Cambridge Battlecode tensor bot — postmortem

An RL agent for Battlecode 2026, trained March–April 2026. Stdlib-only inference, transformer actor and per-unit segmenting critic, behavior-cloned from human replays, fine-tuned with PPO/PMPO under a vectorized self-play league.

Preface

This is an AI-generated article but has been prompted and reviewed by me (Carson) for accurate data and details and expression of my own views and approaches.

Abstract

We trained a neural Battlecode agent end-to-end under a deployment constraint that the inference path use only the Python standard library. Over five weeks the project went from a small MLP self-play bot to a transformer actor with an attention-based per-unit value head, trained with PMPO on top of a centralized critic that uses HL-Gauss distributional value targets (Farebrother et al., 2024) in symlog space. Behavior cloning on the top-rated human replay corpus gave a usable cold start; encoder pretraining and an income-shaped HL-Gauss critic-pretrain stage unblocked PPO from learning the world model and the value function simultaneously. Direct influences: AlphaStar (multi-head categorical actor, league self-play, BC bootstrap), MAPPO/SMAC (centralized critic with decentralized actor), and SAM3 (per-token CLS queries that turn one shared backbone into per-agent outputs). This document is an honest account of what worked, what we abandoned, and the engineering details that mattered.

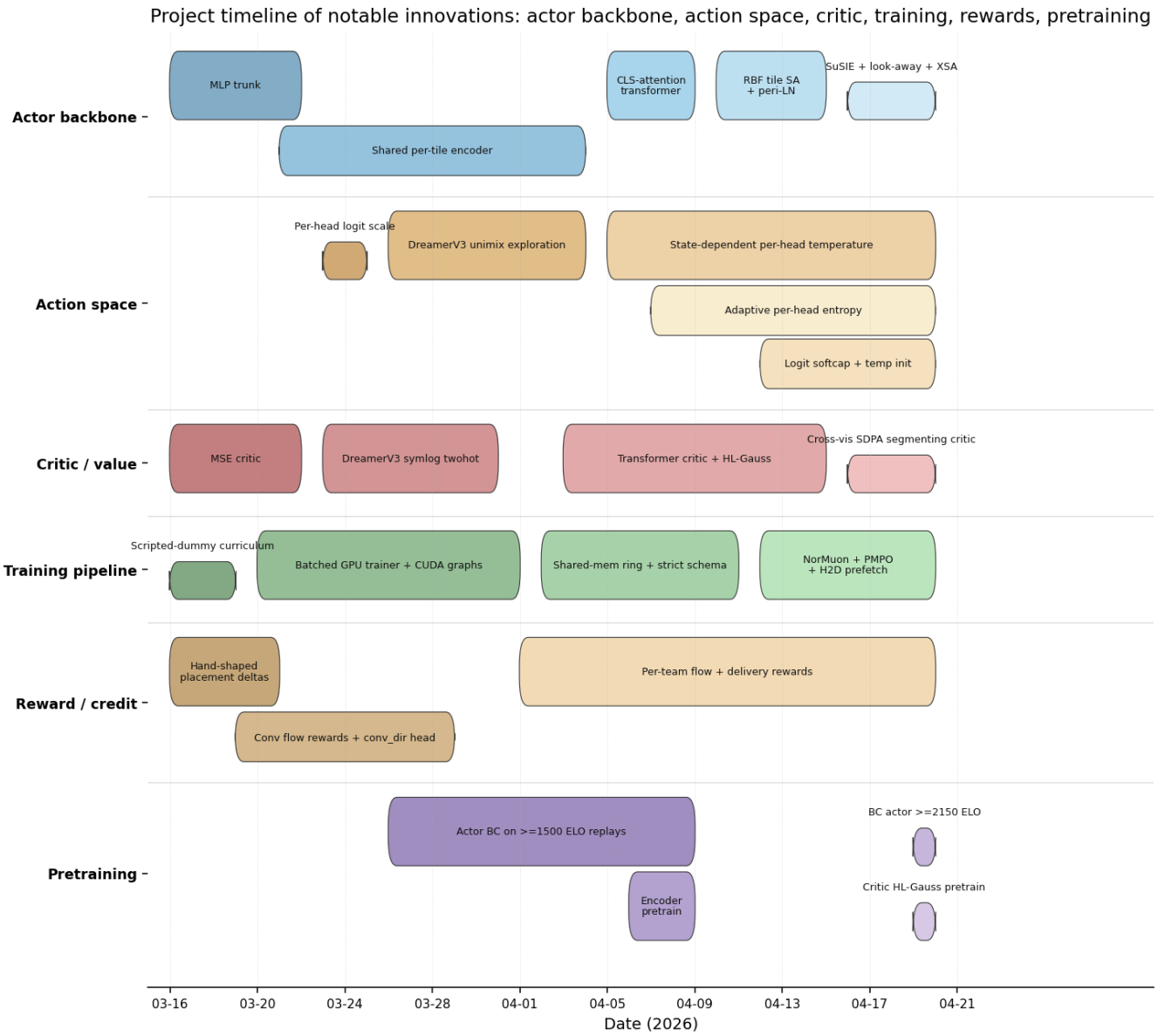

Figure 1. Project timeline. Bands are anchored to the actual git-commit dates in cb-bc; lanes show how each subsystem evolved in parallel.

1. Problem and constraints

Battlecode is a partially observable RTS where each team controls a fleet of heterogeneous units (harvesters, conveyors, gunners, sentinels, bridges, and so on) on a tile grid. Two project-wide constraints shaped every decision: **(a)** inference code under `bots/tensor_inference/` may not import torch or any third-party package, and **(b)** the runtime budget is roughly 2 ms of CPU per unit per round. Training is unconstrained and uses PyTorch, but the trained weights must re-implement as a stdlib forward pass with bit-for-bit parity.

