CrabPath: The Graph is the Prompt Learning Weighted Memory Traversals with LLM-Guided Activation and Corrected Policy Gradients

Jonathan Gu jonathangu@gmail.com Independent Researcher San Francisco

February 2026

Abstract

Retrieval-augmented generation treats memory as a filing cabinet: fetch by similarity, hope for the best. It cannot learn *which sequence of retrievals* leads to good outcomes. CrabPath replaces similarity search with a learned routing policy over a document graph. Nodes are typed information frames; edges are weighted pointers whose values cache past LLM routing decisions. At query time, an LLM traverses the graph for two to three hops, following high-weight edges reflexively and deliberating over uncertain ones. After each episode, a corrected policy-gradient estimator — derived from Gu (2016) — propagates terminal feedback across the *entire* traversal path, not just the final hop. Over repeated interactions, expensive LLM reasoning compiles into cheap graph topology: reflexive routes for familiar tasks, inhibitory edges for known failure modes, and dormant paths for rare contingencies. The result is a memory system that learns procedures, not just facts.

1 Introduction — Memory is a Policy, Not a Database

The $13K$ incident changed how I think about agent memory. I ran three autonomous agents on overlapping tasks for twenty days. The spending was not random: the majority came from repeatedly loading irrelevant context. The agents retrieved the same documents and the same tooling traces regardless of task outcome. They could recall facts, but they could not learn which retrieval sequences led to success. They were filing clerks, not learners.

Memory is not storage. Memory is a policy for attention.

A filing cabinet returns what matches a label. A brain routes what should happen next. Retrieval-augmented generation is good at answering what is true? and weak at answering what sequence of retrievals leads to a correct action? To answer that second question, agents need procedural knowledge — the learned ability to route through `code review` → `ci check` → `fix manifest` → `verify run`, and to adapt that route when conditions change. A system that does not learn this is a lookup service.

I kept a crab metaphor throughout this project because systems under selection pressure tend to converge on the same body plan. In biology, this is carcinisation — independent lineages evolving toward crab-like forms. In memory systems, the convergent shape is *index + similarity retrieval*, with no learning of routing behavior. CrabPath aims to evolve past that local optimum.

This paper makes three contributions:

1. **Memory as a learned policy.** CrabPath converts document-graph traversal into an episodic MDP where the LLM is the activation function and pointer weights parameterize the routing policy.

2. **Corrected credit assignment.** Drawing on Gu (2016) [1], we derive a trajectory-summed policy-gradient update that propagates terminal feedback to every routing decision in the traversal, not just the final hop.

3. **Reasoning-to-topology compilation.** High-weight pointers become reflexive (no LLM call needed); low-weight pointers stay dormant. Over time, expensive deliberation compiles into cheap graph structure, reducing both latency and cost.

2 Architecture — The Graph is the Prompt

CrabPath has three components: nodes, edges, and an LLM activation function.

- **Nodes are documents.** Each stores content, a short summary, and a semantic type (fact, procedure, action, or tool call).
- **Edges are weighted pointers.** Weights are signed and mutable, ranging from -1 (strong inhibition) to $+1$ (strong excitation).
- **The LLM is the activation function.** It reads the current node's content, inspects candidate outgoing pointers with their summaries, and decides which edges to traverse.

2.1 Node and Edge Schema

Listing 1: Node and edge schema in Python.

```python
from dataclasses import dataclass, field
from typing import Literal, List

@dataclass
class Node:
    id: str
    content: str
    summary: str
    type: Literal[fact, procedure, action, tool_call]
    pointers: List[Edge] = field(default_factory=list)

@dataclass
class Edge:
    target: str
    weight: float       # in [-1, 1]
    kind: Literal[support, inhibit, follows, tool]
    summary: str
```

Listing 2: Example serialized node.

```
{
  id: giraffe-codeword,
  content: Giraffe is Jon's codeword for testing CrabPath.,
  summary: Codeword note. Routes to codeword policy anchor.,
  type: fact,
```

```
  pointers: [
    {target: codewords, weight: 0.71, kind: follows, summary: List of codeword mea
    {target: elephant-codeword, weight: 0.12, kind: inhibit, summary: Negative test
  ]
}
```

The graph is persistent JSON on disk. Topology itself is the memory abstraction: the structure of connections and their weights encode what the system has learned, not just what it has stored.

2.2 Forward Pass as Traversal

1. The user submits a query.
2. Semantic seed search selects entry nodes by embedding similarity.
3. The LLM reads the current node and all candidate outgoing pointers with summaries.
4. The LLM chooses which edges to traverse, constrained by depth budget (2–3 hops).
5. Visited node content is assembled as context for the final response.

