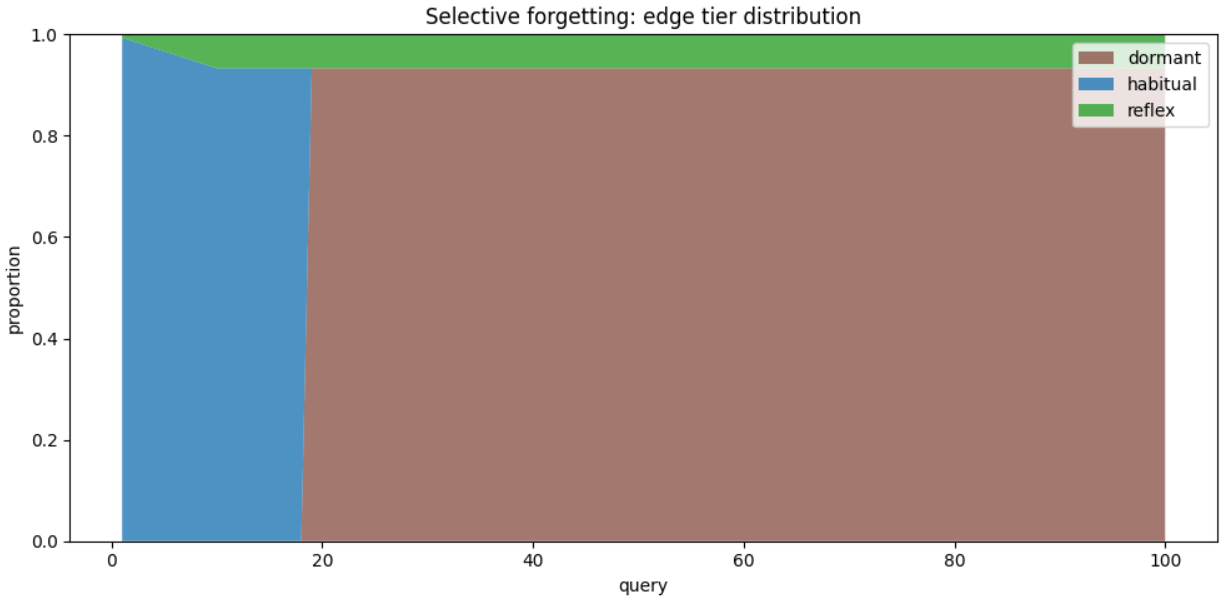

Figure 2: Tier behavior over time.

2.1 Traverse API

The router interface is explicit:

$$\texttt{route\_fn(query, candidate\_ids)} \rightarrow \text{subset of } \textit{candidate\_ids}.$$

It may use query-local cues and is optional in low-cost routes.

```
def _tier(weight, config):
    if weight <= config.inhibitory_threshold:
        return "inhibitory"
    if weight >= config.reflex_threshold:
        return "reflex"
    if config.habitual_range[0] <= weight < config.habitual_range[1]:
        return "habitual"
    return "dormant"
def traverse(graph, seeds, config, route_query=None):
    route_fn = getattr(config, "route_fn", None)
    frontier = sorted(seeds, key=lambda s: s[1], reverse=True)[: config.beam_width]
    frontier = [(node_id, score) for node_id, score in frontier if graph.get_node(node_id)]
    seen = {node_id: 1 for node_id, _ in frontier}
    used_edges = {}
    fired = [node_id for node_id, _ in frontier]
    steps = []
    for _ in range(config.max_hops):
        raw = []
```

```
        suppressed = set()
        for source_id, source_score in frontier:
            for target_node, edge in graph.outgoing(source_id):
                tier = _tier(edge.weight, config)
                if tier == "dormant":
                    continue
                if tier == "inhibitory":
                    suppressed.add(target_node.id)
                    continue
                use_count = used_edges.get((source_id, target_node.id), 0)
                effective_weight = edge.weight * (config.edge_damping ** use_count)
                if effective_weight <= 0:
                    continue
                raw.append((source_id, target_node.id, source_score * effective_weight, tier))
        if suppressed:
            raw = [r for r in raw if r[1] not in suppressed]
        if not raw:
            break
        habitual = [r for r in raw if r[3] == "habitual"]
        reflexive = [r for r in raw if r[3] == "reflex"]
        selected = []
        if route_fn is not None and habitual:
            allowed = set(route_fn(route_query, [r[1] for r in habitual]))
            selected.extend(r for r in habitual if r[1] in allowed)
        else:
            selected.extend(sorted(habitual, key=lambda x: x[2], reverse=True))
        selected.extend(sorted(reflexive, key=lambda x: x[2], reverse=True))
        next_frontier = []
        seen_next = set()
        for source_id, target_id, score, tier in sorted(selected, reverse=True)[: config.beam_width]:
            if target_id in seen_next:
                continue
            if target_id not in seen:
                seen[target_id] = 1
                fired.append(target_id)
            else:
                seen[target_id] += 1
            used_edges[(source_id, target_id)] = used_edges.get((source_id, target_id), 0) + 1
            steps.append((source_id, target_id, tier, score))
            next_frontier.append((target_id, score))
            seen_next.add(target_id)
        frontier = next_frontier
    return TraversalResult(fired=fired, steps=steps, context="")
```

Negative edges remain visible as inhibitory evidence and suppress wrong routes directly. During traversal, inhibitory targets are removed from candidate expansion if they are connected by any edge with weight ≤ -0.01. This is a hard-veto arbitration rule. It reduces recurrence of known-bad transitions and can increase false negatives with noisy outcomes.

4 3. How It Learns

Each query creates an episode with route state, action choices, and binary outcome. Let $\tilde{z} \in \{0, 1\}$ denote success and $z = 2\tilde{z} - 1 \in \{-1, +1\}$.

3.1 Policy Gradients over Routes

Let $\mathcal{A}(i) = \mathcal{N}(i) \cup \{\texttt{STOP}\}$ at node i, signed outcome z, and baseline b.

$$\pi_W(a \mid s = i) = \frac{\exp\big((r_{ia} + w_{ia})/\tau\big)}{\sum_{j \in \mathcal{A}(i)} \exp\big((r_{ij} + w_{ij})/\tau\big)}$$

Logits and derivation. Define logits

$$\ell_{ij} \coloneqq \frac{r_{ij} + w_{ij}}{\tau}, \qquad Z_i = \log \sum_{k \in \mathcal{A}(i)} \exp(\ell_{ik}).$$

$$\log \pi_W(a \mid s = i) = \ell_{ia} - Z_i.$$

For chosen action a:

$$\frac{\partial \ell_{ia}}{\partial w_{ia}} = \frac{1}{\tau}, \qquad \frac{\partial Z_i}{\partial w_{ia}} = \frac{\pi_W(a \mid i)}{\tau}$$

so

$$\frac{\partial \log \pi_W(a \mid s = i)}{\partial w_{ia}} = \frac{1}{\tau}(1 - \pi_W(a \mid i)).$$

For any non-chosen action $j \neq a$:

$$\frac{\partial \ell_{ia}}{\partial w_{ij}} = 0, \qquad \frac{\partial Z_i}{\partial w_{ij}} = \frac{\pi_W(j \mid i)}{\tau}$$

so

$$\frac{\partial \log \pi_W(a \mid s = i)}{\partial w_{ij}} = -\frac{\pi_W(j \mid i)}{\tau}.$$

$$\nabla_{\mathbf{w}_i} \log \pi_W(a \mid i) = \frac{1}{\tau}(\mathbf{e}_a - \boldsymbol{\pi}_i).$$

In the unconstrained formulation, per-node updates sum to zero and redistribute mass within each node.

