TAKO

Hierarchical Reasoning Chess Engine

Project Specification v1.0

Goal: Grandmaster-level chess via self-play RL with minimal compute and minimal heuristics

Architecture
Hierarchical Reasoning Model (HRM)

Parameters
~27M

Target Elo
2500+ (GM level)

Training
Self-play RL + Distributed Pipeline

1. Vision & Core Tenets

Tako is a chess engine built around the Hierarchical Reasoning Model (HRM) — a novel recurrent architecture inspired by multi-timescale processing in the brain. The project aims to demonstrate that grandmaster-level chess play is achievable at low compute by replacing explicit search depth with implicit iterative reasoning.

> **The Four Immutable Tenets**
>
> 1. Primarily trained through self-play reinforcement learning
> 2. Not a re-implementation of an existing engine (not Leela, Stockfish, or AlphaZero)
> 3. Must reach at least GM-level play (2500+ Elo)
> 4. Uses a distributed learning pipeline for scalable data generation

1.1 What Makes Tako Novel

Most competitive chess engines (Leela Chess Zero, AlphaZero) use convolutional or transformer policy networks as static function approximators, relying on MCTS to provide reasoning depth at inference time. Tako instead uses HRM as the neural backbone: a dual-module recurrent architecture that performs iterative reasoning within a single forward pass.

This means Tako's policy network is not a static board evaluator — it actively "thinks" over the position by running multiple high-level/low-level computation cycles before producing a move distribution or value estimate. MCTS then operates on top of this richer per-node evaluation.

The natural mapping to chess is direct:

- The H-module (slow, high-level) handles strategic planning: pawn structure, king safety, long-term piece coordination
- The L-module (fast, low-level) handles tactical computation: material counting, immediate threats, forced lines
- ACT (adaptive computation time) allocates more thinking cycles to complex positions and fewer to simple ones

2. HRM Architecture — Full Specification

HRM consists of four learnable components operating over a structured two-level temporal hierarchy. All components are standard transformer blocks using modern LLM best practices.

2.1 The Four Learnable Components

2.1.1 Input Network f_I(x; θ_I)

Projects the raw board representation x into a working embedding x̃. For Tako, this is a token embedding layer followed by Rotary Positional Encoding (RoPE). The input is a sequence of tokens representing the FEN-encoded board state.

Component	Detail
Input	Tokenized FEN string (board tokens + side to move + castling + en passant)
Output	Dense embedding sequence x̃ ∈ ℝ^{seq_len × d_model}
Implementation	Embedding lookup + RoPE positional encoding

2.1.2 Low-Level Module f_L(·; θ_L)

A fast, tactical computation module. Updates every timestep. Responsible for detailed local pattern recognition — analogous to gamma-wave activity (30–100 Hz) in the brain.

At each timestep i, the L-module updates its hidden state conditioned on three inputs:

- Its own previous state z^{i-1}_L
- The current H-module state z^{i-1}_H (held fixed throughout the cycle)
- The input embedding x̃

```
z^i_L = f_L(z^{i-1}_L, z^{i-1}_H, x̃; θ_L)

# Multiple inputs combined via element-wise addition before transformer block
```

2.1.3 High-Level Module f_H(·; θ_H)

A slow, strategic planning module. Updates only once per cycle, after the L-module has completed T timesteps and converged to a local equilibrium. Analogous to theta-wave activity (4–8 Hz).

```
# H-module update rule (every T steps):
z^i_H = f_H(z^{i-1}_H, z^{i-1}_L; θ_H)  if i ≡ 0 (mod T)
z^i_H = z^{i-1}_H                         otherwise
```

The H and L modules have identical Transformer block architectures and hidden dimensions. The difference is entirely in their update frequency.

2.1.4 Output Network f_O(·; θ_O)

Reads from the final H-module state after all N cycles complete and produces the network's predictions. For Tako, this is two heads:

Head	Description
Policy Head	Softmax distribution over all legal moves (action space = full UCI move set, ~1968 moves). Predicts π(a|s).
Value Head	Win/Draw/Loss probabilities for current player. Predicts V(s) = [P(win), P(draw), P(loss)].

Both heads read from z^{NT}_H, the H-module's state after completing all N cycles of T steps.