The crucial difference from v1 is that there is no LIF-like numerical wave. The router is semantic: it reads content, understands negation, and reasons about which paths serve the current query.

2.3 Beyond Retrieval: What Learned Routing Enables

Standard RAG maps a query to its nearest embedding neighbors. CrabPath maps a query plus accumulated traversal state to a *sequence of routing decisions* through documents. This distinction enables three capabilities that similarity search alone cannot provide:

- **Procedural sequencing.** The graph can encode multi-step workflows as chains of weighted pointers.
- **Native negation.** "Do not use deprecated endpoint X" is represented as an inhibitory edge, not a keyword match against X.
- **Adaptive cost.** Familiar queries follow reflexive high-weight paths with no LLM call, while novel queries trigger deliberation.

3 The Three Tiers — How Reasoning Becomes Topology

Edge weights partition naturally into three operational tiers that control the trade-off between reasoning quality and inference cost.

The following code sketch shows how the three tiers interact during a single traversal step:

Listing 3: Three-tier routing in one traversal step.

```
def choose_next(current_node, graph, router, query, visited):
    One step of three-tier routing.
    edges = graph.outgoing(current_node)
    next_nodes = []
```

Tier	Weight Range	Routing Behaviour	Cost
Reflex	> 0.8	Auto-follow, no LLM call	Very low
Habitual	$0.3 – 0.8$	Presented as candidates to LLM	Moderate
Dormant	< 0.3	Skipped unless explicit override	Near zero

Table 1: Three pointer routing tiers.

```
# Tier 1: Reflex --- auto-follow high-weight edges (no LLM call)
for edge in edges:
    if edge.weight > 0.8 and edge.target not in visited:
        next_nodes.append(edge.target)

# Tier 2: Habitual --- present mid-weight edges to LLM for decision
candidates = [e for e in edges
              if 0.3 <= e.weight <= 0.8 and e.target not in visited]
if candidates:
    prompt = format_routing_prompt(query, current_node, candidates)
    selected = router.decide(prompt)
    next_nodes.extend(selected)

# Tier 3: Dormant --- low-weight edges are skipped entirely
# (accessible only via fresh semantic search at entry)

return next_nodes
```

One empirical observation from the v2 experiments is that a three-hop LLM traversal costs roughly 7,500 input tokens per query (about ten times more than static RAG at small scale). As reflex edges accumulate, average cost per query decreases because fewer hops require LLM calls.

4 Learning — Corrected Policy Gradients for Pointer Weights

4.1 MDP Mapping

Each user request is modelled as one episode of a Markov decision process.

- **State** s_t: current node, plus the query and accumulated context.
- **Action** a_t: which outgoing pointer to follow, or `STOP`.
- **Reward** z: terminal scalar feedback ($+1$ for success, -1 for correction).
- **Policy** $\pi_W(a|s)$: softmax over outgoing pointer weights, optionally augmented by LLM relevance scores.

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

4.2 The Myopic Update (Williams, 1992)

The standard one-step REINFORCE update is:

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

This update focuses on the immediate action.

4.3 The Gu (2016) Corrected Update

The corrected estimator sums score-function gradients across every step in the trajectory:

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

When variance is a concern, a baseline b and discount factor γ can be introduced:

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

4.4 Formal Mapping: Dissertation to CrabPath

The mapping from finite-horizon game RL to routing remains structurally identical.

Gu (2016) — Game RL	CrabPath — Document Routing
State s_t: board position	State s_t: current document node + query + context
Action a_t: move	Action a_t: pointer selection or STOP
Policy parameters ρ	Edge weights $W = \{w_{ij}\}$
Terminal reward $z \in \{+1, -1\}$	Terminal feedback $z \in \{+1, -1\}$
Value function $v_\rho(s)$	Expected utility of starting traversal at node s

Table 2: Mapping of Gu (2016) notation to CrabPath.

4.5 The Giraffe Example (Worked Numerics)

Query: "What is Jon's codeword for testing?"

Nodes: G = giraffe-codeword, C = codewords, E = elephant-codeword, plus STOP.

Weights: $w_{G,C} = 1.0$, $w_{G,E} = 0.0$, $w_{C,E} = 0.0$, $w_{C,\text{STOP}} = 0.0$. With $\tau = 1$, softmax gives $\pi(C \mid G) = 0.731$ and $\pi(E \mid G) = 0.269$.

