Beating (Baseline) KataGo in 4 Weeks:
A Deep, Practical Blueprint for RAG-Augmented MCTS with Alternatives, Metrics, and Engineering Details

ENGS 102 — Implementation Notes, Walkthroughs, and Research Map

October 30, 2025

Abstract

This document is a *long-form, high-detail* plan to augment a stock KataGo installation with a Retrieval-Augmented (RAG) memory that selectively assists Monte Carlo Tree Search (MCTS) when the policy head is uncertain (high-entropy). We (1) define every core concept in concrete Go terms, (2) specify what to store in the retrieval memory and why, (3) show how and when to blend retrieval information into MCTS (two specific places in the loop), (4) design pruning and maintenance policies for a small, relevant memory, (5) enumerate experiments and ablations, and (6) list realistic alternatives if RAG is too slow or brittle. The plan is scoped for 4 weeks with deliverables each week. All symbols are explained and every equation is paired with an intuition/analogy.

Contents

1 Foundations (Everything in Plain Go Terms)

1.1 The game state, rigorously but simply

Board and stones. A Go *state* is the current board position and side-to-move. We model the standard 19×19 grid as indices (x, y) with $x, y \in \{0, \ldots, 18\}$. Let $B \subseteq \{0, \ldots, 18\}^2$ be black stones, $W \subseteq \{0, \ldots, 18\}^2$ white stones, and E the empty points. Stones do not move after placement; groups with zero liberties are captured.

State variable. We write

$$s = \big(B, W, \mathrm{toMove}, \mathrm{captures}, \mathrm{ko}, \mathrm{komi}, \mathrm{rules}\big).$$

For ML input, KataGo encodes s as a stack of binary planes (who occupies each point, who moves, liberties, ko, etc.).

Action/move. An *action* a is either placing a stone at a legal empty point $p \in E$ or passing. In code, a move is typically an integer $a \in \{0, \ldots, 361\}$ mapping to a coordinate or pass.

1.2 What the neural net outputs mean (policy, value, score)

Policy. $\pi_\theta(a \mid s) \in [0, 1]$ is a probability distribution over legal moves. It is the net's *first-glance recommendation*: "these few points look promising."

Value. $v_\theta(s) \in [-1, 1]$ estimates who is winning *from the current player's perspective*: $+1$ means very likely to win, -1 to lose. KataGo can also predict expected score difference; both are equivalent signals for search guidance.

Analogy. Policy is your coach circling hotspots with a marker; value is your gut sense of the scoreboard if play ended now.

1.3 What MCTS keeps track of (precise symbols, human meaning)

For each state s in the tree and legal action a:

$$\underbrace{N(s,a)}_{\text{visits}}, \qquad \underbrace{W(s,a)}_{\text{sum of backed-up values}}, \qquad \underbrace{Q(s,a) = \frac{W(s,a)}{\max(1, N(s,a))}}_{\text{average outcome}}, \qquad \underbrace{P(s,a)}_{\text{prior from policy head}}.$$

Intuition: P is the initial *hunch*, N counts how often we've tried this move during search, Q is how well it has actually worked out on average given our simulated futures.

1.4 How MCTS *chooses* a child (selection)

On a node s, MCTS repeatedly selects a child action $a^\star$ maximizing:

$$a^\star \;=\; \arg\max_{a \in \mathcal{A}(s)} \Big[Q(s,a) + U(s,a)\Big], \qquad U(s,a) = c_{\text{puct}} \cdot P(s,a) \frac{\sqrt{N(s)}}{1 + N(s,a)}. \tag{1}$$

Terms: $N(s) = \sum_b N(s,b)$. $c_{\text{puct}} > 0$ is a knob for *how adventurous* we are: higher values encourage exploring moves with high prior P and low visits $N(s,a)$.

Analogy. You revisit what has worked (Q) but occasionally try things your coach flagged (P), especially if you haven't tried them yet ($N(s,a)$ small).