The action space is multi-head categorical with five heads sampled in a causal hierarchy (`model.py:2499`): *build_dir* (9), *build_type* (13, including a *none* no-op), *conv_dir* (4 cardinal facings, legal only for conveyors/splitters/armoured), *bridge_target* (28 offsets in a radius-3 ring), and *move_dir* (9). Per-head temperatures are predicted by a learned linear head and squashed into [1.0, 3.0] via a log-space sigmoid (`model.py:94–115, 3134–3140`). Action masking runs from the observation alone (`main.py:803` in inference, `shared.py:432` and `model.py:415` in training) so masks agree across the deploy and training paths.

2. Stdlib-only inference: *tensor_lib*

`bots/tensor_inference/tensor_lib/` is a from-scratch forward-pass library in pure Python. The `Tensor` class (`tensor_lib/tensor.py:16–299`) wraps a flat `list[float]` and a shape tuple, leaning on C-backed primitives wherever the standard library exposes them: `math.sumprod` (aliased `_DOT`), `operator.mul` with `map`, and stride slicing for matmul column extraction. Linear, RMSNorm, LayerNorm, Sigmoid/Tanh/SiLU/ReLU, Softmax, Sequential, and a fused Linear+RMSNorm+activation kernel all live alongside it (`tensor_lib/nn.py:504`). The deployed actor is `ActorModel` (`tensor_lib/actor.py:260`): tile embedding, a tile self-attention block with SuSIE additive positional bias, a learned look-away register, exclusive self-attention (XSA), per-tile FFN with LeakyReLU², and cross-attention onto four CLS tokens. `_build_infer_cache` (`actor.py:427`) bakes the same tied weights the torch model produces; `_get_shared_model` (`main.py:210`) caches one model across all units keyed by file mtime. The bot disables cyclic GC (`main.py:11`) and reuses preallocated *memoryview* slabs for QKV, attention, and softmax buffers.

This is the project's most distinctive engineering decision. It narrowed the model space—no sparse ops, no attention kernels we could not reimplement—but let us deploy straight from a JSON weights file inside the 2 ms-per-unit budget.