Suppose the sampled trajectory is $G \to C \to E$, with $z = +1$.

- At G (chose C): $\partial \log \pi(C \mid G)/\partial w_{G,C} = 1 - 0.731 = 0.269$, and $\partial/\partial w_{G,E} = -0.269$.
- At C (chose E): $\partial \log \pi(E \mid C)/\partial w_{C,E} = 0.5$, and $\partial/\partial w_{C,\text{STOP}} = -0.5$.

With learning rate $\eta = 0.1$:

- **Myopic update**: $\Delta w_{C,E} = +0.05$, $\Delta w_{C,\text{STOP}} = -0.05$.
- **Corrected update**: $\Delta w_{G,C} = +0.0269$, $\Delta w_{G,E} = -0.0269$, $\Delta w_{C,E} = +0.05$, $\Delta w_{C,\text{STOP}} = -0.05$.

Negative feedback $z = -1$ flips all signs.

4.6 Algorithm: Traversal and Weight Update

Algorithm 1: Gu-corrected weight update

Input: trajectory $\{(s_t, a_t, \mathcal{C}_t)\}_{t=0}^{T}$, weights W, feedback z, learning rate η, temperature τ, baseline b, discount γ
Output: updated weights W
$G \leftarrow \emptyset$
foreach (s, a, C) *in trajectory* **do**
  $\ell \leftarrow [W[(s, c)] : c \in C]$
  $p \leftarrow \text{softmax}(\ell/\tau)$
  foreach c, p_c *in pairs*(C, p) **do**
    $G[(s, c)] \leftarrow G[(s, c)] + (\mathbf{1}[c = a] - p_c)$
  end
end
foreach $(s, c), g$ *in* G **do**
  $W[(s, c)] \leftarrow \text{clip}(W[(s, c)] + \eta(z - b)g/\tau, -1, 1)$
end
return W

5 Neurogenesis and Death

A static graph cannot represent knowledge the system has never encountered. CrabPath creates nodes when topology is insufficient and prunes nodes that consistently fail to contribute.

5.1 When Does the LLM Create a Node?

Node creation is triggered by three conditions:

1. **The gap trigger.** The LLM reaches a leaf node without finding needed information. The system creates a new node for the missing item and links it to context.
2. **The correction trigger.** The user corrects a response. A correction node is created and attached with inhibitory links to previous wrong routes.
3. **The novel-concept trigger.** A query references a concept with no matching entry point. A new root node is created and linked to co-occurring context.

5.2 The Cosine-Band Gate

Given query embedding e_q and nearest existing node embedding e_n:

- If $\cos(e_q, e_n) > 0.85$, no new node.
- If $\cos(e_q, e_n) < 0.30$, no new node unless explicitly overridden.
- If $0.30 \leq \cos(e_q, e_n) \leq 0.85$, create a probationary node with provisional edges.

Probationary nodes require three positive reinforcements to become permanent.

5.3 Node Lifecycle

1. **Create:** detect novelty, generate node, attach neutral edges.
2. **Strengthen:** repeated successful traversal reinforces links; after three positive episodes, node exits probation.
3. **Reassign:** corrections can create alternate nodes and inhibitory links.
4. **Decay:** weights decay toward zero; orphaned nodes are removed.

The Giraffe cycle in practice: first mention creates a probationary node; positive use makes it permanent; negative feedback downgrades it or replaces it through reassign.

6 Negation and Inhibition

Negation is a critical test for any memory system. CrabPath uses the codeword is elephant, not giraffe scenario as a core stress test.

> User asks: "What's the codeword for this testing flow? The system must answer **elephant**, not **giraffe**.

Similarity-based retrieval fails here because semantically close nodes are both returned and disambiguation is left to generation alone.

7 Evaluation — Preliminary Results

We ran five experiments comparing four retrieval approaches across synthetic benchmark graphs.

7.1 Context Efficiency (The Main Result)

The headline finding: **CrabPath loads 20–90x fewer tokens per turn than static context loading.**

7.2 Two-Tier Cost Model

Cost is split between a cheap router and a costly context reader.

Experiment	Static (tokens/turn)	RAG (tokens/turn)	CrabPath (tokens/turn)	Reduction
Context Bloat (50 nodes)	6,066	744	**297**	95%
Gate Bloat (130 gates)	8,163	407	**89**	99%
Stale Context	895	507	**88**	90%
Negation	546	546	**87**	84%
Procedure	548	467	**205**	63%

Table 3: Context efficiency across benchmark tasks.