1.5 Expansion, evaluation, and backup (one full loop)

When selection hits a leaf s_L not yet expanded:

1. **Expand:** add children for legal moves; initialize $N = 0$, $W = 0$.
2. **Evaluate:** call the net once: $(\mathbf{p}, v) = \big(\pi_\theta(\cdot \mid s_L), v_\theta(s_L)\big)$, set $P(s_L, \cdot) \leftarrow \mathbf{p}$.
3. **Backup:** propagate v along the path to root, flipping sign each ply (since "current player" swaps): if the path is $s_0 \to a_0 \to s_1 \to a_1 \cdots \to s_L$, then for each step t, do

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1, \quad W(s_t, a_t) \leftarrow W(s_t, a_t) + (-1)^t v.$$

2 Our RAG-Augmented MCTS: The *Selective* Assistance Idea

2.1 Why selective? Entropy-based trigger

Policy entropy. Given the net's policy $\pi_\theta(\cdot \mid s)$, define the Shannon entropy

$$H(\pi_\theta(\cdot \mid s)) = -\sum_{a \in \mathcal{A}(s)} \pi_\theta(a \mid s) \log \pi_\theta(a \mid s). \tag{2}$$

High entropy means the probability mass is *spread out* (the net is unsure), low entropy means one/few moves dominate (the net is confident).

Trigger rule (simple, tunable). Introduce a threshold $H_{\max}$. If $H(\pi_\theta(\cdot \mid s)) \geq H_{\max}$ at the **root** $s = s_0$, we *activate RAG* to provide extra context; otherwise we skip RAG to save time.

2.2 Where to plug RAG into MCTS (two precise places)

We modify **two** of the four classic MCTS steps:

I. **Selection** $\to$ *unchanged.*

II. **Expansion** $\to$ RAG prior blend at fresh nodes (optional).

III. **Evaluation** $\to$ RAG prior blend at root (primary).

IV. **Backup** $\to$ *unchanged* (optionally tiny value nudge; see Sec. 2.5).

We keep selection and backup unchanged for simplicity and stability. Our main effect is to alter the *priors* P (the "coach's hunch") when the net is uncertain.

2.3 What exactly the RAG memory stores (rich but compact)

We store *embedding-indexed* experience tuples capturing not only the picture of the board but also behavioral context and short-term consequences.

Entry schema.

$$\texttt{RAG\_Entry} = \Big(\underbrace{\mathbf{e}_t}_{\text{state embedding}}, \underbrace{\boldsymbol{\pi}_t}_{\text{local policy dist}}, \underbrace{v_t}_{\text{value}}, \underbrace{a_t}_{\text{played move}}, \underbrace{\mathcal{T}_t}_{\text{short trajectory}}, \underbrace{\mathcal{S}_t}_{\text{MCTS stats}}, \underbrace{\text{meta}_t}_{\text{metadata}} \Big). \tag{3}$$

Definitions (one by one).

- $\mathbf{e}_t \in \mathbb{R}^d$ — the *embedding* of state s_t: a vector produced by an encoder (e.g., a pooled penultimate layer). *Analogy:* a compact "fingerprint" capturing local shapes, influence, and to-move.
- $\boldsymbol{\pi}_t \in \Delta^{|\mathcal{A}|}$ — the *local policy* distribution the net (or search) had for s_t; gives nuance beyond a single "best move".
- $v_t \in [-1, 1]$ — value estimate at s_t (or expected score).
- $a_t \in \mathcal{A}(s_t)$ — the move actually played in that game/simulation.
- $\mathcal{T}_t = \big((s_t, a_t), (s_{t+1}, a_{t+1}), \ldots, (s_{t+k}, \cdot)\big)$ — short *trajectory* of length $k \in \{2, 3\}$: tells "where this line tended to go" (critical for ladders, fights).
- $\mathcal{S}_t = (N(s_t, \cdot), Q(s_t, \cdot))$ — optional *subtree statistics* if this state was part of a previous deep search.
- meta_t — metadata like phase $\in$ {opening, mid, end}, komi, rules, capture diff, policy entropy, etc.

Why all these fields? $\mathbf{e}$ is needed for fast nearest-neighbor (**ANN**) search. $\boldsymbol{\pi}$ and v provide richer context than a single move. $\mathcal{T}$ adds short-term consequences (2–3 ply lookahead). $\mathcal{S}$ lets us *reuse search knowledge*, not just net predictions. Metadata enables **filtering** (e.g., avoid grabbing endgame patterns in the opening).