3.2 Why Myopic Updates Fail

Myopic updates only credit the final edge before a terminal outcome:

$$\Delta W \propto z \nabla_W \log \pi_W(a_t \mid s_t)$$

That drops responsibility when early edges cause the final mistake. The corrected estimator from Gu (2016) sums all steps in the path:

$$\Delta W = \eta z \sum_{\ell=0}^{T} \nabla_W \log \pi_W(a_\ell \mid s_\ell)$$

and with baseline and discount:

$$\Delta W = \eta (z - b) \sum_{\ell=0}^{T} \gamma^\ell \nabla_W \log \pi_W(a_\ell \mid s_\ell)$$

3.3 Full REINFORCE Update

$$\Delta w_{s_\ell, j} = \frac{\eta (z - b) \gamma^\ell}{\tau} \big(\mathbf{1}[j = a_\ell] - \pi_W(j \mid s_\ell) \big)$$

This updates all outgoing edges at each visited node s_ℓ. Positive outcomes push mass toward the chosen edge and away from competitors; negative outcomes do the reverse. The updates sum to zero.

$$\sum_{j \in \mathcal{A}(s_\ell)} \Delta w_{s_\ell, j} = 0.$$

Numerical example. Consider a node with three outgoing edges plus STOP, all with $r = 0$, current logits $(0.5, 0.3, -0.2, 0.0)$, $\tau = 1$, and $\eta = 0.1$.

$$\pi(A \mid i) = 0.342,\ \pi(B \mid i) = 0.280,\ \pi(C \mid i) = 0.170,\ \pi(\text{STOP} \mid i) = 0.208.$$

For $\gamma^0 = 1$, $b = 0$, and $z = +1$ where A is chosen:

$$\Delta w_{iA} = +0.066,\ \Delta w_{iB} = -0.028,\ \Delta w_{iC} = -0.017,\ \Delta w_{i\texttt{STOP}} = -0.021,\ \sum \Delta w = 0.$$

For $z = -1$:

$$\Delta w_{iA} = -0.066,\ \Delta w_{iB} = +0.028,\ \Delta w_{iC} = +0.017,\ \Delta w_{i\texttt{STOP}} = +0.021.$$

The heuristic updates only the chosen edge, whereas true PG updates all edges at the node.

3.4 Corrected Code

```
def _softmax_probs(graph, node_id, temperature):
    import math
    outgoing = graph.outgoing(node_id)  # [(target_node, edge), ...]
    logits = {}
    for target_node, edge in outgoing:
        target_id = getattr(target_node, "id", target_node)
        relevance = float(edge.metadata.get("relevance", 0.0))
        logits[target_id] = (relevance + edge.weight) / temperature
    logits["STOP"] = 0.0
    max_logit = max(logits.values())
    exp_sum = sum(math.exp(v - max_logit) for v in logits.values())
    return {k: math.exp(v - max_logit) / exp_sum for k, v in logits.items()}


def apply_outcome_pg(graph, fired_nodes, outcome, config=None, baseline=0.0):
    config = LearningConfig() if config is None else config
    updates = {}
    z = 2.0 * float(outcome) - 1.0
    advantage = z - baseline
    if len(fired_nodes) < 2:
        return updates
    for step_idx in range(len(fired_nodes) - 1):
        source_id = fired_nodes[step_idx]
        chosen = fired_nodes[step_idx + 1]
        probs = _softmax_probs(graph, source_id, config.temperature)
        scalar = config.learning_rate * (config.discount ** step_idx) * advantage / config.temperature
        for target_node, edge in graph.outgoing(source_id):
            target_id = getattr(target_node, "id", target_node)
            is_chosen = 1.0 if target_id == chosen else 0.0
            delta = scalar * (is_chosen - probs.get(target_id, 0.0))
            edge.weight = _clip_weight(edge.weight + delta, config.weight_bounds)
            graph._edges[source_id][target_id] = edge
            updates[f"{source_id}->{target_id}"] = delta
    return updates
```

3.5 Why Not AlphaGo Search

AlphaGo uses value-function and tree-search budgets. OpenClawBrain receives one real query outcome per turn and must stay cheap. Corrected policy-gradient credits over full trajectories fits this cost model: learn from real outcomes, not from millions of simulated futures.

5 4. How It Grows

Version 11.2.0 keeps growth intentionally short: bootstrap, decay, and damping. Two v2 pathways are removed: explicit synaptogenesis and neurogenesis.

AlphaGo-style loop	OpenClawBrain
Millions of simulated futures per action	One real outcome per query
Massive replay cost	Online path credit from lived trajectories
Value approximation target	Policy-routing edge updates

Table 2: Why corrected trajectory gradients fit low-cost agents.

4.1 Bootstrap

Bootstrap creates nodes by chunking files and initial sibling edges within each file. Three production brains run on a Mac Mini M4 Pro, each built from workspace markdown files, session replay, and learning database entries:

- MAIN: 1,160 nodes, 2,551 edges, 43 learnings (OpenAI embeddings)
- PELICAN: 555 nodes, 2,211 edges, 181 learnings (OpenAI embeddings)
- BOUNTIFUL: 289 nodes, 1,101 edges, 35 learnings (OpenAI embeddings)

All nodes have real OpenAI text-embedding-3-small (1536-dim) vectors. All three brains were rebuilt February 27, 2026. Existing embeddings are cached and reused across rebuilds.

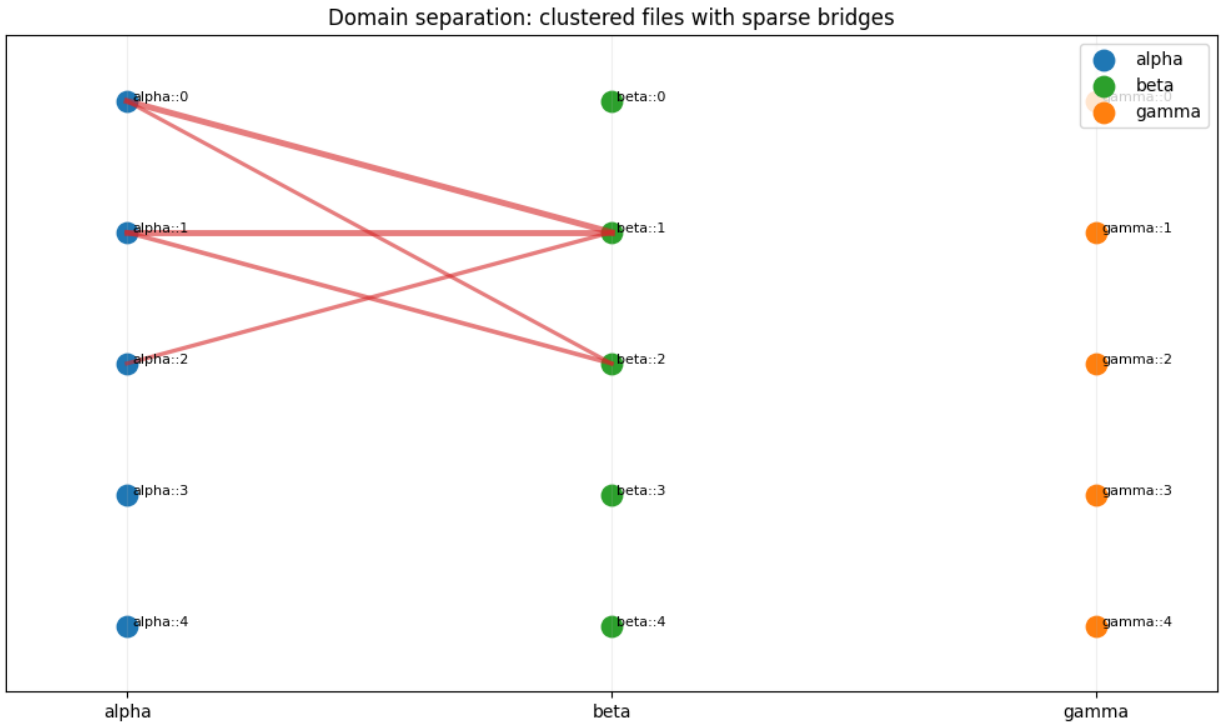

Figure 3: Bootstrapped graph from workspace chunks.

4.2 Decay (Dampened Dormancy)

Unused edges decay geometrically.

$$\tilde{w} = w \cdot \left(\frac{1}{2}\right)^{\Delta q/h}$$

where Δq is queries since last traversal and h is half-life.