2.2 Forward Pass — Step-by-Step

A single forward pass (one "segment") unfolds over N × T total timesteps:

1. Input embedding: x̃ = f_I(x; θ_I)
2. Initialize hidden states: z^0_H, z^0_L sampled from truncated normal (std=1, trunc=2), held fixed across training
3. For each cycle k = 1..N, run T L-module updates
4. At end of each cycle, run one H-module update using L's final state
5. After N full cycles, extract prediction: ŷ = f_O(z^{NT}_H; θ_O)
6. ACT Q-head decides: halt (return ŷ) or continue (run another segment)

```
def hrm_forward(z, x, N=4, T=4):
    x_emb = input_network(x)            # f_I
    zH, zL = z
    with torch.no_grad():
        for i in range(N * T - 1):
            zL = L_module(zL, zH, x_emb)  # f_L updates every step
            if (i + 1) % T == 0:
                zH = H_module(zH, zL)     # f_H updates every T steps
    # 1-step gradient: only final step has grad enabled
    zL = L_module(zL, zH, x_emb)
    zH = H_module(zH, zL)
    policy, value = output_head(zH)      # f_O
    return (zH, zL), policy, value
```

2.3 Hierarchical Convergence

Standard RNNs fail at depth because they converge to a fixed point early, making further computation inert. HRM prevents this via hierarchical convergence:

- Within a cycle, the L-module runs T steps and converges toward a local equilibrium dependent on the current H-state
- At cycle end, the H-module incorporates this equilibrium and updates — establishing a new context for the L-module
- The L-module must now converge to a different local equilibrium, effectively "restarting" its computational trajectory
- This produces NT effective computational depth rather than just T or N separately

> **Key Insight**
> The H-module's periodic updates prevent the L-module from settling permanently, sustaining high computational activity across all NT steps. Empirically this produces sustained forward residuals (network stays active) versus rapid decay in standard RNNs.

2.4 Transformer Block Implementation

Both f_L and f_H use encoder-only Transformer blocks with identical architecture, incorporating modern LLM improvements based on Llama:

Component	Implementation
Positional Encoding	Rotary Positional Encoding (RoPE) — no learned absolute positions
Feed-Forward	Gated Linear Units (GLU) — improved expressivity over standard FFN
Normalization	RMSNorm — more stable than LayerNorm, no bias terms
Layout	Post-Norm (normalization after residual addition)
Attention	FlashAttention 2/3 for memory-efficient attention computation
Input Merging	Element-wise addition of multiple module inputs before block
Biases	Removed from all linear layers

3. Training System

HRM training replaces BPTT entirely with three nested mechanisms. Together they provide O(1) memory overhead, frequent gradient feedback, and adaptive compute allocation.

3.1 One-Step Gradient Approximation

Standard BPTT through N×T timesteps requires O(N×T) memory — intractable for large configurations. HRM uses a 1-step approximation grounded in Deep Equilibrium Model theory via the Implicit Function Theorem.

The key insight: if the L-module converges to a fixed point z*_L within each cycle, the IFT gives the exact gradient without unrolling. Approximating (I - J_F)^{-1} ≈ I (first Neumann series term) yields:

```
∂z*_H/∂θ_H ≈ ∂f_H/∂θ_H
∂z*_H/∂θ_L ≈ ∂f_H/∂z*_L · ∂f_L/∂θ_L
∂z*_H/∂θ_I ≈ ∂f_H/∂z*_L · ∂f_L/∂θ_I
```

In practice: run N×T - 1 steps inside torch.no_grad(), then run the final single step with gradients enabled. This gives O(1) memory regardless of N and T.

The gradient path is: Output Head → final H-state → final L-state → Input Embedding

3.2 Deep Supervision

A single forward pass may not provide sufficient gradient signal to the H-module for learning long-horizon strategies. Deep supervision runs multiple segments (forward passes) sequentially, computing loss and updating parameters after each.

Critical detail: the hidden state z is detached between segments. This creates a sequence of independent 1-step approximations, providing more frequent feedback to the H-module and acting as a regularizer.

```
# Deep supervision training loop
for x, y_true in dataloader:
    z = z_init                              # fixed initialization
    for step in range(N_supervision):
        z, policy, value = hrm_forward(z, x)
        loss = policy_loss + value_loss
        z = z.detach()    # ← critical: no grad flow across segments
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

3.3 Adaptive Computation Time (ACT)

ACT enables dynamic allocation of compute per board position — spending more segments on complex middlegame positions and fewer on simple endgames. This is HRM's "System 1 / System 2" mechanism.

Q-Head

A small linear head reads from z^{NT}_H after each segment and predicts Q-values for two actions:

```
Q̂^m = σ(θ_Q^T · z^{mNT}_H)     # sigmoid, outputs [Q_halt, Q_continue]
```

Halting Strategy

Uses stochastic minimum segment enforcement to balance compute efficiency with exploration of longer reasoning chains:

- With probability ε: sample M_min uniformly from {2, ..., M_max} (force longer thinking)
- With probability 1-ε: set M_min = 1 (allow early halting)
- Halt when: segment count > M_max, OR (Q_halt > Q_continue AND segments ≥ M_min)

Q-Head Training

Trained via Q-learning on an episodic MDP with sparse binary reward:

```
G_halt     = 1{ŷ^m = y}                          # 1 if correct, 0 if wrong
G_continue = max(Q̂^{m+1}_halt, Q̂^{m+1}_continue)   # bootstrap from next segment
```

3.4 Full Objective Function

Each supervision segment m contributes a combined loss. The total loss over one training example is the sum across all M segments executed before halting:

```
# Per-segment loss (game setting):
L^m = α · PolicyLoss(π̂^m, π_target)
    + β · ValueLoss(v̂^m, z_outcome)
    + γ · BinaryCrossEntropy(Q̂^m, Ĝ^m)

# Policy loss: cross-entropy against MCTS visit distribution
PolicyLoss = -Σ_a π_target(a) · log(π̂^m(a))

# Value loss: cross-entropy against game outcome [W, D, L]
ValueLoss  = -Σ_r z(r) · log(v̂^m(r))

# Total:
L_total = Σ_{m=1}^{M} L^m
```

Hyperparameter	Default

α (policy weight)	1.0
β (value weight)	1.0
γ (ACT weight)	0.1
N_supervision segments	5–10
M_max (max thinking depth)	5–20
ε (long-thinking probability)	0.15

3.5 Self-Play & MCTS Integration

Tako uses MCTS with HRM as the policy/value network. HRM is not a static evaluator — each MCTS node evaluation runs a full HRM forward pass (potentially multiple segments via ACT). This provides richer search signal at the cost of slower per-node evaluation, which is compensated by HRM's better evaluation quality.

Parameter	Value
MCTS Simulations	200–800 per move (configurable)
Exploration	UCB / PUCT with Dirichlet noise at root
Policy Target	Visit count distribution after MCTS (with policy target pruning)
Value Target	Game outcome $z \in \{\text{win}, \text{draw}, \text{loss}\}$ from self-play result
Forced Playouts	KataGo-style forced playouts + policy target pruning
Temperature	$\tau=1.0$ for first 30 moves, $\tau \to 0$ thereafter

4. Proof-of-Concept Plan — Simpler Games First

Before committing to chess, the HRM architecture will be validated on two simpler two-player perfect information games. This confirms the architecture generalizes before investing chess-scale compute.

4.1 Stage 1 — Othello (Reversi)

Othello is the primary validation target. It has complex positional dynamics with dramatic material swings requiring both local tactics and global strategy — a direct test of the dual-module design.

Property	Value
Board	8×8 grid, 64 positions
State Space	~10^28 (large enough to stress generalization)
Move Branching	~10 legal moves average
Game Length	~58 moves average
Reference Bot	Edax (world-class Othello engine), Logistello
Target	Defeat Edax at level 5+ (strong amateur)
Key Validation	H-module learns corner control / mobility strategy; L-module handles disc counting

> **Why Othello First**
>
> Games are short (~3 min), state space is tractable, strong bots exist for benchmarking, and the strategic concepts (corners, edge play, parity) create a clear distinction between tactical and strategic reasoning that HRM's modules should exhibit.

4.2 Stage 2 — Hex (11×11)

Hex tests generalization to a topologically different game with no draws and a very different winning condition. Success here guards against Othello-specific overfitting.

Property	Value
Board	11×11 rhombus grid (121 cells)
State Space	~10^57
Key Property	Always has a winner; no draws possible
Strategy	Requires both local group connectivity (L-module) and global path planning (H-module)
Reference	MoHex, Benzene (strong Hex engines)

Target	Reach intermediate strength vs MoHex

4.3 Stage 3 — Chess

After validating the HRM approach on both Othello and Hex, the architecture scales to chess with confidence. The self-play pipeline, distributed training, and MCTS integration are reused directly.

Phase	Detail
Bootstrap	Supervised pretraining on ~1M grandmaster PGN games (targets ~1800 Elo start)
Self-Play	Full HRM self-play with distributed worker fleet
Target	2500+ Elo (GM level) on Lichess blitz
Benchmark	Stockfish at various depth limits, Leela Chess Zero

5. Project Structure

The codebase is organized around three separable concerns: game environments, the HRM model, and the training/evaluation infrastructure.

5.1 Directory Layout