2.4 How retrieval constructs a prior (equations and alignment)

At a query state s with embedding $\mathbf{e}$, we find K nearest neighbors $\{R_k\}_{k=1}^K$ by cosine similarity $\sigma_k = \cos(\mathbf{e}, \mathbf{e}_k)$. We weight neighbors by a temperature-smoothed softmax:

$$w_k = \frac{\exp(\lambda\,\sigma_k)}{\sum_{j=1}^K \exp(\lambda\,\sigma_j)}, \qquad \lambda > 0. \tag{4}$$

We then map neighbor recommendations into the current board's coordinate system (handling rotations/reflections via an alignment function align($\cdot$)), and form a retrieval prior:

$$\tilde{P}_{\text{ret}}(a \mid s) = \sum_{k=1}^K w_k \underbrace{\text{AlignRec}\big(R_k, a\big)}_{\text{from } \boldsymbol{\pi}_k,\, a_k,\, \mathcal{S}_k}. \tag{5}$$

AlignRec can be as simple as:

$$\text{AlignRec}(R_k, a) = \begin{cases} \boldsymbol{\pi}_k\big(\text{align}^{-1}(a)\big), & \text{if using neighbor policy} \\ 1 \text{ if } a = \text{align}(a_k),\ 0 \text{ else}, & \text{if using single played move} \\ (\text{or a blend using } N, Q \text{ if available}). & \end{cases}$$

Blending at root (primary site). If the entropy trigger fires at the root s_0, we set:

$$P'(s_0, a) \;=\; (1-\beta)\, P_\theta(s_0, a) \;+\; \beta\, \tilde{P}_{\text{ret}}(a \mid s_0), \qquad \beta \in [0,1]. \tag{6}$$

β is the *influence knob.* Keep β small (0.05–0.15).

Blending at fresh leaves (optional). When we first expand a high-entropy leaf s_L, we can blend the net prior with a local retrieval prior to initialize $P(s_L, \cdot)$. This is secondary; root blending gives most of the benefit at minimal overhead.

2.5 A tiny optional value nudge (with guardrails)

If neighbors strongly agree the position type is winning/losing, we may apply a *very small* value adjustment:

$$v'(s) \;=\; \text{clip}\Big(v_\theta(s) + \gamma \cdot \underbrace{\big(\sum_k w_k\, v_k - v_\theta(s)\big)}_{\Delta_{\text{ret}}(s)},\, -1,\, +1\Big), \qquad \gamma \ll 1. \tag{7}$$

In our 4-week scope, this is optional—prior blending is safer and simpler.

2.6 Gating rules to avoid bad retrievals

We only use retrieval if:

$$H(\pi_\theta(\cdot \mid s)) \geq H_{\max} \quad \text{and} \quad \max_k \sigma_k \geq \tau_{\text{sim}} \quad \text{and} \quad \text{phase}(R_k) = \text{phase}(s). \tag{8}$$

τ_{sim} is a similarity threshold; phase matching prevents opening patterns from polluting endgame.

3 Embeddings, Similarities, and "Equilibrium" Similarity

3.1 What is an embedding, really? (and why Go needs them)

Definition. An *embedding* is a vector $\mathbf{e} \in \mathbb{R}^d$ that numerically encodes a Go position's salient features (shapes, liberties, thickness, to-move) so that similar positions have *nearby vectors.*

Analogy. Think of compressing a high-resolution photo (the whole board) into a small, meaningful thumbprint that preserves the important patterns.

How to get it. Use KataGo's trunk features (penultimate activations) and pool them (e.g., global average pooling) or train a thin projection head to produce d-dimensional vectors.

3.2 Similarity functions we will use

Cosine similarity.

$$\cos(\mathbf{e}, \mathbf{e}') = \frac{\mathbf{e} \cdot \mathbf{e}'}{\|\mathbf{e}\|\, \|\mathbf{e}'\|},$$

scale-invariant and widely used for ANN libraries.

KL divergence for policies (behavioral similarity). Given two policies π and π' over the same action set,

$$D_{\mathrm{KL}}(\pi \,\|\, \pi') = \sum_a \pi(a) \log \frac{\pi(a)}{\pi'(a)},$$

small if they prefer similar moves. We can combine *visual similarity* (cosine of embeddings) and *behavioral similarity* (inverse KL) for more robust retrieval.

3.3 "Equilibrium" similarity (reachability-aware)

Problem. Two local shapes may look alike yet be *globally* different; one position may be unreachable from the other given optimal play (fights elsewhere change life/death status).