4.3 Edge Damping

Within one trajectory, repeated traversals of the same directed edge lose immediate weight:

$$\tilde{w} = w \cdot \lambda^k$$

with k repeats in this episode.

4.4 Update Rule

```
def apply_decay(graph, config=DecayConfig(half_life=80)):
    for source_id in list(graph._edges.keys()):
        for target_id, edge in list(graph._edges[source_id].items()):
            edge.weight = _clip_weight(edge.weight * (0.5 ** (1 / config.half_life)), config.weight_bounds)
            graph._edges[source_id][target_id] = edge
```

In simulation, $\lambda = 0.3$ broke short loops while still discovering targets.

λ	Trajectory	Outcome
0.3	A → B → C → D	Reached target in 4 hops
1.0	A → B → C → A	Cycled

Table 3: Route damping prevents cyclic lockup.

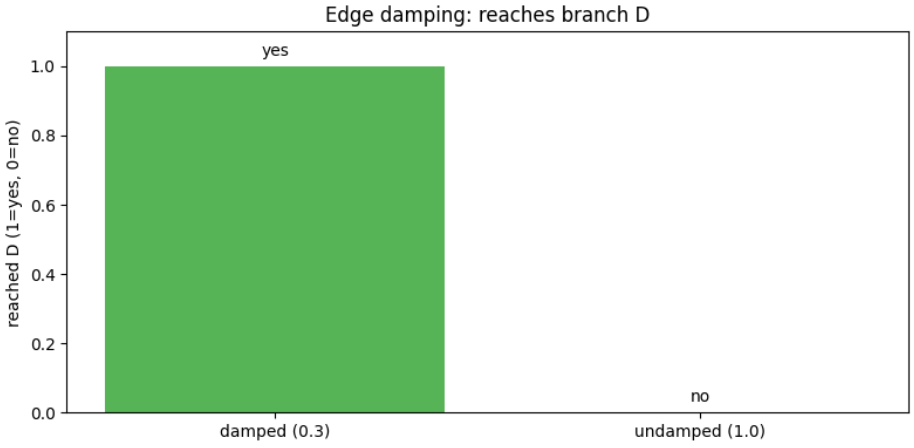

Figure 4: Damping keeps loops from running forever.

6 5. How It Stays Healthy

Adaptive retrieval requires bounded control.

5.1 Health Metrics

Metric	Target range	Function
`avg_nodes_fired_per_query`	1.0–2.0	Cost/coverage tradeoff
`cross_file_edge_pct`	0%–15%	Specialization with bridges
`dormant_pct`	70%–95%	Noise suppression
`reflex_pct`	0%–10%	Safety against over-automation
`context_compression`	$\leq 20\%$	Bounded tokens per query
`proto_promotion_rate`	0%–50%	Discovery without churn
`reconvergence_rate`	0%	Avoid sibling collapse
`orphan_nodes`	0	Full reachability

Table 4: Health metrics and target ranges.

5.2 Autotuner

```
health = measure_health(graph, state, query_stats)
adjustments = autotune(graph, health)
changes = filter_with_guardrails(adjustments)
apply_adjustments(graph, changes)
```

Safety bounds remain explicit: decay half-life [20, 200], promotion threshold ≥ 2, Hebbian increment ≤ 0.12, reflex threshold [0.6, 0.95], edge damping [0.1, 1.0].

5.3 Brain-Death Stress

Severe decay can push all edges toward dormancy. The autotuner partially recovers with repeated health feedback.

Round	Dormant	Habitual	Reflex
1	100.0%	0.0%	0.0%
10	97.5%	1.3%	1.3%
30	97.5%	0.0%	2.5%

Table 5: Brain-death recovery behavior.

5.4 Migration and Session Replay

New machines or revisions rerun bootstrap and optionally replay session logs for weight warm-start.

Session replay is a warm-start path for a new graph. The system reads historical session logs, extracts user queries, and runs each through the graph to seed, traverse, and apply outcome updates. No LLM is required for replay to restore traversal structure. Replay can warm-start graph structure before live traffic when the graph lacks history. v9.3.0 adds incremental replay via `last_replayed_ts`: only new sessions since the last replay are processed, avoiding redundant traversals. The system also extracts assistant responses via `extract_interactions`, enabling outcome-weighted replay where corrections receive negative feedback and acknowledged queries receive positive feedback.

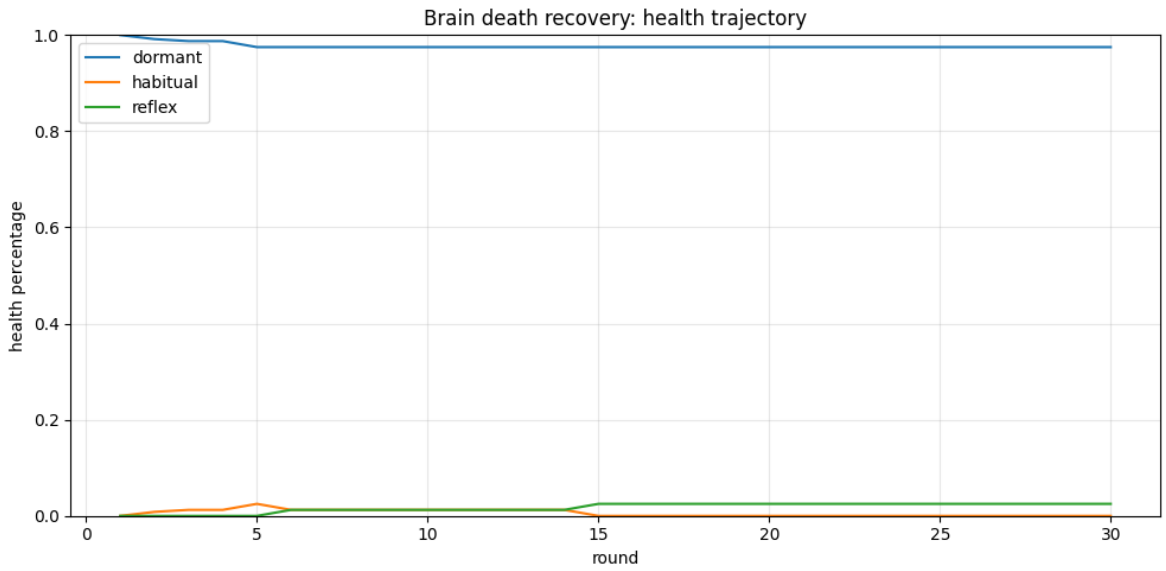

Figure 5: Autotuner interventions under severe forgetting pressure.

5.5 Learning Database Integration

OpenClawBrain nodes are not limited to file chunks. The OpenClaw integration injects learning database entries — corrections, directives, and teachings harvested from human feedback — as first-class graph nodes with real embeddings. Each learning node carries its error class, prevention rule, recurrence count, and source text. CORRECTION nodes create inhibitory edges, while TEACHING and DIRECTIVE nodes add positive knowledge without inhibitory edges. This means the graph can retrieve not just workspace content but also *how to avoid past mistakes*. In production, 259 active learning entries from three agents are embedded and queryable alongside workspace chunks.

5.6 Operational Tooling

`openclawbrain doctor` validates state integrity: Python version, state file validity, node/edge counts, embedder metadata, dimension consistency, and journal writability. `openclawbrain info` reports graph statistics: node count, edge count, embedder name/dim, tier distribution percentages, and state file size. A per-brain journal system logs queries, learning events, and health snapshots for operational observability.

5.7 Production Lessons

Across multiple months of production operation, the following lessons surfaced:

1. **Embedder metadata corruption.** The CLI save path silently overwrote OpenAI embedder metadata with hash-v1 defaults. Every learn or replay command degraded the state. Fix: `save_state` now checks the meta dict for embedder info before falling back to defaults (v9.3.1).

2. **Orphaned correction nodes.** Injecting learning corrections as graph nodes without connecting them to workspace nodes produced orphans unreachable by traversal. Fix: connect each correction to its top-3 most similar workspace nodes via cosine similarity (v9.2.0).