```
tako/
│
├── tako-hex-core/              # Rust crate: Hex game logic (Phase 2)
│   ├── Cargo.toml              # Crate manifest (pyo3 feature flag)
│   └── src/
│       ├── lib.rs              # PyO3 module entry point (#[pymodule])
│       ├── board.rs            # Hex board state (bitboard representation)
│       ├── moves.rs            # Legal move generation
│       ├── win_check.rs        # Union-Find connectivity (winner detection)
│       └── zobrist.rs          # Zobrist hashing for transposition table
│
├── tako-chess-core/            # Rust crate: Chess game logic (Phase 3+)
│   ├── Cargo.toml
│   └── src/
│       ├── lib.rs              # PyO3 module entry point
│       ├── board.rs            # Chess board (bitboard, mailbox)
│       ├── movegen.rs          # Pseudo-legal + legal move generation
│       ├── tokenizer.rs        # FEN → token sequence (matches Python tokenizer)
│       └── zobrist.rs          # Zobrist hashing
│
├── tako-worker/                # Rust binary: distributed self-play worker (Phase
4)
│   ├── Cargo.toml
│   └── src/
│       ├── main.rs             # Worker entry point + CLI args
│       ├── mcts.rs             # MCTS tree search (calls inference server)
│       ├── inference_client.rs # IPC client → Python GPU inference server
│       ├── self_play.rs        # Game loop, experience collection
│       └── sender.rs           # Sends experience batches to Python learner
│
├── Cargo.toml                  # Workspace manifest (all Rust crates)
├── pyproject.toml              # Maturin build config (PyO3 → .so extension)
│
├── games/                      # Python game environment abstractions
│   ├── base.py                 # Abstract BaseGame interface
│   ├── othello.py              # Othello (pure Python, Phase 1)
│   ├── hex.py                  # Hex Python wrapper → tako_hex_core (Phase 2)
│   └── chess/                  # Chess Python wrapper → tako_chess_core (Phase 3)
│       ├── board.py            # Thin wrapper over Rust board
│       └── tokenizer.py        # FEN tokenizer (delegates to Rust)
│
├── model/                      # HRM neural network
│   ├── hrm.py                  # Core HRM forward pass
│   ├── modules.py              # H-module, L-module, Input/Output networks
```

```
│   ├── transformer.py          # Shared Transformer block (RoPE, GLU, RMSNorm)
│   ├── act.py                  # Adaptive Computation Time + Q-head
│   └── heads.py                # Policy head, Value head
│
├── search/                     # MCTS implementation (Python, Phases 1–3)
│   ├── mcts.py                 # Core MCTS with PUCT
│   ├── node.py                 # MCTSNode (visit counts, Q-values)
│   └── forced_playouts.py      # KataGo-style forced playouts + pruning
│
├── training/                   # Training pipeline
│   ├── self_play.py            # Python self-play worker (Phases 1–3)
│   ├── learner.py              # Central learner (loss, optimizer, checkpoint)
│   ├── replay_buffer.py        # Experience replay buffer
│   ├── distributed.py          # Worker ↔ learner communication (Ray / IPC)
│   ├── inference_server.py     # GPU inference server for Rust workers (Phase 4)
│   └── supervisor.py           # Deep supervision training loop
│
├── eval/                       # Evaluation suite
│   ├── arena.py                # Pit models against each other
│   ├── elo.py                  # Bayesian Elo estimation
│   └── benchmarks.py           # vs Stockfish, Edax, MoHex etc.
│
├── engine/                     # UCI interface for chess GUIs
│   ├── uci.py                  # UCI protocol handler
│   └── tako_engine.py          # Engine wrapper (analyze + play)
│
├── config/
│   ├── othello.yaml            # Othello experiment config
│   ├── hex.yaml                # Hex experiment config
│   └── chess.yaml              # Chess experiment config
│
├── scripts/
│   ├── train.py                # Launch training run
│   ├── eval.py                 # Run evaluation suite
│   └── pretrain.py             # Supervised pretraining on PGN/game logs
│
└── tests/
    ├── test_hrm.py             # HRM forward pass unit tests
    ├── test_games.py           # Game environment correctness (Python + Rust)
    ├── test_mcts.py            # MCTS unit tests
    └── test_bindings.py        # PyO3 binding integration tests
```

5.2 Core Interfaces

BaseGame Interface

All game environments implement a common interface so the training loop is game-agnostic:

```
class BaseGame(ABC):
    @abstractmethod
```

```
    def reset(self) -> State: ...

    @abstractmethod
    def legal_moves(self) -> List[int]: ...

    @abstractmethod
    def make_move(self, move: int) -> None: ...

    @abstractmethod
    def is_terminal(self) -> bool: ...