Idea. Define a composite similarity that includes a cheap proxy for reachability:

$$S_{\mathrm{eq}}(s, s') \;=\; \alpha \cdot \cos\big(\mathbf{e}(s), \mathbf{e}(s')\big) \;+\; (1-\alpha) \cdot \big(-\overline{D}_{\mathrm{KL}}(\pi(s), \pi(s'))\big) \;-\; \eta \cdot \Delta_{\mathrm{phase}}(s, s') \;-\; \xi \cdot \Delta_{\mathrm{global}}(s, s').$$

Here $\overline{D}_{\mathrm{KL}}$ is symmetrized KL (average of $D_{\mathrm{KL}}(\pi\|\pi')$ and $D_{\mathrm{KL}}(\pi'\|\pi)$), Δ_{phase} penalizes phase mismatch, and Δ_{global} penalizes large global differences (e.g. score lead discrepancy, capture diff, or number of unsettled groups). We don't need perfect reachability; we need a quick filter to avoid misleading neighbors.

4 Memory Size, Pruning, and Online Maintenance

4.1 Small, relevant memory vs. giant archive

Two strategies.

1. **Small, curated memory** (our default): a few thousand entries, constantly pruned and refreshed.
2. **Massive archive**: many millions of entries; requires heavy hardware and careful latency engineering.

Given our 4-week scope, we choose the small curated memory.

4.2 What to *keep*: an importance score

For each candidate entry R, define an *importance*:

$$\mathrm{Imp}(R) \;=\; w_1 \cdot \mathrm{Uncertainty}(R) \;+\; w_2 \cdot \mathrm{Novelty}(R) \;+\; w_3 \cdot \mathrm{Decisiveness}(R) \;+\; w_4 \cdot \mathrm{UseCount}(R).$$

Uncertainty: high policy entropy or high value variance. *Novelty*: large distance to nearest existing entry in embedding space. *Decisiveness*: large swing in value across the short trajectory $\mathcal{T}$ (sharp tactical turning point). *UseCount*: how often retrieval has actually used this entry recently.

4.3 Pruning policy (keep it fresh)

Budget. Keep at most M entries (e.g. $M = 5{,}000$). When M is exceeded, prune the lowest-Imp entries or those dominated by near-duplicates.

Near-duplicate pruning. If two entries R, R' satisfy $S_{\text{eq}}(R, R') \geq \tau_{\text{eq}}$ and their metadata phases match, drop the one with lower Imp.

Aging. Decay importance slowly with time to favor fresher data: $\text{Imp} \leftarrow \gamma_{\text{age}} \cdot \text{Imp}$, $\gamma_{\text{age}} \in (0, 1)$.

4.4 Online maintenance (add/prune cycle during play)

When to add. When the root entropy is high and a search finishes, extract the root (and perhaps a couple of critical leaves) as candidates to add: compute $\mathbf{e}, \boldsymbol{\pi}, v, \mathcal{T}, \mathcal{S}, \text{meta}$, compute Imp, insert if above a threshold.

When to prune. Whenever size exceeds M, prune by the rule above. This yields a simple *add/prune cycle* that keeps the memory short and sharp.

5 Approximate Nearest Neighbors (ANN) and Latency Control

5.1 Index choices (practical)

Use FAISS with an HNSW or IVF+PQ index:

- **HNSW** (hierarchical navigable small world): great recall-speed tradeoff for small-to-medium M.
- **IVF+PQ**: inverted lists with product quantization; good for larger memories with compression.

5.2 Query budgeting (root-only, gated)

We *only* query at the root when entropy is high (Eq. 8). Announce a query-time budget, e.g. $\leq 2\,\text{ms}$ per root retrieval. Tune K, index parameters (efSearch for HNSW), and prefetch/cache embeddings.

5.3 Batching and caching

Cache the current root embedding $\mathbf{e}(s_0)$ and its top neighbors for re-use across a few visits if the root hasn't changed (pondering).