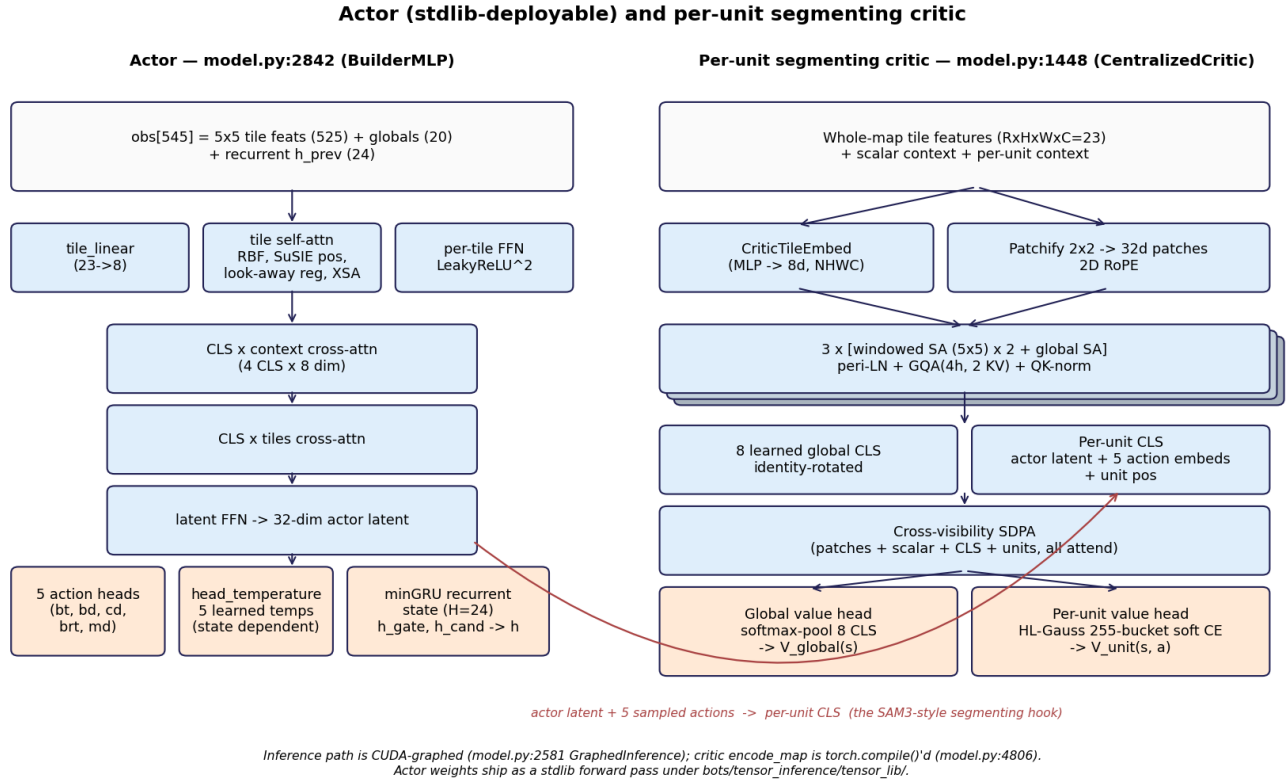

Figure 2. The deployable actor (left) feeds its 32-dim latent and the five sampled actions into the per-unit CLS tokens of the centralized critic (right). The critic is training-only; only actor weights ship.

3. Actor and segmenting critic

The actor (`BuilderMLP`, `model.py:2842`) is a small CLS-attention transformer over 25 tile tokens (5×5 visibility) plus a 20-dim global vector. Tile self-attention (`_tile_self_attention`, `model.py:2975`) scores tokens with an RBF kernel $-\gamma\|q-k\|^2$ with learnable γ, plus a SuSIE additive sinusoidal position bias (`model.py:364`) and a zero-initialized look-away register so the softmax has a calibrated distance sink. The RBF + SuSIE formulation comes from Artemis Cofer's `rbf_attention` reference (github.com/4rtemi5/rbf_attention); on the stdlib inference path it ran roughly **20% cheaper** than dot-product softmax attention with no measurable policy-quality regression in back-to-back ablations, which made it the right call for a budget measured in milliseconds. After SDPA, exclusive self-attention (XSA, `model.py:686`) strips the self-V-aligned component of each tile's output, so each tile absorbs only information orthogonal to what it already carries. Two CLS cross-attention stages collapse the grid into a 32-dim actor latent. A 24-dim minGRU-style *(1–z)·h+z·tanh(...)* recurrent state carries context across rounds (`model.py:3161`).