    @abstractmethod
    def outcome(self) -> float:
        """Returns 1.0 (current player won), 0.0 (draw), -1.0 (lost)"""

    @abstractmethod
    def to_tokens(self) -> torch.Tensor:
        """Tokenize game state for HRM input"""

    @abstractmethod
    def action_size(self) -> int: ...
```

HRM Interface

```
class HRM(nn.Module):
    def forward(
        self,
        x: torch.Tensor,           # [batch, seq_len] token ids
        z: Optional[HRMState],     # (zH, zL) hidden states, None = use z_init
        n_cycles: int = 4,         # N: high-level cycles
        t_steps: int = 4,          # T: low-level steps per cycle
    ) -> Tuple[HRMState, PolicyOutput, ValueOutput]:
        """Single forward pass (one segment). Returns updated state +
predictions."""

    def predict(self, x, use_act=True) -> Tuple[PolicyOutput, ValueOutput]:
        """Full inference with ACT: runs segments until halt decision."""
```

5.3 Rust / PyO3 Layer

The Rust layer is introduced in two distinct phases with different integration patterns. The Cargo workspace at the repo root contains all Rust crates and is built via Maturin, which compiles PyO3 extensions into native .so files importable directly from Python.

Phase 2 — PyO3 Bindings for Hex (tako-hex-core)

The Hex game environment is reimplemented in Rust and exposed to Python via PyO3 bindings. Python code calls into Rust as if it were a regular Python module — no IPC, no serialization, just a direct native function call.

> **Learning Goal**
> Hex is chosen for this introduction because the game logic is simple (no rule edge cases), making it easy to validate correctness of the bindings. The patterns learned here — PyO3 structs, #[pymethods], returning numpy arrays, GIL handling — transfer directly to the chess crate.

The Python hex.py wrapper calls the compiled Rust extension transparently:

```
# games/hex.py  — Python side stays unchanged from training loop's perspective
import tako_hex_core as _hex   # compiled Rust .so via maturin

class HexGame(BaseGame):
    def __init__(self, size: int = 11):
        self._board = _hex.HexBoard(size)    # Rust struct via PyO3

    def legal_moves(self) -> List[int]:
        return self._board.legal_moves()     # returns Python list from Rust Vec<u8>

    def make_move(self, move: int) -> None:
        self._board.make_move(move)

    def is_terminal(self) -> bool:
        return self._board.is_terminal()

    def to_tokens(self) -> torch.Tensor:
        # Rust returns numpy array → torch.from_numpy() (zero-copy)
        return torch.from_numpy(self._board.to_array())
```

The Rust side uses PyO3 to expose the HexBoard struct:

```
// tako-hex-core/src/lib.rs
use pyo3::prelude::*;
use numpy::PyArray1;

#[pyclass]
struct HexBoard {
    board: Board,   // internal Rust struct (bitboard representation)
}

#[pymethods]
impl HexBoard {
    #[new]
    fn new(size: usize) -> Self { HexBoard { board: Board::new(size) } }
```

```
    fn legal_moves(&self) -> Vec<u8> { self.board.legal_moves() }

    fn make_move(&mut self, mv: u8) { self.board.make_move(mv) }

    fn is_terminal(&self) -> bool { self.board.is_terminal() }

    fn to_array<'py>(&self, py: Python<'py>) -> &'py PyArray1<f32> {
        PyArray1::from_vec(py, self.board.to_vec())
    }
}

#[pymodule]
fn tako_hex_core(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_class::<HexBoard>()?;
    Ok(())
}
```

Build and install with Maturin:

```
# Install Maturin
pip install maturin

# Build and install tako-hex-core as a Python extension (dev mode)
cd tako-hex-core
maturin develop --release

# Verify import
python -c "import tako_hex_core; print(tako_hex_core.HexBoard(11))"
```

Phase 3 — Chess PyO3 Bindings (tako-chess-core)

The same pattern extends to chess. The Python python-chess library is replaced by a Rust chess backend. The tokenizer — which converts board state to the token sequence fed into HRM — is also implemented in Rust to eliminate Python overhead on the hot path.

```
// tako-chess-core/src/tokenizer.rs
// Matches the Python tokenizer's token space exactly
#[pyfunction]
fn fen_to_tokens(fen: &str) -> Vec<u32> {
    let board = Board::from_fen(fen);
    board.to_token_sequence()   // same token IDs as Python implementation
}
```

Phase 4 — Full Rust Workers (tako-worker)

At chess self-play scale, the self-play workers become standalone Rust binaries. The key architectural change is that the inner MCTS loop — which runs entirely in the worker — no

longer crosses any Python boundary. Neural network evaluation is the only Python/GPU call, made via batched IPC to a Python inference server.