6 Algorithms in One Place (Pseudocode)

6.1 Selective RAG at root (primary)

Algorithm 1 Root Evaluation with Selective RAG Prior Blend

```
Require: State $s_0$; net $\pi_\theta, v_\theta$; memory index $\mathcal{M}$; thresholds $H_{\max}, \tau_{\text{sim}}$; blend $\beta$; temp $\lambda$.
 1: $(\mathbf{p}, v) \leftarrow \big(\pi_\theta(\cdot \mid s_0), v_\theta(s_0)\big)$
 2: $H \leftarrow -\sum_a \mathbf{p}(a) \log \mathbf{p}(a)$
 3: if $H \geq H_{\max}$ then
 4:     $\mathbf{e} \leftarrow \text{Embed}(s_0)$
 5:     $\{(R_k, \sigma_k)\}_{k=1}^{K} \leftarrow \text{ANN\_query}(\mathcal{M}, \mathbf{e})$
 6:     if $\max_k \sigma_k \geq \tau_{\text{sim}}$ and $\text{phase}(R_k) = \text{phase}(s_0)$ then
 7:         Compute weights $w_k$ via Eq. (4)
 8:         Build $\tilde{P}_{\text{ret}}$ via Eq. (5)
 9:         $\mathbf{p} \leftarrow (1-\beta)\mathbf{p} + \beta \tilde{P}_{\text{ret}}$                ▷ Eq. (6)
10:     end if
11: end if
12: return $(\mathbf{p}, v)$
```

6.2 Optional: fresh-leaf prior blend

Algorithm 2 Leaf Initialization with Optional RAG

```
Require: New leaf $s_L$; net $\pi_\theta$; memory index $\mathcal{M}$; small $\beta_{\text{leaf}}$.
 1: $\mathbf{p} \leftarrow \pi_\theta(\cdot \mid s_L)$
 2: $H \leftarrow -\sum_a \mathbf{p}(a) \log \mathbf{p}(a)$
 3: if $H \geq H_{\max}^{\text{leaf}}$ then
 4:     $\mathbf{e} \leftarrow \text{Embed}(s_L)$
 5:     $\{(R_k, \sigma_k)\}_{k=1}^{K'} \leftarrow \text{ANN\_query}(\mathcal{M}, \mathbf{e})$
 6:     if $\max_k \sigma_k \geq \tau_{\text{sim}}$ and phase matches then
 7:         Build $\tilde{P}_{\text{ret}}$ as before
 8:         $\mathbf{p} \leftarrow (1-\beta_{\text{leaf}})\mathbf{p} + \beta_{\text{leaf}} \tilde{P}_{\text{ret}}$
 9:     end if
10: end if
11: Initialize $P(s_L, \cdot) \leftarrow \mathbf{p}$
```

6.3 Memory add/prune cycle

Algorithm 3 Online Memory Maintenance

```
Require: Memory $\mathcal{M}$, cap $M$, thresholds $(\text{Imp}_{\min}, \tau_{\text{eq}})$
 1: Add: for high-entropy roots (and a few critical leaves) after search, build candidate entries;
    compute Imp; insert if $\text{Imp} \geq \text{Imp}_{\min}$
 2: if $|\mathcal{M}| > M$ then
 3:     Prune near-duplicates: for pairs $(R, R')$ with $S_{\text{eq}}(R, R') \geq \tau_{\text{eq}}$, drop the one with lower
    Imp
 4:     Prune low importance: drop lowest Imp until $|\mathcal{M}| \leq M$
 5: end if
```

7 Concrete Code Snippets (Readable, Minimal)

7.1 Turn neighbors into a prior and blend

Listing 1: Neighbors → retrieval prior → blended prior.

```python
import numpy as np
from collections import defaultdict

def softmax(x, temp=1.0):
    x = np.array(x, dtype=np.float32)
    x = x - x.max()
    e = np.exp(x / max(1e-6, temp))
    return e / (e.sum() + 1e-8)

def build_ret_prior(neighbors, legal_moves, align_move, sim_temp=0.125):
    # neighbors: list of dicts like {"sim": float, "policy": {mv: p, ...}} OR {"played": "
        Q16"}
    sims = [nb["sim"] for nb in neighbors]
    ws   = softmax(sims, temp=sim_temp)
    votes = defaultdict(float)
    for w, nb in zip(ws, neighbors):
        if "policy" in nb:
            for mv, p in nb["policy"].items():
                mv2 = align_move(mv)
                if mv2 in legal_moves:
                    votes[mv2] += w * p
        else:
            mv2 = align_move(nb["played"])
            if mv2 in legal_moves:
                votes[mv2] += w
    out = {m: votes.get(m, 0.0) for m in legal_moves}
    Z = sum(out.values()) or 1.0
    for m in out: out[m] /= Z
    return out

def blend_prior(nn_prior, ret_prior, beta=0.10):
    out = {}
    Z = 0.0
    for m, p in nn_prior.items():
        q = (1 - beta)*p + beta*ret_prior.get(m, 0.0)
        out[m] = q; Z += q
    for m in out: out[m] /= (Z or 1.0)
    return out
```