The critic (`CentralizedCritic`, `model.py:1448`) is a swin-style transformer over the entire map: tiles are MLP-embedded, packed into 2×2 patches, and processed by three blocks of *windowed-SA* × *2* + *global-SA* with grouped-query attention (4 heads, 2 KV groups), peri-LN normalization, and partial 2D RoPE. The SAM3 inspiration is the per-unit CLS builder (`CriticUnitCLSBuilder`, `model.py:1323`): each unit gets its own CLS token from *actor_latent* ⊕ per-head action embeddings (so the critic conditions on the sampled actions), and every per-unit CLS, eight learned global CLS tokens, the scalar context token, and the patch tokens go through one fully-cross-visible cuDNN SDPA call (`CriticGlobalSA`, `model.py:1084`). The action-slice of the CLS content projection is zero-initialized so the critic starts identical to a no-action baseline; the action channel only opens as gradient flows. One shared backbone thus produces a separate per-agent value, the same trick SAM3 uses to turn a shared image encoder into per-token outputs by giving each token its own query.

The value head is a 255-bucket HL-Gauss loss (Farebrother et al., 2024) with buckets uniformly spaced in symlog space over [–9, 9] (`model.py:153–167`). Targets are erf-CDF differences (`_ghl_target_probs`, `model.py:191`) forced to fp32 even under bf16 autocast to avoid bin-mass distortion. Versus MSE, the distributional head was a clear win on optimization stability and explained variance for both end-to-end PPO and the supervised critic-pretrain stage (Section 5).

4. Training pipeline and throughput

Training is a single GPU server multiplexing many concurrent games. `collect_games_batched` (`train_batched.py:615`) launches one OS subprocess per game with `tensor_proxy/main.py` on side A and either `tensor_proxy_opp/main.py` (a frozen league opponent) or `tensor_dummy/main.py` (a scripted no-op curriculum) on side B. Proxy bots talk to the GPU server through a single-producer/single-consumer ring buffer in `/dev/shm` (`shared_collector.py:24`) with tiny wakeup pipes—no Python pickling on the hot path. Maps and team assignments are randomized per game with 50% side-swap probability (`train_batched.py:316`), so the bot trains as both A and B; winners are normalized to the actor's perspective afterwards.

Inference is fully CUDA-graphed: `GraphedInference` (`model.py:2581`) captures forward + masking + Gumbel-max sampling + log-prob gather inside a single `cuda.graph` region. Opponent inference uses bucketed cached graphs at batch sizes (8, 16, 32, 64, 128, 256), keyed by (weight path, hidden, mtime). The critic *encode_map* is `torch.compile`'d (`model.py:4806`); critic minibatch transfers are double-buffered across two CUDA streams (`model.py:4099`) so batch *k*+1 staging copies overlap batch *k* compute. A `HiddenStateBank` (`hidden_state_bank.py`) keeps the recurrent state on GPU with row-key gather/update, avoiding H2D ping-pong.

All training ran on a single **RTX 5090**. Rollout throughput on the current transformer-actor + transformer-critic setup is **~900 samples/second** end-to-end (rollout samples divided by elapsed time, averaged over the most recent flagship runs: `sp_20260409_manual36` at 935 samples/s, `sp_20260417_115120` at 888 samples/s). PPO update config: **4 epochs** per episode for both actor and critic, actor minibatch **2048** with TBPTT unroll **K=32**, critic minibatch sized to give **8 batches/epoch** over the rollout, **16 parallel envs/episode** (4 opponents × 4 games). A typical ablation was a 300-episode run, completing in roughly an hour of wall clock; flagship runs that ramped past 1000 episodes (e.g., `sp_20260405_193453`) took most of a working day.

5. Pretraining: BC actor + HL-Gauss critic

Both pretrain stages live in `bots/tensor/torch/pretrain/`. Replays are parsed by a Rust PyO3 crate (`packages/replay_parser/`) that returns raw bytes consumable via `torch.frombuffer` with zero copies; `_decode_sequential_replay` (`dataset.py:189`) zero-pads the observation axis on older replays whose obs vector predates a global-features bump, so the encoder still ingests them. The `--min-rating` filter defaults to **1500 average team Elo**; the most recent BC and critic-pretrain runs raised it to **2150** (`bc_actor_20260420_135121`, `critic_pretrain_income_20260420_145944`) to concentrate on top-tier macro play.