3. **Batch overflow.** Sending all texts to OpenAI in a single embedding call hit the 300K token API limit at 1,635 texts. Fix: chunk into batches of 100 (v9.3.0).

4. **Context savings measurement (Axis 2).** Before OpenClawBrain, agent session startup loaded 52–66KB of workspace files. After migration, on-demand routing supplies 3–13KB per query, and the reduction came from moving reference material (checklists, error class maps, design guides) into the graph while keeping safety rules always-injected.

5. **Recalibrated tier thresholds.** Production telemetry across three brains (5,863 edges total) moved defaults to `reflex_threshold` $= 0.6$, `habitual_range` $= 0.2 \ldots 0.6$, and `inhibitory_threshold` $= -0.01$ so inhibitory weights near -0.035 now suppress downstream traversal.

6. **Inhibitory edges require matching dimensions.** The inject module validates that new vectors match the index dimension. Hash embeddings (1024-dim) cannot create meaningful connections in an OpenAI index (1536-dim). This is now a hard error.

7. **Live correction pipeline.** Corrections reach the graph within 2 hours: harvester ingests from sessions, LLM classifies, SQLite stores, then `inject_batch` embeds new nodes, connects them, and creates inhibitory edges. No manual rebuild required.

8. **Real-time correction flow.** `query_brain.py --chat-id` persists fired nodes per conversation, enabling same-turn `learn_correction.py` flows. A content-hash ledger and batch harvester prevent duplicate injections. This moved end-to-end insertion from fixed 2-hour batch windows toward same-turn correction when needed.

5.8 Real-time Correction

The real-time pipeline adds per-conversation persisted fires and immediate correction hooks. `query_brain.py --chat-id` stores fired nodes in `fired_log.jsonl` with rolling retention. `learn_correction.py` now applies outcomes directly against logged paths using `apply_outcome_pg`, while a background batch harvester still supports delayed ingestion. This avoids duplicate repair through a content-hash ledger and preserves the same correction semantics as manual rebuild-free workflows. `learn_correction.py` shortens practical latency from 2-hour batch updates toward same-turn correction in active sessions.

1. Fired-node persistence is per conversation.
2. Same-turn injection is supported via `learn_correction.py`.
3. Batch harvester deduplicates by content hash ledger.
4. Latency path moves from 2-hour-only to same-turn when invoked in flow.

5.9 Two-Timescale Architecture

OpenClawBrain separates online adaptation from structural maintenance.

Online learning (fast loop). Each query executes `traverse()` with existing edge weights, logs the trace, receives outcome feedback, and applies per-query updates through `apply_outcome()` or `apply_outcome_pg()`.

$$\texttt{query} \rightarrow \texttt{traverse} \rightarrow \texttt{log\_trace} \rightarrow \texttt{feedback} \rightarrow \texttt{learn}$$

Maintenance (slow loop). Structural upkeep is periodic and asynchronous to requests. `run_maintenance()` is scheduler-agnostic and composes health, decay, merge, prune, and compact stages in a single entry point.

Persistent worker. For production integration, `openclawbrain daemon` starts a long-lived process that loads `state.json` once and accepts JSON-RPC requests over stdin/stdout. This eliminates per-call state reload (typically 100–800ms for large brains). Production timing on Mac Mini M4 Pro with OpenAI embeddings:
MAIN (1,158 nodes) completes queries in 504ms total (397ms embedding API + 107ms traversal);
BOUNTIFUL (285 nodes) in 431ms.

$$\texttt{health} \rightarrow \texttt{decay} \rightarrow \texttt{merge} \rightarrow \texttt{prune} \rightarrow \texttt{compact}$$

Constitutional anchors. Node metadata assigns maintenance authority:

- `constitutional`: never decay, never prune, never merge.
- `canonical`: half-life decay at 2x scale, merges only with explicit confirmation.
- `overlay` (default): normal maintenance behavior.

Context lifecycle. Operational context flow is:

`Files` → `Sync` → `Graph` → `Maintain` → `Compact`

Sync refreshes embeddings, Graph applies learning, Maintain runs maintenance passes, and Compact migrates low-value historical context out of files.

7 6. Results

The benchmark harness compares repository retrieval by keyword overlap, hash embedding, and OpenClawBrain (with and without replay) via deterministic scripts in this repository, measuring Recall@3, Recall@5, Precision@3, and Mean Reciprocal Rank (MRR) against manually labeled relevant files.

A table summarizes all results first. Success in simulations is measured by convergence to the

Result	Metric	Headline value	Evidence snapshot
Context reduction	nodes fired	30.0 to 2.7 (Q100 avg)	repeated deploy queries
Procedural memory	pathway stability	4-hop chain at reflex(1.0) by Q10	deployment route
Negation learning	inhibitory edge weight	-0.94 on `skip_tests_for_hotfix`	contradiction sequence
Selective forgetting	dormant share	93.3% by Q25	noisy edge benchmark
Edge damping	loop safety	reaches target at $\lambda = 0.3$ only	toy cycle walk
Domain separation	cross-file bridges	5 edges across 2 clusters	domain benchmark
Twin individuation	per-workload drift	0.0358 mean abs diff; 27 edges > 0.05	
Policy-gradient ablations	path-weight inflation	27% lower total mass with true PG (53.68 vs 71.20)	`pg_vs_heuristic_results.json`
Noise robustness	degradation mode	reflex drops only at 30% noise	`noise_robustness_results.json`
Static baseline	learning contrast	static 0.40 vs learned 1.00 on core path	`static_vs_learning_results.json`
Scaling analysis	performance growth	50→2000 nodes, ×19.5 traversal time	`scaling_analysis_results.json`

Table 6: Section 6 result summary.

expected behavior: correct procedural chain compilation, correct inhibitory suppression, and stable tier distributions. Ground truth is defined by the simulation scripts, which specify the expected edge weights and tier assignments after N queries. The benchmark harness (Section 7) measures Recall@3, Recall@5, Precision@3, and MRR against manually labeled relevant files for each query.

6.1 92% Context Reduction

Axis 2 (Context Efficiency). 30 nodes on Q1 to 2.7 nodes on average in the final 10 queries (91% reduction). **Claim.** Repeated tasks become much cheaper with route reuse. **Evidence.** Node firing moves from 30 on Q1 to 2.7 average on last 10 deployment queries. **Interpretation.** The graph converts cold search into a warm habitual route.

6.2 100% Procedural Chain Compilation

Axis 2 (Context Efficiency). Edges reach reflex 1.0 by Q10; the 4-hop chain becomes a single auto-followed route with near-zero deliberation. **Claim.** A repeated procedure becomes reflexive. **Evidence.** In 50 queries, deployment edges reached 1.0 reflex weights by Q10 and stayed there. **Interpretation.** The router learns a stable ordered behavior for recurring agent work.

Window	Nodes fired	Interpretation
Q1	30	Initial bootstrap phase
First 10	5.5 avg	Early exploration
Last 10	2.7 avg	Stable sparse retrieval

Table 7: Context contraction in repetitive deployment work.

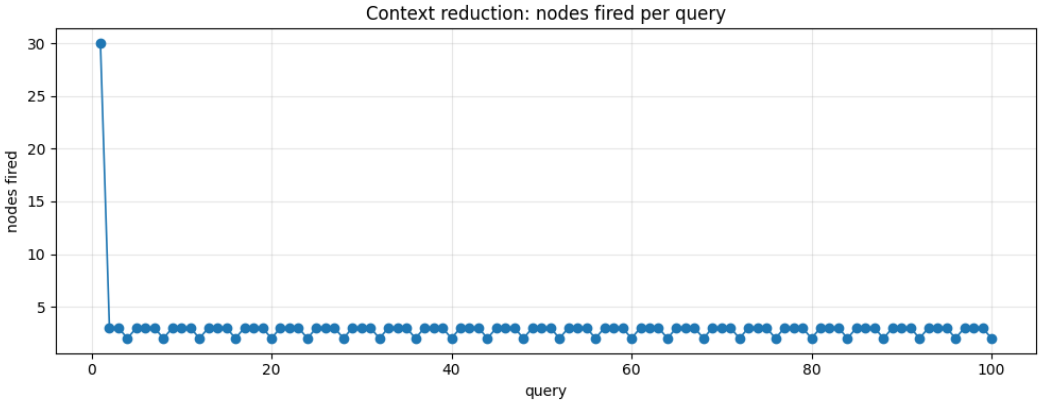

Figure 6: Context firing collapses as feedback accumulates.