Method	Router Cost	Context Cost
Static (Opus reads everything)	–	6,066 tok × \$15/M = \$0.091
CrabPath (GPT-mini routes, Opus reads fired)	200 tok × \$0.30/M = \$0.00006	297 tok × \$15/M = \$0.004

Table 4: Two-tier cost comparison (router + context).

Episode	Query	LLM Chose	Reward	w(giraffe)	w(elephant)
1	What is the codeword?	giraffe	+1	0.740	0.260
2	Remember the codeword is giraffe	giraffe	+1	0.778	0.222
3	*The codeword is now elephant, not giraffe*	**elephant**	+1	0.715	0.285
4	What is the codeword?	**elephant**	+1	0.654	0.346
5	What is the codeword?	giraffe	-1	0.612	0.388
6–8	Various codeword queries	**elephant**	+1	0.453	**0.547**

Table 5: Giraffe test trajectory and weight trajectory.

7.3 The Giraffe Test (LLM-Guided Routing)

The system crossed from giraffe to elephant by episode 8.

7.4 Benchmark Tasks

- **Context Bloat:** 50+ nodes, 20 queries each, 3–5 nodes per query.
- **Gate Bloat:** 130 behavioral gates, 20 queries testing per-turn gate precision.
- **Stale Context:** 30 episodes with corrections mid-sequence.
- **Negation:** Conflicting nodes with one correction then evaluation of inhibition.
- **Procedure:** 4-step sequential procedure spread across nodes.
- **Deploy Pipeline:** 8-node graph with safe path (check tests → CI → staging → prod) and dangerous shortcut (skip tests → prod). Tests procedural learning and shortcut suppression.

7.5 Deploy Pipeline (Procedural Learning)

We built a graph with two paths to production: a safe path and a dangerous shortcut. Both start at equal weight (0.5). Over 15 episodes, the agent deploys, receives feedback, and updates pointer weights.

Episode	Event	w(check_tests)	w(skip_tests)
1–2	Safe deploy, $z = +1$	0.572	0.473
3	Dangerous shortcut, $z = -1$	0.587	0.436
5	Dangerous shortcut, $z = -1$	0.637	0.386
10	Safe deploy, $z = +1$	**0.802**	0.327
13	Safe deploy, $z = +1$	0.896	**0.294**
15	Safe deploy, $z = +1$	**0.957**	**0.273**

Table 6: Deploy pipeline weight trajectory. Safe path reaches reflex (> 0.8) by episode 10. Dangerous shortcut becomes dormant (< 0.3) by episode 13. Two negative rewards were sufficient.

7.6 Where RAG Fails

On the deploy pipeline, RAG loads 525 tokens per turn — the same as static. All deployment nodes match "deploy" semantically, so top-k retrieval is indiscriminate. CrabPath loads 199 tokens, following only the learned path.

7.7 Baselines

1. **Static:** Load all node content every turn.
2. **RAG top-k:** Retrieve 8 most similar nodes.
3. **CrabPath, myopic:** LLM-guided traversal with Williams-style last-hop-only updates.
4. **CrabPath, corrected:** Full trajectory-summed corrected policy gradient.

Experiment	Static	RAG	CrabPath	Note
Context Bloat	6,066	744	**297**	RAG helps
Gate Bloat	8,163	407	**89**	RAG helps
Deploy Pipeline	525	**525**	**199**	RAG = Static
Negation	546	**546**	**87**	RAG = Static

Table 7: When documents are semantically similar, RAG degrades to static loading. CrabPath routes by learned outcomes, not similarity.

All code, experiment graphs, scenarios, and results are available at `https://github.com/jonathangu/crabpath`.

8 Related Work

The system touches several streams: spreading activation, memory-augmented neural models, agentic memory, graph retrieval, and RL for language systems.

Collins and Loftus (1975) introduced spreading activation for semantic processing, and graph retrieval baselines remain important conceptual references. For memory-augmented architectures, NTM and DNC provide differentiable external memory, while Memory Networks introduce attention over memory banks and kNN-LM adds retrieval from a datastore. See[3, 4, 5, 6, 11].

Generative agents and lifelong agents provide related narratives around correction and procedural adaptation, while MemGPT and Reflexion emphasize long-horizon tooling and self-correction with separate memory layers. ReAct and Self-RAG are central to adaptive retrieval with reasoning. For a formal RL treatment, Williams, PPO, GRPO, and DPO remain canonical for policy and preference optimisation. [7, 8, 9, 10, 15, 16, 12, 13, 14]

GraphRAG and Think-on-Graph show graph-based retrieval structure, but not the same trajectory-level weight learning mechanism.[17, 18, 19]

9 Discussion

CrabPath is a graph-level model where nodes are inspectable artifacts and edges are inspectable policy parameters. If a tool-call node fails repeatedly, routing weakens it; if it succeeds repeatedly, the edge becomes reflexive.