Actor BC (`train_bc.py`) trains the same `BuilderMLP` with masked cross-entropy on each action head and TBPTT through `BPTT_K=32`. Critic pretrain (`train_critic.py`) freezes the BC actor, precomputes per-step actor latents (`train_critic.py:1206`), and fits the centralized critic to discounted returns. The reward mode defaults to *income* (per-step `resources_collected_delta × 2.5` with no terminal bonus); the `--reward-mode` override added in commit `8121501` lets us match the RL-time reward without re-derivation. Targets are HL-Gauss distributions; the loss is soft cross-entropy (`train_critic.py:801, 876`).

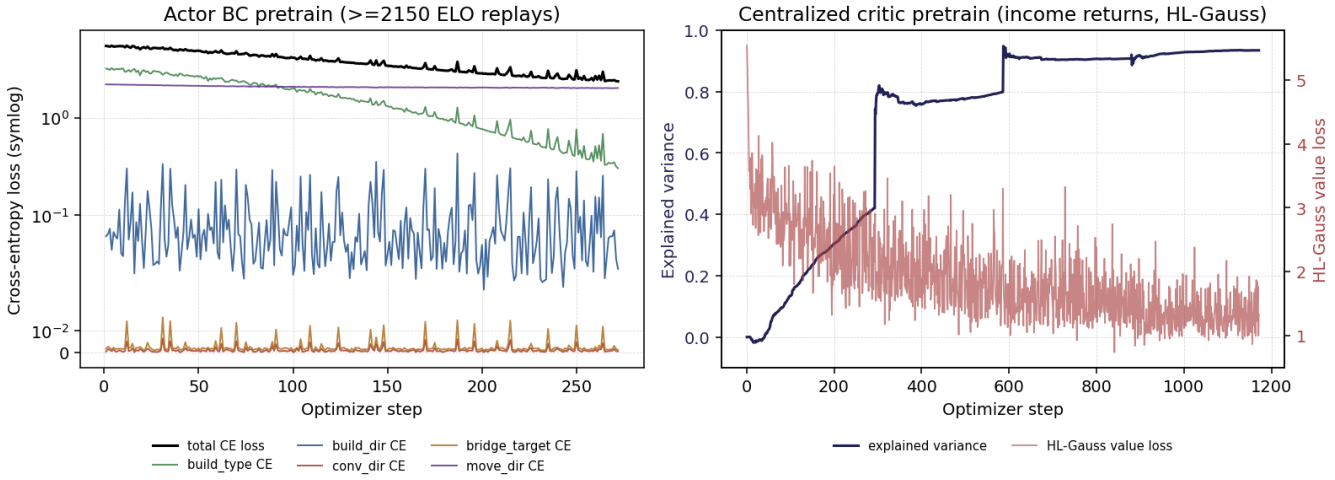

*Figure 4. Pretraining curves. Left: BC actor loss (symlog y) falls from 5.45 to 2.36 over 20 epochs, broken out into all five action-head cross-entropies. build_type drops fastest (categorical, low entropy); build_dir tracks build_type because they share the masking flow; conv_dir and bridge_target stay near zero because their loss is gated by build_type and the labels are very sparse; move_dir dominates the total loss and floors near the entropy of the human policy. Right: critic explained variance climbs to **0.93** on income returns within 1.2K steps; the step-up discontinuities mark epoch boundaries (4 epochs over the kessoku replay index).*

6. Rewards: a timeline

Reward shaping went through five distinct formulations between mid-March and early April. This section walks them in order, describing what each variant rewarded and the asymptotic resources/episode it produced (the same scalar tracked in Section 7).

(a) Hand-shaped placement deltas (03-16 to 03-21). `shared.py:497–611`. A successful harvester on a fresh ore tile pays `PLACE_HARV=5.0`; a successful conveyor pays `PLACE_CONV/(1+effective_dist)`. Failed placements fall back to a delta-style proximity reward keyed on per-ore `GameState.harv_proximity` / `conv_proximity`, paying only when a unit strictly improves on the previously-claimed best for that ore; a successful placement sets the proximity entry to −1 to lock out further stipends (single payment per ore). Carried the MLP era to basic harvester placement, but the asymptote stayed near ~1 resource/episode—there was no signal for what happens *after* placement (conveyor lanes, delivery).