```
# Python inference server (training/inference_server.py)
# Receives batched board tokens over a socket, returns policy + value
class InferenceServer:
    def __init__(self, model: HRM, port: int = 9000):
        self.model = model
        self.socket = zmq.Context().socket(zmq.REP)
        self.socket.bind(f"tcp://*:{port}")

    def serve(self):
        while True:
            # Receive batch of token sequences from Rust workers
            tokens = self.socket.recv_pyobj()          # [N, seq_len]
            policy, value = self.model.predict(tokens) # GPU inference
            self.socket.send_pyobj((policy, value))    # return to Rust
```

```
// tako-worker/src/inference_client.rs
// Rust side: batch leaf nodes → send to Python server → distribute results
pub struct InferenceClient { socket: zmq::Socket }

impl InferenceClient {
    pub fn evaluate_batch(&self, boards: &[Board]) -> Vec<(Policy, Value)> {
        let tokens = boards.iter().map(|b| b.to_tokens()).collect();
        self.socket.send(serialize(tokens));
        let (policies, values) = self.socket.recv();
        zip(policies, values).collect()
    }
}
```

Component	Implementation
Inner MCTS loop	Pure Rust — zero Python overhead per simulation
Game logic	tako-chess-core (bitboard move gen, ~200M pos/sec)
Neural net eval	Batched IPC to Python GPU server (zmq or shared memory)
Experience sending	Rust serializes (state, policy, value) → Python replay buffer
GIL contention	Zero — workers are independent OS processes
Parallelism	One Rust binary per CPU core; each manages its own MCTS tree

6. Key Hyperparameters

6.1 Model Architecture

Parameter	Value	Description
d_model (hidden size)	512	Dimension of zH and zL
n_layers	8	Transformer layers per module (L and H)
n_heads	8	Attention heads
d_ff	2048	Feed-forward inner dimension (4× d_model)
N (high-level cycles)	4	Cycles per segment
T (low-level steps)	4	L-module steps per cycle
Total NT per segment	16	Effective computational depth
Total parameters	~27M	Matches original HRM paper

6.2 ACT & Deep Supervision

Parameter	Value	Description
M_max	10	Maximum segments at training time
M_max (inference)	20	Can increase without retraining
ε	0.15	Probability of forcing long thinking
N_supervision	5	Segments per training example
ACT loss weight γ	0.1	Relative weight of Q-head loss

6.3 Self-Play & MCTS

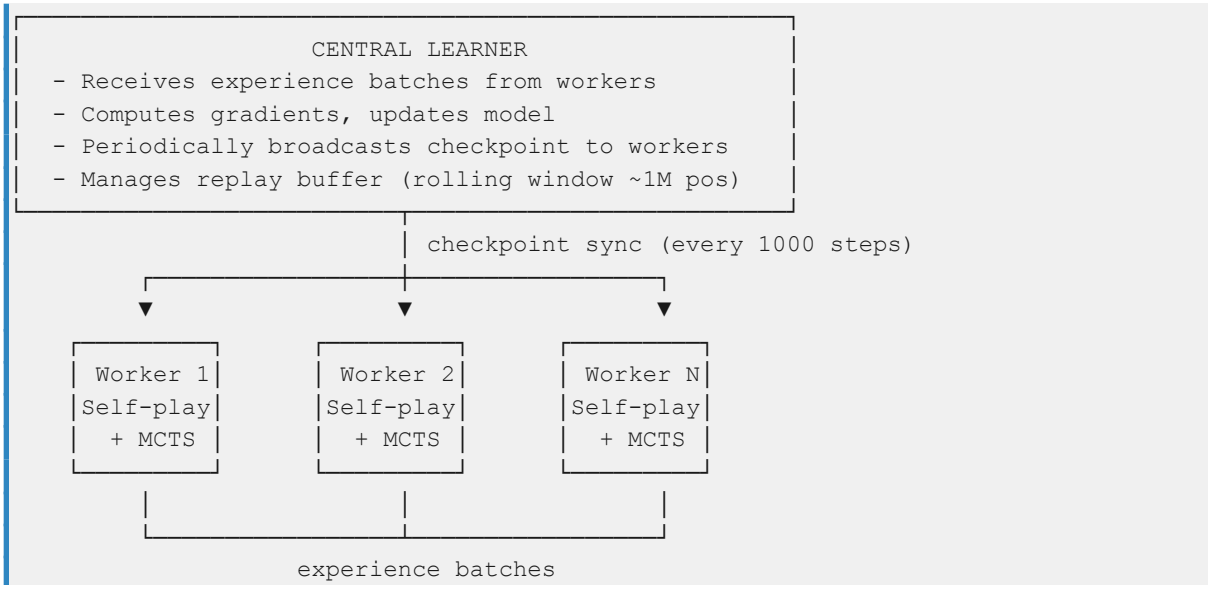
Parameter	Value	Description
MCTS simulations	400	Per move during self-play
MCTS simulations (eval)	800	Higher for tournament games
Dirichlet α	0.3 (chess) / 1.0 (Othello)	Root noise for exploration
Dirichlet ε	0.25	Noise mixing fraction
PUCT constant c	1.5	Exploration–exploitation balance

Temperature τ	1.0 for first 30 moves	Then 0 (greedy)
Replay buffer size	1M positions	Rolling window
Batch size	512	Training batch
Learning rate	2e-4 → 2e-5	Cosine decay
Optimizer	Adam (β1=0.9, β2=0.999)	
Mixed precision	bfloat16	Faster on A100/H100

7. Distributed Learning Pipeline

The distributed pipeline is architecturally separated into workers (data generation) and a central learner (gradient updates). Workers use slightly stale model checkpoints — a lag of 1000–5000 learner steps is acceptable and dramatically simplifies communication.

7.1 Architecture



Communication uses a simple async queue (Python multiprocessing or Ray). Workers push (state, policy, value_target) tuples; the learner samples from the replay buffer for training.

7.2 Opponent Pool (Preventing Policy Collapse)

Pure self-play against only the latest checkpoint causes training instability — the model can forget earlier strategies and cycle. Tako maintains a pool of past checkpoints as opponents:

- Every N training iterations, add the current checkpoint to the opponent pool