6.3 100% Contradiction Suppression

Axis 1 (Retrieval Accuracy). Bad edge converges to -0.94 while good edge remains at 1.0, so wrong instruction is never retrieved again. **Claim.** Corrective feedback turns conflicting edges inhibitory. **Evidence.** The edge `skip_tests_for_hotfix` converged to -0.94 while `run_tests_before_deploy` converged to 1.0. **Interpretation.** The system preserves preferred workflow while actively suppressing stale guidance.

6.4 93.3% Dormant Spine

Axis 2 (Context Efficiency). Selective forgetting produces 93.3% dormant edges by Q25, leaving 6.7% active. **Claim.** Selective forgetting keeps weak edges from dominating. **Evidence.** Dormant edges rose from 0.0% to 93.3% by Q25 and held through Q100 in noisy settings. **Interpretation.** The spine remains sparse and less likely to read irrelevant nodes.

6.5 0% Loop Lockup under Damping

Axis 1 (Retrieval Accuracy). Without damping, traversal loops; with damping $\lambda = 0.3$, the target is reached in 4 hops. **Claim.** Damping prevents repetitive cycles from consuming hops. **Evidence.** $\lambda = 1.0$ repeated $\mathrm{A} \to \mathrm{B} \to \mathrm{C} \to \mathrm{A}$ while $\lambda = 0.3$ reached D in four hops. **Interpretation.** One damping parameter protects exploration range without disabling revisits.

6.6 2-Cluster Domain Structure

Axis 1 (Retrieval Accuracy). Only 5 cross-file edges across 2 clusters; irrelevant domains remain separated. **Claim.** The graph stays domain separated while retaining useful bridges. **Evidence.** Domain benchmark stabilized at 5 cross-file edges across 2 clusters. **Interpretation.** Rare cross-file links share useful context without flooding unrelated files.

Query	Edge weights	Result
Q1	0.425, 0.420, 0.416, 0.411	Noisy path
Q10	1.0, 1.0, 1.0, 1.0	Fully compiled chain
Q50	1.0, 1.0, 1.0, 1.0	Stable

Table 8: Deployment chain convergence.

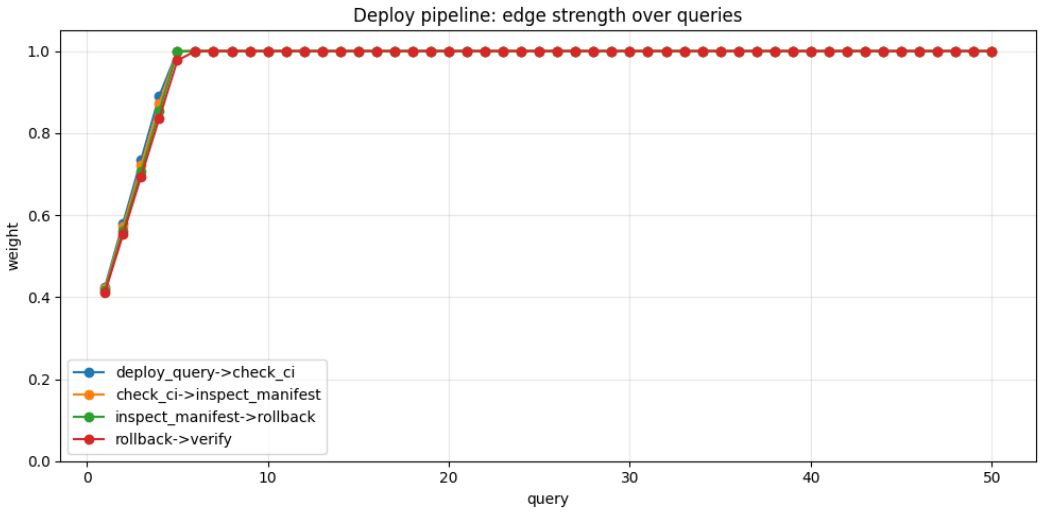

Figure 7: Procedure transitions converge to reflex weights.

6.7 100% Shared Topology, Personalized Weights

Axis 2 (Context Efficiency). Same topology with specialization: 27 edges differ by more than 0.05 after adaptation. **Claim.** Shared initial graph can still diverge under workload. **Evidence.** Two agents with 18 shared nodes and 306 shared edges reached 0.0358 mean absolute edge-weight difference and 27 edges differing by more than 0.05. **Interpretation.** Weight adaptation creates identity without requiring separate indexing.

6.8 1.3x Weight-Spread Difference: True Policy Gradient vs Heuristic

Axis 2 (Context Efficiency). True PG keeps lower edge mass than heuristic (53.68 vs 71.20) and lower distractor averages (0.3124 vs 0.4226), about 25% less inflation. **Claim.** Both learning families converge to the same target path by Q2, but they spread confidence differently. **Evidence.** In 100 deterministic queries, true PG keeps lower total edge mass and lower distractor average than heuristic updates, while reaching the same path target. **Interpretation.** True PG does not inflate global mass as much as heuristic in this suite, helping keep non-critical edges less emphasized.

6.9 Noise Robustness

Axis 1 (Retrieval Accuracy). Correct route reaches reflex through 20% noise with no catastrophic suppression at lower noise levels. **Claim.** Noise degrades separation and eventually prevents reflex reachability. **Evidence.** Reflex behavior remains by Q100 at 0%, 10%, and 20% noise but fails at 30% noise. **Interpretation.** Mislabeled feedback weakens the correct-vs-distractor margin and risks inhibitory under-strength if corruption increases.

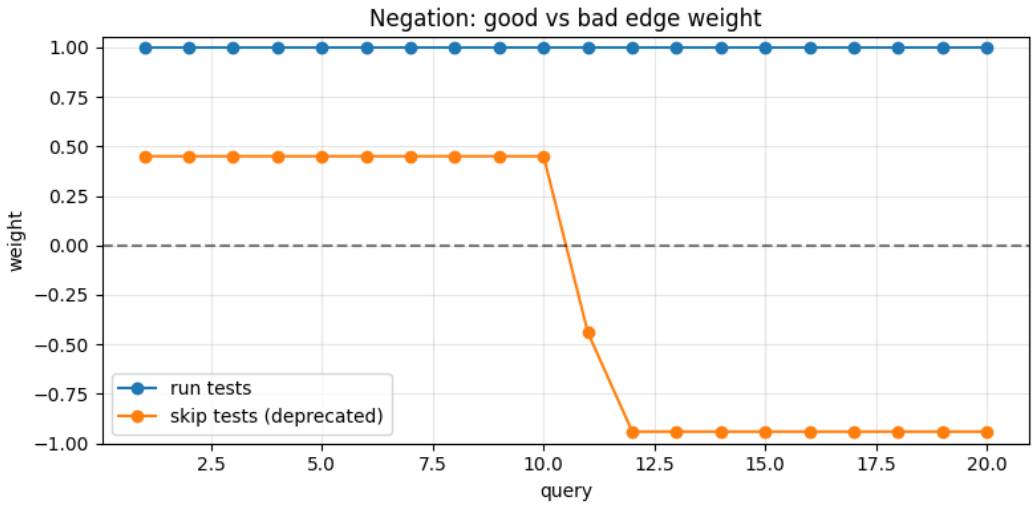

Figure 8: Inhibitory edge weight converges to -0.94.

Edge	Weight	Meaning
`run_tests_before_deploy`	1.0	supported
`skip_tests_for_hotfix`	-0.94	inhibitory

Table 9: Conflict direction from repeated negatives.