9.1 Limitations

- **LLM latency:** three-hop LLM-guided traversal adds 1–3 seconds.
- **Cost at scale:** traversal tokens are initially higher than static RAG.
- **Hallucination risk:** route selection can still fail via misread metadata.
- **Cold start:** fresh graphs require deliberation for all routes.
- **Evaluation status:** this draft still has limited controlled evaluation.
- **Single-user feedback:** multi-user credit isolation is not yet implemented.

10 Conclusion

Memory is not storage; memory is a policy for attention. CrabPath reframes retrieval as learned graph routing, where pointer weights cache the LLM's own past reasoning. The core novelty is a direct application of trajectory-corrected policy gradients to pointer weights.

> The first retrieval system where the index structure is a cached projection of the LLM's own reasoning history.

Open issues remain — cost, cold start, and evaluation breadth — but the architectural move is clear: replace static similarity search with learned routing.

Code and design notes: `https://github.com/jonathangu/crabpath`.

Figure 1: LLM-guided memory traversal over a weighted document graph.

References

References

[1] J. Gu. Corrected policy-gradient update for recurrent action sequences. *UCLA Econometrics Field Paper*, 2016.

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

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

[4] A. Graves, G. Wayne, and I. Danihelka. Neural Turing Machines. *arXiv:1410.5401*, 2014.

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

[6] J. Weston, S. Chopra, and A. Bordes. Memory Networks. In *ICLR*, 2015.

[7] J. S. Park et al. Generative Agents: Interactive simulacra of human behavior. In *UIST*, 2023.

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

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

[10] N. Shinn et al. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

[11] U. Khandelwal et al. Generalization through memorization: Nearest neighbor language models. In *ICLR*, 2020.

[12] J. Schulman et al. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

[13] Z. Shao et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv:2402.03300*, 2024.

[14] R. Rafailov et al. Direct preference optimization: Your language model is secretly a reward model. In *NeurIPS*, 2023.

[15] S. Yao et al. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

[16] A. Asai et al. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *ICLR*, 2024.

[17] Microsoft Research. From local to global: A graph RAG approach to query-focused summarization. Microsoft GraphRAG, 2024. `https://arxiv.org/abs/2404.16130`.

[18] J. Sun et al. Think-on-Graph: Deep and responsible reasoning of large language model on knowledge graph. In *ICLR*, 2024.

[19] O. Khattab et al. DSPy: Compiling declarative language model calls into self-improving pipelines. arXiv preprint, 2022.

A Migration Guide

CrabPath replaces static workspace files that load every turn regardless of relevance.

A.1 Bootstrap

```
python3 scripts/bootstrap_from_workspace.py \
  /path/to/workspace --output graph.json
```

The script reads all `.md` files, splits by heading, classifies each section (fact, procedure, tool_call, guardrail), and creates edges between related sections.

A.2 Three Phases

1. **Shadow.** Run CrabPath in parallel with static files. Compare what fires vs what was needed. No risk.
2. **Reduce.** Trim TOOLS.md, MEMORY.md, AGENTS.md, USER.md. Their content lives in CrabPath nodes.
3. **Soul only.** Keep only SOUL.md (identity) + safety rules + session context. Everything else is in the graph.

What stays static: identity (SOUL.md), hard safety rules, session context. Everything else moves to the graph.

B Production Deployment

We bootstrapped a production AI agent workspace into CrabPath. The workspace had 181 markdown files with 30.7K chars of static context loaded every turn.

The agent is currently running Phase 1 (shadow mode) against this graph in production.

Metric	Value
Files scanned	181
Nodes created	3,667
Edges created	32,698
Facts	2,480
Tool calls	591
Guardrails	318
Procedures	278

Table 8: Bootstrap results from production workspace.

Query	Nodes	Chars	Result
worktree drift after codex task	4	1,931	Found reset rule, hygiene rules
bountiful auth for browser testing	4	1,188	Found browser cookie note
what are the cron jobs running	4	880	Found cron job tables
tell me about CrabPath	4	1,665	Found CrabPath section
codeword is hippo	4	1,414	Found codeword node + history

Table 9: Query results on production workspace graph. Static context: 1,656,670 chars/turn. CrabPath: ∼1,500 chars/turn. Reduction: 99.9%.