- Sample opponents weighted toward recent checkpoints (70%) but include older ones (30%)
- This prevents forgetting and ensures the policy is robust across diverse opponent styles
- Workers randomly sample from the pool for each game, rather than always playing self

7.3 Scaling

Setup	Detail
Minimum viable setup	1 learner GPU + 4 CPU workers (~5k games/day)
Recommended PoC setup	1 learner GPU + 16 CPU workers (~20k games/day)
Chess scale target	1 learner GPU + 64+ workers (~100k games/day)
Communication library	Ray (preferred) or Python multiprocessing
Checkpoint format	PyTorch .pt with full model state + optimizer state
Checkpoint interval	Every 1000 learner update steps

8. Development Roadmap

Phase	Timeline	Goals
Phase 0	Foundation (Weeks 1–2)	Core HRM implementation and unit tests. Verify forward pass, 1-step gradient, deep supervision loop. Pure Python Othello game environment. Cargo workspace scaffolded but no Rust code yet.
Phase 1	Othello PoC (Weeks 3–6)	Self-play pipeline on Othello (pure Python). HRM + MCTS. Evaluate vs Edax. Validate H/L module dimensionality hierarchy. No Rust yet — focus entirely on getting the HRM + self-play loop correct.
Phase 2	Hex + PyO3 (Weeks 7–10)	Introduce Rust: implement tako-hex-core crate with PyO3 bindings. Build HexBoard in Rust, expose to Python via Maturin. Replace Python Hex logic with Rust backend. Validate correctness parity. Confirm architecture generalizes from Othello to Hex.
Phase 3	Chess Bootstrap (Weeks 11–15)	tako-chess-core Rust crate: bitboard move generation + FEN tokenizer. PyO3 bindings wrap chess board for Python training loop. Supervised pretraining on ~1M GM PGN games. Target 1800 Elo starting point.
Phase 4	Chess Self-Play + Rust Workers (Weeks 16–24)	Promote from PyO3 bindings to full tako-worker Rust binary. Python inference server (zmq). Rust workers run MCTS, call GPU server for neural eval, push experience to Python learner. Distributed pipeline at 64+ workers. Opponent pool. Target 2200+ Elo.
Phase 5	GM Push (Weeks 25+)	Scale compute, tune hyperparameters. Profile and optimize Rust worker throughput. Target 2500+ Elo on Lichess blitz. UCI interface operational.

8.1 Success Criteria Per Phase

Phase	Criterion
Phase 0	HRM produces valid policy/value outputs. Unit tests pass. Gradient flows correctly through 1-step approximation. Cargo workspace compiles.
Phase 1	Beats Edax level 3 (intermediate). H-module participation ratio > 2× L-module (dimensionality hierarchy emerges). Self-play stable with no policy collapse.
Phase 2	tako-hex-core PyO3 bindings pass correctness parity tests vs Python reference. Maturin builds cleanly. HexGame wrapper is drop-in for BaseGame interface. Beats MoHex at beginner level.

Phase 3	tako-chess-core move generator passes perft tests (standard move count verification). Achieves 1700+ Elo on Lichess bot ladder from supervised pretraining alone.
Phase 4	tako-worker binary runs stable self-play at 64+ workers. GPU inference server maintains >80% utilization. Achieves 2200+ Elo. Games show novel tactical patterns not in training PGN.
Phase 5	Achieves 2500+ Elo on Lichess blitz. UCI interface operational for GUI play. Rust worker throughput profiled and bottlenecks documented.

9. Technology Stack

Library	Role	Usage
Python 3.11+	Primary language	Model code, training loops, evaluation, inference server
PyTorch 2.x	Deep learning framework	HRM implementation, autograd, mixed precision (bfloat16)
FlashAttention 2/3	Attention kernel	Memory-efficient attention for Transformer blocks
Ray	Distributed compute (Phase 1–3)	Python worker ↔ learner communication, actor model
Rust (Edition 2021)	Systems language	Game logic (Phases 2–3), self-play workers (Phase 4+)
PyO3	Python ↔ Rust FFI	Expose Rust game structs to Python; zero-copy numpy arrays