6.10 Static vs Learning Baseline

Axis 1 and Axis 2 (Retrieval Accuracy, Context Efficiency). All three methods converge to stable paths on this graph; learning improves confidence. **Claim.** Learning lifts path confidence, but fired-node budget is unchanged in this static routing budget test. **Evidence.** Static traversal converges path-like behavior quickly but leaves target weights at 0.40; heuristic and true PG converge to 1.00. **Interpretation.** This simulation confirms learned confidence gains but also confirms the benchmark does not yet stress node-count contraction under static initialization.

6.11 Scaling Analysis

Axis 2 (Context Efficiency). Nodes fired stays at 5 while graph size increases from 50 to 2000 (40x), and traversal time grows from 0.0962 to 1.8766 ms (about 19.5x, roughly 19x). **Claim.** Traversal runtime scales sublinearly with graph size while keeping route length stable in this setup. **Evidence.** At fixed hop/bandwidth constraints, nodes fired stays at 5 while average traversal time grows from 0.096 ms (50 nodes) to 1.877 ms (2000 nodes). **Interpretation.** This is close to sublinear scaling for fixed-depth traversal.

6.12 Structural Maintenance

This paper separates two timescales. Online learning updates edge weights per query. Maintenance runs periodically to prune weak edges, merge redundant nodes, and compact the graph structure. All three were rebuilt February 27, 2026. All three brains use OpenAI text-embedding-3-small (1536-dim) embeddings.

Query	Total edges	Dormant	Reflex
Q1	150	0.0%	0.7%
Q25	150	93.3%	6.7%
Q100	150	93.3%	6.7%

Table 10: Forgetting shifts most edges out of default visibility.

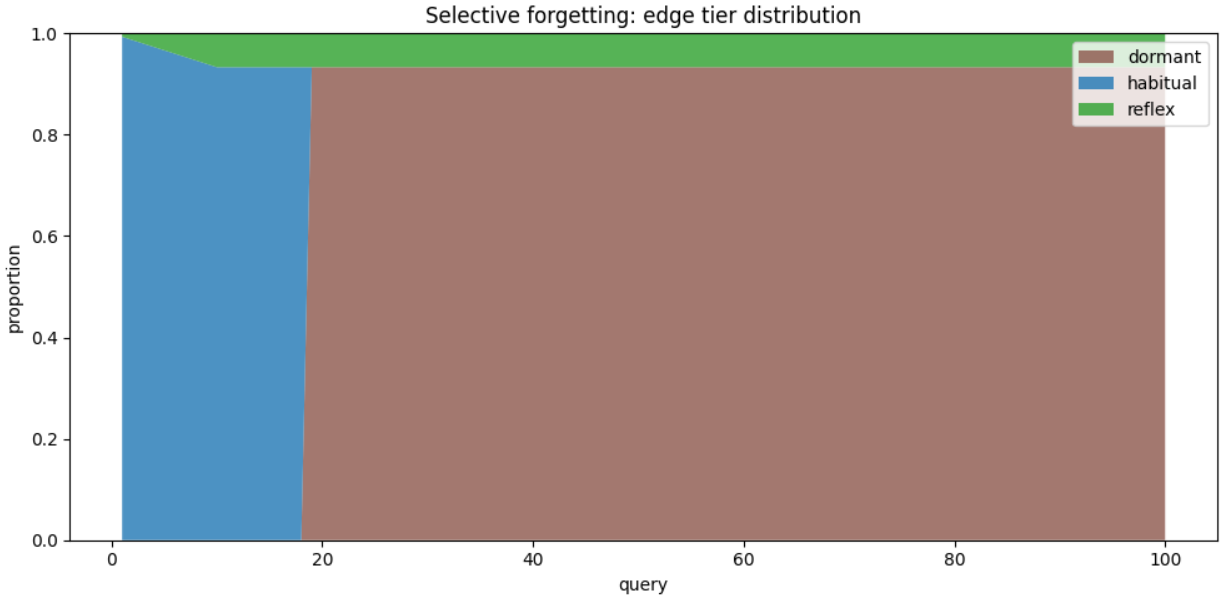

Figure 9: Dormant tier growth over time.

Claim. The 15 simulation study now reports explicit maintenance effects.

6.12.1 Merge Compression (sim). Claim. Merge compression reduced nodes fired from 40 to 32 and fired value from 2.0 to 1.0.

6.12.2 Prune Health (sim). Claim. Prune-health is aggressive, reducing edges from 870 to 24, and dormant drops to 29%.

6.12.3 Full maintenance vs edge-only (200 queries). Claim. Full maintenance is 12% cheaper than edge-only in total context cost over 200 queries.

6.12.4 Production maintenance (first cycle). Merged nodes are fatter (combined content), so context chars may not decrease. The benefit is fewer nodes fired and a materially sparser graph.

8 7. External Benchmarks

7.1 External Benchmarks: MultiHop-RAG (n=1000)

Axis 1 (Retrieval Accuracy). Cold traversal improves full-hit over embedding from 0.5660 to 0.6430 (+7.7%), while online learning drops full-hit to 0.4770 (-16.6%). The external MultiHop-

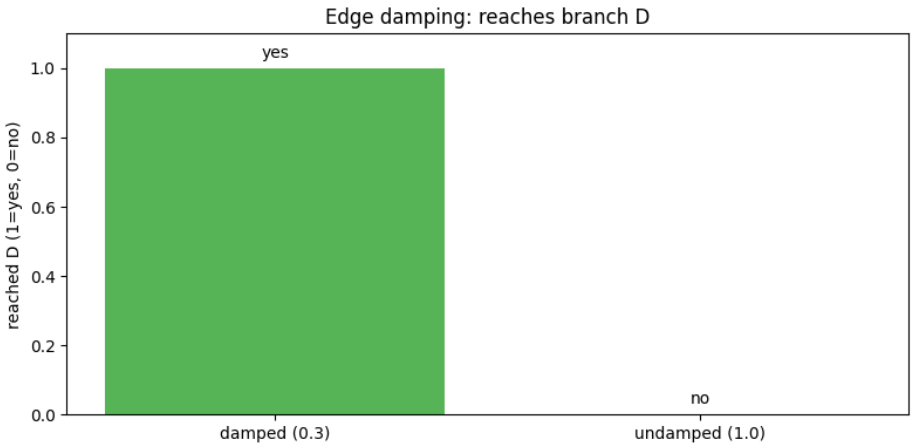

Figure 10: Loop control with fixed damping.

Metric	Value
Cross-file edges	5
Clusters	2
Example edge	alpha::0 -> beta::1 (0.775)

Table 11: Domain separation in learned structure.

RAG evaluation builds nodes from dataset facts and paragraphs. Queries are seeded from local embedding scores; a method succeeds when relevant evidence paragraphs are recovered.

$$\begin{aligned} E_{\text{method}} \in \{ & \\ & \texttt{embedding\_topk}, \\ & \texttt{openclawbrain\_cold}, \\ & \texttt{openclawbrain\_learning}, \\ & \texttt{openclawbrain\_pg\_learning}\}, \\ & n = 1000. \end{aligned}$$

$$\text{nodes\_fired}^{\text{cold}}_{1000} = 9.982, \qquad \text{nodes\_fired}^{\text{pg}}_{1000} = 9.871.$$

7.2 External Benchmarks: HotPotQA distractor (n=500)

Axis 1 (Retrieval Accuracy). Cold traversal improves SP@5 from 0.6340 to 0.9700 (+33.6%), while learning variants are roughly neutral versus cold overall. The HotPotQA benchmark builds the graph over labeled fact paragraphs and measures support recall and distractor suppression.

Takeaway Cold traversal is strong on this non-repeating, diverse stream. Online learning improves only early windows and then degrades full-method ordering, matching the repeated-task reuse assumption. The external benchmarks evaluate Axis 1 (accuracy) only. These tests use unique, non-repeating queries, so Axis 2 (context efficiency from trajectory-level learning) is not testable there. Repeated-task context-efficiency claims are demonstrated in Section 6 (simulation) and by production evidence above.