(b) Outcome-only training (`sp_outcome_20260330`). Pure terminal win/loss reward, no shaping. The signal was too sparse for the 200-round horizon: resources/episode never left the floor, and the run sits as the lowest curve in Figure 5.

(c) Income shaping (`sp_income_20260330c`). Per-step `resources_collected_delta × 2.5` with no terminal bonus, broadcast per-round. Asymptote ~17 resources/episode—a clear step over outcome-only, but income alone could not credit the infrastructure that produced the collection (conveyor topology, delivery routes).

(d) BFS flow distance + KL-LR (`sp_20260401`). Per-conveyor distance-weighted credit via BFS over the placed graph, paired with the KL-adapted learning rate. Asymptote ~35 resources/episode, roughly 2× income-only—the bot now had a gradient pointing at the conveyor structure itself, not just the harvester delta.

(e) Per-team flow + delivery rewards (04-01 onward, final). `flow_reward.py`. The Rust parser exposes per-turn `transfer_events` and `resources_collected` scalars; `compute_dense_flow_rewards` (`flow_reward.py:83`) sums weighted conveyor hops and core-delivery deltas into a per-round team reward, broadcast identically to every active unit (`augment_trajectories_with_flow_rewards`, `flow_reward.py:116`). This is the formulation underneath every flagship run from 04-01 forward; it carried the RBF/peri-LN actor (`sp_20260409_manual36`) to the project's best asymptote at **~305 resources/episode** and is still the active reward for the cross-visibility SDPA critic run.

Two decisions held uniform across all five variants. **(i)** No reward normalization, only advantage normalization (`model.py:2429`); a running-mean reward normalizer was tried and reverted (commits `77aecce` →`123f851`) after interacting badly with PMPO sign-weighted updates. **(ii)** Per-unit GAE over the broadcast team reward, with the

per-unit critic baseline subtracting “what a typical unit at this state would contribute”—the MAPPO/SMAC compromise. There is no explicit reward decomposition (a known weakness when one or two units dominate the scoring delta), but the segmenting critic (Section 3) at least lets the baseline *differ* per unit.

7. Results

Our primary scalar metric is *mean cumulative harvested resources per episode*: a stable, comparable signal across opponents (winrate is not—it converges to ~50% in balanced self-play by construction) and the literal quantity the bot maximizes under the income-shaped reward. Figure 5 overlays nine flagship transformer-era runs, one per architectural regime, spanning the project's lifetime. The trend is consistent: each step raised asymptotic resources/episode by roughly an order of magnitude. The income-shaping run (`sp_income_20260330c`) tops out near 17; the flow-distance run (`sp_20260401`) jumps to ~35; the CLS-attention actor (`sp_20260405_193453`, 1001 episodes) ramps from ~1 to ~120; the RBF/peri-LN actor (`sp_20260409_manual36`) plateaus near **305 resources/episode**; the encoder-pretrained run (`sp_encoder_pt_20260407_004234`) reaches ~190; and the most recent cross-visibility SDPA critic (`sp_20260417_115120`) is still ramping. The RBF/peri-LN run is our cleanest stability datapoint: no policy collapse, KL-adapted learning rate stays in band (`train_batched.py:1703`, bounds [3e-5, 3e-3]), and resources/episode keeps climbing as the league hardens.