7.2 Entropy gating and phase match

Listing 2: Gating: use retrieval only when it makes sense.

```python
def should_use_retrieval(policy_dict, best_sim, same_phase, Hmax=3.5, sim_thresh=0.35):
    # policy_dict: dict move->prob
    H = -sum(p*np.log(max(p,1e-12)) for p in policy_dict.values())
    return (H >= Hmax) and (best_sim >= sim_thresh) and same_phase
```

7.3 Pseudocode for Monte Carlo Tree Search

Algorithm 4 Monte Carlo Tree Search (MCTS)

```
1: procedure MCTS($s_0$)
2:     $root \leftarrow$ Node($s_0$)
3:     while not time_limit_reached() do
4:         $node \leftarrow$ SELECT($root$)
5:         $child \leftarrow$ EXPAND($node$)
6:         $outcome \leftarrow$ SIMULATE($child$)
7:         BACKPROPAGATE($child, outcome$)
8:     end while
9:     return $\arg\max_a \frac{root.Q[a]}{root.N[a]}$                    ▷ Best move
10: end procedure
11:
12: procedure SELECT($node$)                    ▷ Traverse tree guided by UCT
13:     while $node$ is fully expanded and not terminal($node$) do
14:         $node \leftarrow \arg\max_{a \in \text{Actions}(node)} \left[ \frac{Q(node,a)}{N(node,a)} + c\sqrt{\frac{\log N(node)}{N(node,a)}} \right]$
15:     end while
16:     return $node$
17: end procedure
18:
19: procedure EXPAND($node$)                    ▷ Create new child for unexplored action
20:     $a \leftarrow$ untried_action($node$)
21:     $s' \leftarrow$ simulate_environment($node.state, a$)
22:     $child \leftarrow$ Node($s'$)
23:     $node$.add_child($child, a$)
24:     return $child$
25: end procedure
26:
27: procedure SIMULATE($node$)                    ▷ Rollout using random or policy network
28:     $state \leftarrow node.state$
29:     while not terminal($state$) do
30:         $a \leftarrow$ rollout_policy($state$)          ▷ e.g., random or policy_net.forward($state$)
31:         $state \leftarrow$ step($state, a$)
32:     end while
33:     return outcome($state$)                    ▷ Final reward
34: end procedure
35:
36: procedure BACKPROPAGATE($node, outcome$)                    ▷ Update value and visit counts
37:     while $node \neq$ null do
38:         $node.N \leftarrow node.N + 1$
39:         $node.Q \leftarrow node.Q + \frac{(outcome - node.Q)}{node.N}$
40:         $node \leftarrow node.parent$
41:     end while
42: end procedure
```

7.4 Our Planned Changes

Algorithm 5 Simulation with Retrieval-Augmented Rollout

```
procedure SIMULATE($node$)
    $state \leftarrow node.state$
                                                  ▷ Trigger condition for RAG rollout
    if is_complex($state$) then          ▷ Complexity determined by entropy/variance trigger
        $retrieved \leftarrow$ RAG_RETRIEVE($state$)          ▷ Fetch similar states/subtrees
                                              ▷ Blend retrieved priors into rollout policy
        $rollout\_policy \leftarrow$ combine(policy_net, $retrieved$)
    else
        $rollout\_policy \leftarrow$ policy_net
    end if

    while not terminal($state$) do
        $a \leftarrow$ sample($rollout\_policy(state)$)
        $state \leftarrow$ step($state, a$)
    end while
    return outcome($state$)
end procedure
```