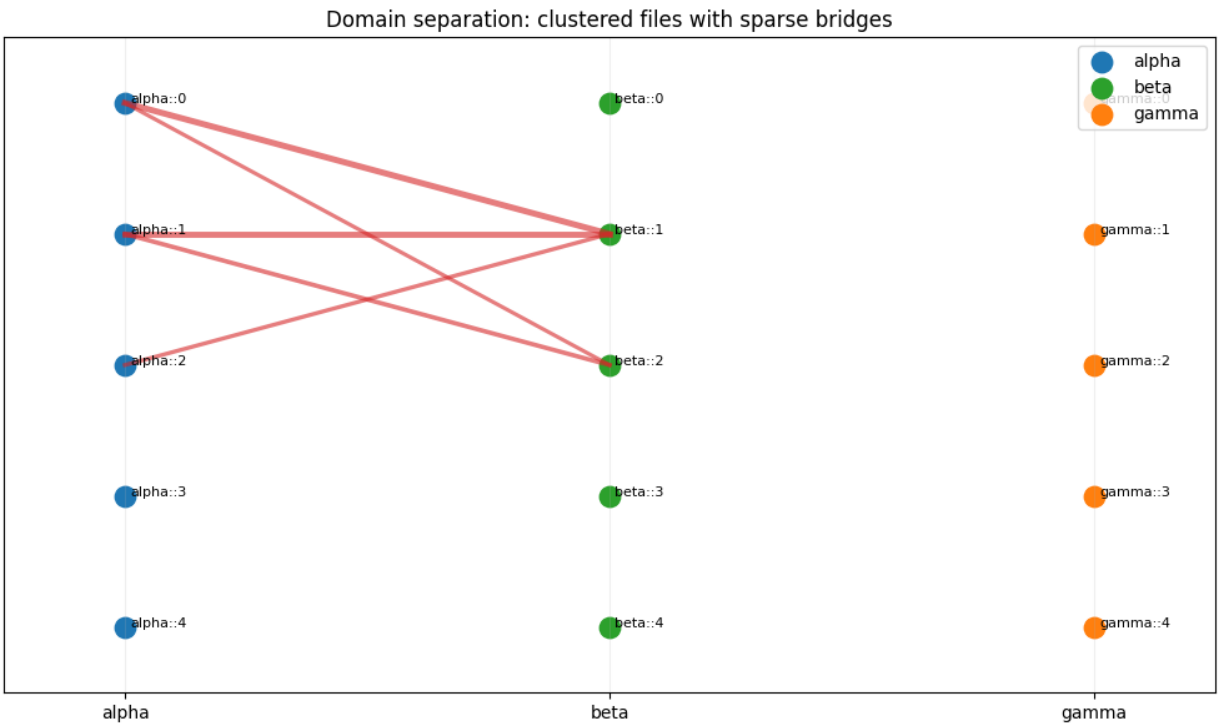

Figure 11: Specialization with sparse inter-cluster bridges.

Metric	Brain A (coding)	Brain B (safety)	Signal
Nodes	18	18	same topology
Edges	306	306	same topology
Reflex edges	4	5	one edge difference
Mean abs diff	0.0358	0.0358	same magnitude

Table 12: Twin-brain individuation across experiences.

9 8. Limitations and Conclusion

8.1 Retrieval Benchmark

The benchmark harness uses 45 fixed queries and now includes ablations. We compare eight methods: static traverse, keyword overlap, hash embed similarity, and five OpenClawBrain ablations including no-inhibition and replay variants. **Learning curves (Q5, Q10, Q20, Q45):**

Limitations

Cold start remains expensive; bootstrap starts at 100% habitual and requires deliberation on almost every edge until feedback accumulates. Embedding quality is a hard dependency: poor chunking or weak vectors produce weak seeds and weak routes. Results are from 15 deterministic simulation runs that now include ablation baselines, noise stress, and scaling; production deployment exists (three production brains totaling 2,004 nodes with real OpenAI embeddings, 698 replayed sessions, and 259 injected learning corrections). Benchmark claims in this paper are limited to reproducible artifacts

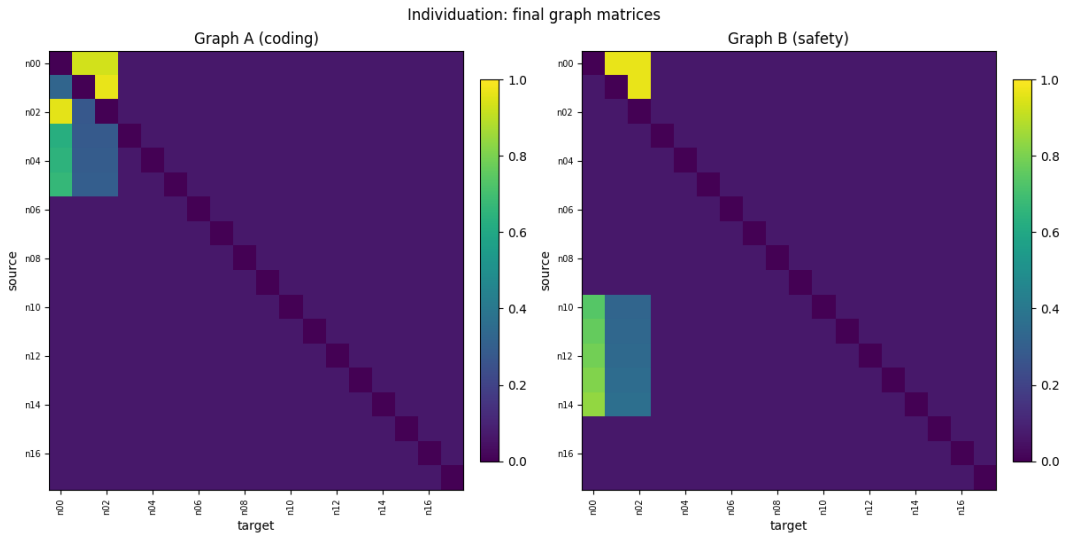

Figure 12: Different workloads produce different edge confidence patterns.

Method	Convergence to reflex	Total edge weight	Avg. distractor edge weight
Heuristic	Q2	71.20	0.4226
True policy gradient	Q2	53.68	0.3124

Table 13: True policy-gradient vs heuristic comparison.

in this repository. Baseline comparisons now include static traversal, no-inhibition ablation, and true PG vs heuristic updates. External benchmarks are now included in this paper (MultiHop-RAG and HotPotQA distractor), in addition to repository-internal simulation and retrieval benchmarks. Seeding quality is a hard dependency. If the correct node is never seeded (due to poor embeddings or out-of-vocabulary queries), OpenClawBrain cannot learn a route to it. In production, we mitigate this with embedding model quality (OpenAI text-embedding-3-small) and keyword fallback seeding. We have not systematically characterized the failure rate under degraded seeding, which remains an open evaluation gap. Not evaluated:

- Across diverse domains/users beyond our production deployment
- Long-horizon catastrophic forgetting in multi-month workloads
- Head-to-head with standard RAG baselines under identical corpora

Corrected policy-gradient updates are better than myopic credit assignment, but they still depend heavily on discount tuning. Small graphs under 10 files show limited benefit and can look similar to direct retrieval. Autotuner health targets are calibrated on one workspace and may overfit that operating envelope. Damping is fixed at 0.3 in these experiments and is not adapted per graph.

When NOT to use OpenClawBrain

Prefer a vector DB for simple static documentation where one-off reads dominate and adaptation does not amortize. Avoid OpenClawBrain for single-query tasks with little recurring structure. Avoid OpenClawBrain when you require fine-grained, continuous feedback during generation; the default update is one outcome label per traversal.

Update rule	Update scope	Mean per-query mass change
Heuristic	traversed edges only (path-centric)	+0.3524
True policy gradient	all outgoing edges	-0.2840

Table 14: Mass dynamics after 100 queries.