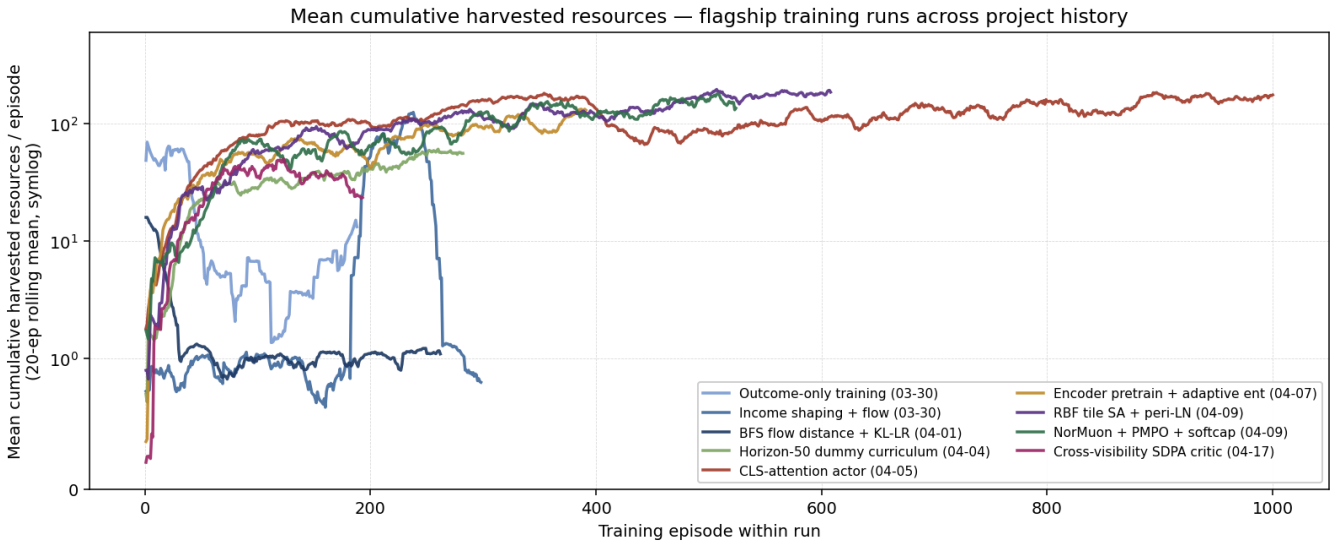

Figure 5. Mean cumulative harvested resources per episode across nine flagship training runs (20-episode rolling mean, symlog y-axis). Each run is from a distinct architectural regime; together they trace roughly two orders of magnitude of policy improvement from late March through April.

Why the curves plateau. The most informative pathology in late-run replays is shown in Figure 6: even after long training, agents fail to *specialize into complementary roles*. They all build toward the same ore deposit—almost always in a straight line vertically or horizontally from the core—and ignore equally accessible ore in other directions. This falls out of two design choices interacting. **(i)** The actor is small (a 32-dim CLS-attention transformer with five categorical heads, parameter count deliberately constrained by the 2 ms/unit deploy budget). Without enough capacity, near-identical local observations collapse to a near-deterministic policy: every unit that sees a similar 5×5 view picks the same direction. **(ii)** Credit is assigned at the team level—the per-team flow + delivery reward in `flow_reward.py` is broadcast identically to every active unit. The segmenting critic gives each unit a different baseline, but with one global reward signal there is no gradient pressure to *distinguish* “you should have gone east while your sibling went west.” Both factors push toward the easy attractor: a single straight conveyor lane to the nearest ore. Fixing this needs (a) a larger actor with diversified per-unit context (e.g., a unit-id embedding or a learned role token) and (b) per-unit reward decomposition—crediting each unit with the throughput it actually carries—rather than broadcasting the team total.

Figure 6. Three late-run replay frames. Left: a single straight north-south lane from the core toward the nearest ore cluster, ignoring equally accessible ore. Center: the same pattern—harvesters and conveyors stack into one collinear chain. Right: a similar convergence on the same axis-aligned conveyor topology. Across frames the policy fails to specialize per-unit: with one team-level reward and a small actor, similarly-conditioned units collapse to the same deterministic action.

On the optimization side, the sequence *NorMuon* → *PMPO* → *logit softcap + learned temperature* was decisive. NorMuon (Li et al., 2025) is Muon plus per-row second-moment normalization; our implementation (`normuon.py:85`) is a **fused** port—Newton–Schulz orthogonalization, second-moment scaling, and the shape-aware RMS rescale happen in one kernel-fused pass over the parameter tensor. Versus AdamW it won on three axes. The apples-to-apples harness (`scripts/compare_tile_sa_loss.py`) trains each tile-SA variant once with Adam and once with NorMuon under identical seeds; on that toy regression NorMuon converged to a lower MSE in fewer steps. In end-to-end PPO we also saw (i) a steadier actor gradient norm—the Newton–Schulz step orthogonalizes the update so spikes from a single high-variance row are damped—and (ii) lower peak GPU memory, since Muon stores only first-momentum plus a per-row second-moment scalar where AdamW carries both moments at full shape. PMPO (Zhao et al., 2025) replaces the PPO surrogate's advantage magnitude with a learned **separate positive/negative weighting** on the sign of the advantage (`model.py:2432`), letting positive- and negative-advantage updates be tuned independently. We adopted it after seeing the centralized critic occasionally emit magnitude-unstable advantages on rare units (the broadcast team reward pushes variance up when one unit happens to be near a delivery event). Logit softcap and learned per-head temperatures (Section 3) replaced an earlier DreamerV3-style unimix and gave the entropy schedule a cleaner handle: we now adapt the entropy *coefficient* per head (`model.py:3859`, commit `517889c`) rather than the temperature.