Algorithm 6 Expansion with Retrieval-Augmented Priors

```
procedure EXPAND($node$)
    $a \leftarrow$ untried_action($node$)
    $s' \leftarrow$ simulate_environment($node.state, a$)
    $child \leftarrow$ Node($s'$)
                                         ▷ Retrieve similar states to estimate better prior
    $retrieved \leftarrow$ RAG_RETRIEVE($s'$)
    $child.P \leftarrow$ mix(policy_net($s'$), prior_from($retrieved$))
    $node$.add_child($child, a$)
    return $child$
end procedure
```

8 Week-by-Week Plan (4 Weeks, With Deliverables)

Week 1: Baseline KataGo + Tools + Datasets

- Install KataGo (use analysis engine). Verify GPU and `benchmark`.
- Write Python client to feed boards and read policy/value/ownership.
- Parse SGFs $\rightarrow$ build a table of positions: $(s, \text{toMove}, a^\star, \text{phase}, \text{komi})$.
- Export a test set of tricky mid-game positions; compute net policy entropies H.
- Deliverable: baseline fixed-visit matches (e.g., 400–800 visits) and a CSV of $(\text{posID}, H, \text{top-5 moves}, v)$.

Week 2: Embeddings + ANN Index + Memory Schema

- Expose an embedding head (pooled trunk or thin projection); dimension $d \in [256, 1024]$.
- Build FAISS HNSW index; create `RAG_Entry` objects per Eq. (3).
- Populate with $\sim$ 2,000 curated entries from pro games/self-play focusing on medium/high entropy states.
- Implement alignment (rotations/reflections) for corner/side positions.
- Deliverable: nearest-neighbor demos (visual sanity checks), latency measurement ($\leq 2\,\text{ms}$ per query).

Week 3: Root-Only RAG Integration + Gating + Pruning

- Implement selective root blend (Eq. 6) with gating (Eq. 8).
- Tune $K \in \{8, 16, 24\}$, $\beta \in \{0.05, 0.10, 0.15\}$, $\lambda \in \{4, 8, 12\}$, $H_{\max}$.
- Implement online add/prune cycle with cap $M = 5{,}000$; near-duplicate pruning via S_{eq}.
- Deliverable: 400-game matches per color at fixed visits vs. baseline; report win rate, average latency, and ablations (RAG on/off).

Week 4: Optional Leaf Blend + Light Tune + Alternatives

- Optional: small leaf blend with $\beta_{\text{leaf}} \leq 0.05$ (only for very high-entropy leaves).
- Light encoder tune (2–4 epochs): policy CE + small InfoNCE to improve clustering.
- Evaluate alternatives (Sec. 11) if RAG latency too high.
- Deliverable: final report with win rates, speed, stability; curated memory release; code and config.

9 What to Research/Decide (Checklist with Rationale)

1. **Embedding dimension** d: tradeoff between expressivity and ANN speed. Start $d = 384$ or 512.
2. **Similarity mix**: pure cosine vs. mixed with policy KL (reachability proxy).
3. **Gating thresholds**: $H_{\max}$ (entropy), τ_{sim} (similarity), and phase rules.
4. **Neighbor weighting** λ: sharper (larger λ) trusts the best match more.
5. **Blend weights**: β (root), β_{leaf} (leaf).
6. **Memory cap** M and **importance weights** $(w_1, \ldots, w_4)$.
7. **Pruning threshold** $\tau_{\mathbf{eq}}$ for near-duplicates.
8. **Trajectory length** k (2 or 3 plies is usually enough).
9. **ANN index parameters**: ef_Construction, ef_Search (HNSW), or IVF lists and PQ bits.
10. **Latency budget**: max retrieval time per root; decide to drop retrieval if exceeded.

10 Evaluation Suite (What Success Looks Like)

10.1 Core metrics

- **Win rate @ fixed visits**: e.g., 800 visits per move; 200+ games per color; confidence intervals.
- **Search efficiency**: agreement with a high-visit oracle (e.g., 20k visits) on tough positions.
- **Latency**: mean and 95th percentile per move; fraction of moves where retrieval was activated.
- **Stability**: variance of winrate across seeds, robustness across rule sets and komi.

10.2 Ablations

- RAG off (baseline).
- RAG root-only vs. root+leaf.
- With vs. without policy-KL in similarity.
- Small vs. large K; small vs. large β.
- Small memory ($M = 2k$) vs. larger ($M = 10k$).