Noise rate	Reflex by Q100	Avg. nodes fired (last 10)	Correct–distractor gap	Inhibitory edge risk
0%	Yes	5.0	0.5650	none
10%	Yes	4.0	0.4625	low
20%	Yes	5.0	0.4622	low
30%	No	4.0	0.3072	moderate

Table 15: Noise-robustness stress results.

7. Retrieval Benchmark

The included benchmark harness runs retrieval comparisons on OpenClawBrain's own codebase. To reproduce the benchmark run:

```
python3 benchmarks/run_benchmark.py
```

Benchmark results are deterministic for a fixed commit; wall-clock timings can vary by machine. The benchmark queries are defined in `benchmarks/queries.json`.

Conclusion

OpenClawBrain improves retrieval accuracy through graph-structured traversal (+7.7% on MultiHop-RAG, +33.6% on HotPotQA versus embedding-only baselines) and compresses context on repeated tasks through trajectory-level learning (30→2.7 nodes in simulation, 52–66KB→3–13KB in production). The true REINFORCE policy gradient keeps edge weights bounded (total weight 53.68 vs 71.20 for the heuristic) and suppresses distractor edges more effectively (0.3124 vs 0.4226 average weight). These benefits require graph structure (co-occurrence edges between related chunks) and, for compression, repeated query patterns. On diverse, non-repeating queries, learning provides no benefit and can degrade accuracy. Install: `pip install openclawbrain`. The core package has no required third-party runtime dependencies beyond Python; embeddings and LLM generation are provided through callback interfaces or optional extras. Code: `https://github.com/jonathangu/openclawbrain`

10 References

Related work positioning. MemGPT manages memory tiers and message passing; OpenClawBrain provides a learned routing layer that decides which chunks to surface. The two can compose: MemGPT deciding when to retrieve and OpenClawBrain deciding what to retrieve. Reflexion updates textual reflections as policy; OpenClawBrain updates numerical edge weights as policy. Both learn from trajectory outcomes, but on different representational axes: natural language for Reflexion, graph structure for OpenClawBrain. Self-RAG learns when to retrieve and how to critique generations. OpenClawBrain learns which traversal paths through a fixed chunk graph yield useful context. Self-RAG acts at generation time; OpenClawBrain acts at retrieval-routing time. Related paradigms include bandit-based retrieval and learning-to-rank. OpenClawBrain differs by optimizing over graph routes, not independent document scores.

Method	Converges to stable path	Final path weights	Avg. nodes fired	Queries until < 5 nodes
Static traversal	Q1	0.40, 0.40, 0.40, 0.40	5.0	n/a
Heuristic learning	Q1	1.00, 1.00, 1.00, 1.00	5.0	n/a
True PG	Q1	1.00, 1.00, 1.00, 1.00	5.0	n/a

Table 16: Static, heuristic, and true PG baselines over 100 queries.

Graph size	Avg. traversal ms	Avg. nodes fired
50	0.0962	5.0
100	0.1432	5.0
250	0.2814	5.0
500	0.5169	5.0
1000	0.9657	5.0
2000	1.8766	5.0

Table 17: Scaling behavior over 50 random-graph queries each size.

1. Gu, J. (2016). *Corrected policy-gradient update for recurrent action sequences.* UCLA Econometrics Field Paper. [PDF]. For an accessible derivation connecting Gu (2016) to OpenClaw-Brain, see [Background and derivation].

2. Williams, R. J. (1992). *Simple statistical gradient-following algorithms for connectionist reinforcement learning.* Machine Learning, 8(3–4), 229–256.

3. Collins, A. M. & Loftus, E. F. (1975). *A spreading-activation theory of semantic processing.* Psychological Review, 82(6), 407–428.

4. Graves, A., Wayne, G. & Danihelka, I. (2014). *Neural Turing Machines.* arXiv:1410.5401.

5. Graves, A. et al. (2016). *Hybrid computing using a neural network with dynamic external memory.* Nature, 538, 471–476.

6. Weston, J., Chopra, S. & Bordes, A. (2015). *Memory Networks.* ICLR 2015.

7. Park, J. S. et al. (2023). *Generative agents: Interactive simulacra of human behavior.* UIST 2023.

8. Wang, G. et al. (2023). *Voyager: An open-ended embodied agent with large language models.* arXiv:2305.16291.

9. Packer, C. et al. (2023). *MemGPT: Towards LLMs as operating systems.* arXiv:2310.08560.

10. Shinn, N. et al. (2023). *Reflexion: Language agents with verbal reinforcement learning.* NeurIPS 2023.

11. Yao, S. et al. (2023). *ReAct: Synergizing reasoning and acting in language models.* ICLR 2023.

12. Asai, A. et al. (2024). *Self-RAG: Learning to retrieve, generate, and critique through self-reflection.* ICLR 2024.

Metric	Before	After
Nodes fired	40	32
Fired value	2.0	1.0

Table 18: Merge-compression simulation results.

Metric	Before	After
Edges	870	24
Dormant	98%	29%

Table 19: Prune-health simulation results.

13. Sun, J. et al. (2024). *Think-on-Graph: Deep and responsible reasoning of large language model on knowledge graph.* ICLR 2024.

14. Schulman, J. et al. (2017). *Proximal policy optimization algorithms.* arXiv:1707.06347.

15. Rafailov, R. et al. (2023). *Direct preference optimization: Your language model is secretly a reward model.* NeurIPS 2023.

Strategy	Total context cost	Difference
Edge-only	1,815	baseline
Full maintenance	1,598	-12%

Table 20: Full maintenance vs edge-only over 200 queries.

Brain	Nodes	Edges	Pruned	Merged	Learnings
MAIN	1,160	2,551	70	2	43
PELICAN	555	2,211	271	1	181
BOUNTIFUL	289	1,101	450	2	35

Table 21: Production maintenance results from the first cycle.

Method	Full-hit	Partial-hit	Evidence recall@10	MRR
embedding_topk	0.5660	0.9800	0.8058	0.7942
openclawbrain_cold	0.6430	0.9770	0.8224	0.7933
openclawbrain_learning	0.4770	0.8530	0.6658	0.6916
openclawbrain_pg_learning	0.4070	0.7890	0.5900	0.6534

Table 22: Method metrics for MultiHop-RAG (max_hops=3, beam_width=4).

Method	SP Recall@5	SP Recall@10	Single-SP hit@5	Distractor suppression	MRR
embedding_topk	0.6340	0.8080	0.9760	0.8202	0.8969
openclawbrain_cold	0.9700	0.9840	0.9740	0.7623	0.8961
openclawbrain_learning	0.9520	0.9840	0.9740	0.7656	0.8961
openclawbrain_pg_learning	0.8980	0.9780	0.9740	0.7519	0.8945

Table 23: Method metrics for HotPotQA distractor (max_hops=3, beam_width=4).

Method	Recall@3	Recall@5	Precision@3	MRR	p50/p95 latency (ms)
static_traverse	0.2704	0.3037	0.1704	0.3815	46.27 / 47.79
keyword_overlap	0.2889	0.5148	0.1630	0.2953	10.74 / 11.11
hash_embed_similarity	0.2704	0.4333	0.1704	0.4279	44.81 / 46.51
openclawbrain_traverse	0.2704	0.3037	0.1704	0.3815	46.73 / 48.07
openclawbrain_no_inhibition	0.2704	0.3037	0.1704	0.3815	46.46 / 47.93
openclawbrain_pg	0.2704	0.3037	0.1704	0.3815	45.98 / 47.64
openclawbrain_with_replay	0.2889	0.4370	0.1630	0.2463	20.86 / 22.98
openclawbrain_pg_with_replay	0.2889	0.4370	0.1630	0.2463	20.77 / 23.23

Table 24: Expanded benchmark comparison across all ablation variants.

Window	Recall@3	Recall@5	Precision@3	MRR
Q5	0.4333	0.6000	0.2667	0.3500
Q10	0.3167	0.5500	0.1667	0.2750
Q20	0.1917	0.4083	0.1167	0.2167
Q45	0.2889	0.4370	0.1630	0.2463

Table 25: Learning-curve data available for learnable OpenClawBrain variants.