8. Future work

The two failure modes above—collapse to a single straight ore lane (Section 7) and a ceiling on resources/episode under broadcast team reward (Section 6)—both point at one lever: **more actor capacity and a richer signal to update it with**. Two concrete next steps.

(1) A bigger, temporally-aware actor. The current actor is deliberately tiny: one tile-SA block, two CLS cross-attentions, and a 32-dim latent. Stacking more tile-SA / context-SA layers (reusing the RBF + SuSIE structure that already runs cheaply on the stdlib path) would let the actor reason about more global structure—e.g., that an ore cluster east is worth more than a longer northbound lane—instead of collapsing every spawn to the same local greedy choice. The temporal axis is even more pressing: minGRU is a budget-driven compromise and the wrong primitive for long-horizon credit assignment in an RTS where round-30 decisions only pay out at round 200. An **autoregressive transformer over time**—a per-unit causal stack over the last K rounds with KV-cache reuse on inference—would replace the single 24-dim hidden with a real sequence model and condition each step's actions on the unit's actual build and movement history. The deploy cost is tractable: with KV caching the marginal per-round cost is one attention step per cached token, which fits the 2 ms budget for modest K.

(2) A world model for long-horizon credit assignment. Today's team-broadcast reward and per-unit GAE squeeze a thin signal: some unit on the team caused a delivery and you are credited because you were alive. A learned world model trained on the same high-ELO replay corpus would turn that thin signal into a much richer one. The template is

Dreamer 4 (Hafner et al., 2025): a transformer-based latent world model with per-token policy and value heads, trained jointly on observations, actions, and rewards, with imagination rollouts that propagate credit back tens to hundreds of steps without touching the real environment. That buys us three things: (a) per-unit counterfactual values ("what would the team's resources be at round 200 if *this* unit had built east instead of north?"), the per-unit decomposition our segmenting critic currently only approximates with a baseline; (b) much more signal extracted from the fixed replay corpus than BC + HL-Gauss critic-pretrain alone—the world model generalizes over states BC has never visited; and (c) a principled handle on the long-horizon delivery dynamics (lanes that take 100+ rounds to mature) that flow-reward shaping currently papers over.

9. Works cited

Carion, N., Gustafson, L., Hu, Y.-T., Debnath, S., Hu, R., Suris, D., et al. (2025). *SAM 3: Segment Anything with Concepts.* arXiv:2511.16719.

Cofer, A. (2024). *rbf_attention* [Source code]. GitHub. https://github.com/4rtemi5/rbf_attention.

Farebrother, J., Orbay, J., Vuong, Q., Taïga, A. A., Chebotar, Y., Xiao, T., et al. (2024). *Stop Regressing: Training Value Functions via Classification for Scalable Deep RL.* arXiv:2403.03950.

Hafner, D., Pasukonis, J., Ba, J., & Lillicrap, T. (2023). *Mastering Diverse Domains through World Models.* arXiv:2301.04104.

Hafner, D., Yan, W., & Lillicrap, T. (2025). *Training Agents Inside of Scalable World Models.* arXiv:2509.24527.

Li, Z., Liu, L., Liang, C., Chen, W., & Zhao, T. (2025). *NorMuon: Making Muon More Efficient and Scalable.* arXiv:2510.05491.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., et al. (2019). *Grandmaster level in StarCraft II using multi-agent reinforcement learning.* Nature, 575(7782), 350–354.

Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., & Wu, Y. (2022). *The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games.* NeurIPS. arXiv:2103.01955.

Zhao, C., Liu, Z., Wang, X., Lu, J., & Ruan, C. (2025). *PMPO: Probabilistic Metric Prompt Optimization for Small and Large Language Models.* arXiv:2505.16307.