11 Alternatives Beyond RAG (Feasible within 4 Weeks)

11.1 A. Pattern-Prior Head (local shapes)

Train a small auxiliary head that outputs priors from local $n \times n$ patterns (e.g. 7×7) using a curated joseki/tesuji dataset. Blend with net prior:

$$P'(s,a) = (1-\alpha)P_\theta(s,a) + \alpha P_{\text{pattern}}(s,a).$$

Pros: very low latency. *Cons:* less flexible than retrieval.

11.2 B. Uncertainty-Calibrated Exploration

Adapt c_{puct} based on entropy/variance:

$$c_{\text{puct}}(s) = c_0 \cdot \big(1 + \kappa \cdot \frac{H(\pi_\theta(\cdot \mid s))}{\log |\mathcal{A}(s)|}\big).$$

Encourages broader search only when the net is unsure.

11.3 C. Speculative Policy (draft-and-verify)

Use a fast, shrunken policy head to propose a short list of moves; run deeper verification on those only (a form of selective expansion). This is compatible with or without RAG.

11.4 D. Score-Utility Tweaks (KataGo-specific)

KataGo already models score; you can adjust utility shaping (e.g., reduce slack in endgame by slightly increasing score-sensitivity). Keep changes small and validate carefully.

12 FAQ, Pitfalls, and Safeguards

Will RAG make it "too human" and miss superhuman lines? We only *nudge* priors with small β; MCTS still verifies via Q. If a humanish move is actually bad, its Q will sink and search will abandon it.

Is ANN too slow? Root-only, gated by entropy, with small K and HNSW ef_Search tuned, is typically within a few ms on a modern CPU/GPU host. Measure and cap.

What if retrieval returns mismatched contexts? Phase filtering, similarity thresholding, and optional policy-KL help; if best_sim$<\tau_{\text{sim}}$, we skip.

Representation drift? If you lightly tune the encoder, periodically re-encode a sample of memory entries or bias to fresher ones.

13 Glossary (Every Symbol, Human Meaning)

s	Go state (board, to-move, rules, etc.).
a	Legal move (a coordinate or pass).
$\pi_\theta(a \mid s)$	Policy (move probabilities) from the net.
$v_\theta(s)$	Value (winrate or score) from the net.
$P(s, a)$	Prior stored in the search node s for move a.
$N(s, a)$	Visit counts; how many times search tried a at s.

$W(s,a)$	Sum of backed-up values along simulations through (s,a).
$Q(s,a)$	Average outcome W/N.
$c_{\mathbf{puct}}$	Exploration parameter in Eq. (1).
$\mathbf{e}$	Embedding (vector fingerprint) of a position.
σ_k	Similarity score (e.g., cosine) to neighbor k.
w_k	Weight of neighbor k (Eq. 4).
$\tilde{P}_{\mathbf{ret}}$	Retrieval-based prior (Eq. 5).
β	Blend weight for retrieval (root); small (0.05–0.15).
$\beta_{\mathbf{leaf}}$	Blend at fresh leaves (optional, even smaller).
H	Shannon entropy of policy; how spread-out/confused it is.
$H_{\max}$	Entropy threshold to trigger retrieval.
$\tau_{\mathbf{sim}}$	Similarity threshold to accept nearest neighbors.
λ	Temperature shaping neighbor weights.
γ	Small value-nudge weight (usually 0).
M	Memory cap (max entries).
Imp	Importance score guiding which entries to keep.

14 Closing: What We Will Actually Build, Physically

Code changes (small, surgical).

- A Python (or C++) module to *embed* states and query FAISS.
- Hooks at the *root evaluation* (and optionally leaf expansion) to call retrieval only when $H \geq H_{\max}$.
- A blender to mix $\tilde{P}_{\text{ret}}$ with P_θ (Eq. 6).
- A memory manager that adds/prunes entries online using Imp.
- A match runner to evaluate win rate and latency @ fixed visits.

What success looks like in 4 weeks.

- $\geq$3–5% win-rate gain vs. baseline at the *same* visit budget on a balanced test suite, *or*
- Same win rate with $\sim$5–10% lower visits (i.e., efficiency gain).
- Latency overhead < 3 ms/move on average; retrieval used only on $\sim$20–40% of positions (high-entropy ones).

Mindset. We are not rewriting KataGo. We are giving it a pocket playbook it opens only when its instincts say, "this is complicated." Then MCTS does what it always does best: verify.