Maturin	Build tool	Compiles PyO3 crates into .so Python extension modules
zmq (zeromq)	IPC (Phase 4)	Rust workers ↔ Python GPU inference server communication
python-chess	Chess reference (Phase 0)	Used for correctness validation only; replaced by Rust in Phase 3
W&B / TensorBoard	Experiment tracking	Loss curves, Elo progression, module dimensionality analysis
pytest / cargo test	Testing	Python unit tests + Rust unit/integration tests

9.1 Future Considerations

- JAX migration: Better TPU support and XLA compilation for distributed training at scale. PyTorch stays for all PoC phases.
- Quantization: INT8 quantization of HRM for faster MCTS node evaluation at inference. Particularly relevant for GPU inference server throughput in Phase 4.
- Shared memory IPC: For Phase 4, zmq can be replaced with POSIX shared memory for lower-latency Rust worker ↔ GPU server communication if zmq round-trip becomes the bottleneck.

10. Key Design Decisions & Rationale

Why HRM instead of a standard transformer policy?

Standard transformers are static: a board position always maps to the same output regardless of time budget. HRM's recurrent structure means the same model can think "harder" on difficult positions by running more segments — without retraining. This is inference-time scaling for free.

Additionally, HRM's 27M parameter count with NT=16 effective depth achieves reasoning comparable to much larger static models. For low-compute goals, this is a significant advantage.

Why retain MCTS if HRM reasons internally?

HRM's internal reasoning and MCTS are complementary. HRM provides better per-node evaluation (richer than a static network), while MCTS provides structured look-ahead across multiple positions. The combination achieves stronger play than either alone.

In later phases, it is worth testing HRM policy-only (no MCTS) as a baseline to quantify the contribution of each component.

Why pretrain on human games before self-play?

Pure self-play from random initialization generates ~100% garbage games for thousands of iterations before the model learns basic chess rules. Supervised pretraining on GM games bootstraps the policy to ~1800 Elo, ensuring self-play generates meaningful signal from the start.

This is not a violation of the "primarily self-play" tenet — the vast majority of capability comes from self-play RL. Pretraining is initialization, not the final product.

Why Othello + Hex before chess?

Building and debugging a self-play pipeline on chess directly is expensive — feedback cycles are long, games take many more moves, and the state space makes debugging hard. Othello and Hex provide fast, cheap validation environments where we can confirm:

(1) HRM's dual modules develop the expected hierarchical representations
(2) The self-play pipeline is stable (no policy collapse)
(3) ACT allocates compute sensibly based on position complexity

A month of Othello experiments costs the same compute as a week of chess experiments.

Why introduce Rust at Hex rather than at chess?

Hex has simple, unambiguous game rules — no castling, no en passant, no promotion, no threefold repetition. This means the Rust implementation can be validated against a trivial Python reference with high confidence. Any bug is a Rust/PyO3 bug, not a game logic dispute.

By the time chess arrives, the team will already have experience with: PyO3 struct exposure, Maturin build pipelines, numpy zero-copy array passing, and GIL handling. Chess just adds more game logic — not more unknown unknowns about the Rust integration itself.

The promotion to full Rust workers at Phase 4 (not Phase 2) follows the same principle: validate the pattern cheaply (PyO3 bindings), then scale to the full architecture (IPC workers) only when the cost of Python overhead is actually proven to be the bottleneck.