ChatGPT

PPO Inference Scheduler Improvement Plan

Reward Design Fixes

The current hybrid reward function is too sparse and saturates due to the use of a tanh-like flattening, yielding almost no learning signal. We need to redesign it to provide **dense, differentiable feedback** and explicitly reward good load balancing. First, remove or replace the tanh squash so that reward scales linearly (or near-linearly) with performance instead of plateauing. This ensures small improvements in scheduling yield proportional reward changes, avoiding 99% zero-reward situations. Second, incorporate a **load balancing term** directly into the reward. For example, include a penalty based on the variance or standard deviation of loads across the 4 replicas – this will explicitly reward more even distribution of requests [1]. By combining throughput/latency metrics with a fairness term, the agent can distinguish good vs. bad actions. Finally, make the reward **dense** by giving feedback at each scheduling step. Instead of only end-of-episode rewards or threshold-based bonuses, provide incremental rewards for balancing the load and meeting performance targets on each decision.

Actionable changes:

- **Remove reward saturation:** Replace the `tanh` (or any hard cap) in the reward computation with a linear or gently clipping function. For instance, if reward was `tanh(x)` for some performance metric x, use a scaled linear reward (e.g., `reward = 0.001 * x` within a reasonable range) so that differences in x always produce differences in reward (no early flatline). This avoids flattening the reward signal and preserves gradient information.
- **Add a load-balance term:** Explicitly include a term that penalizes imbalance. For example, set `reward_balance = -α * stdev(num_requests_per_replica)` or use the difference between max and min load. Increase the weight on this term (e.g., **`load_balance_penalty` from 0.1 to 1.0**) so it significantly influences the reward [1]. This means the agent gets a higher reward when requests are evenly spread (lower standard deviation of load). If using a sum of terms, the reward could be: `reward = performance_term – α*(load_std)`, where α is tuned to ensure balancing is strongly incentivized.
- **Dense step-wise rewards:** Provide reward feedback at every dispatch decision. For instance, after each request assignment, compute a small reward: `r_t = - (stdev(current_loads))` or an increase in negative reward if one replica becomes overloaded. Additionally, give a positive reward for each request served (throughput) and perhaps a mild penalty for any violation of latency SLA, without thresholding to zero. This way, even suboptimal actions produce a gradient (not a zero reward). Over an episode, the agent will accumulate higher total reward for keeping the system balanced and efficient, rather than only getting a binary success/failure signal.
- **No early saturation:** Ensure no component of the reward function saturates within the normal operating range. For example, if using a latency term, avoid `tanh(latency)`; instead use a scaled linear penalty (e.g., `-β * latency` milliseconds) clipped only at extreme values if necessary. This preserves reward differences between "good" and "better" actions. In summary, the reward function should be a weighted **sum of metrics (throughput, latency, fairness)** that remain sensitive across the spectrum of possible values [2]. Calibrate the weights so that improvements in load balancing yield noticeable reward gains alongside throughput/latency improvements.

Exploration Strategy Improvements

The agent' s policy currently shows limited exploration – the action distribution is stuck (e.g. [22,37,29,40] allocations) and the temperature controller remains near 1.0. To avoid premature convergence to a suboptimal policy, we must encourage more exploration especially in early training. One fix is to **adapt the temperature parameter** dynamically to ensure sufficient entropy in the policy. For example, start with a higher temperature (e.g. 1.5 or 2.0) so the policy' s softmax output is more uniform (more random assignments), then gradually anneal it toward 1.0 as training progresses. This higher initial temperature will prevent the agent from locking onto one replica and instead encourage trying all replicas. We can automate this via a temperature scheduler that **maintains a target entropy level** – if the policy' s entropy drops too low (meaning it' s becoming too deterministic), increase temperature; if entropy is high, allow it to decrease [3]. This approach keeps the policy stochastic until it has found a truly better strategy.

In parallel, utilize **entropy regularization** in the PPO algorithm to encourage exploration. PPO' s loss has an entropy bonus term that, when weighted properly, prevents the policy from collapsing to a greedy deterministic choice [4]. By increasing the entropy coefficient (the factor multiplying the entropy bonus), we directly reward the policy for maintaining unpredictability. This helps the agent escape local optima and explore more of the action space. For instance, if the entropy coefficient (c_2) was 0.01, we might raise it to 0.05 early in training; the agent then gets a modest reward for keeping its action probabilities spread out. We can later decay this coefficient back down (e.g. from 0.05 to 0.01 after several epochs) once we are confident the agent has sufficiently explored and needs to exploit the best-found policy. Ensuring a healthy exploration phase will likely yield a more balanced final policy (closer to evenly distributed actions, rather than heavily favoring one replica).

Actionable changes:

- **Temperature schedule:** Initialize the softmax temperature at **1.5 (150%)** instead of 1.0. Allow the temperature controller to increase above 1.0. For example, set a schedule to **decay temperature from 1.5 to 1.0 linearly over the first N training iterations/episodes**. This ensures early actions are more random (promoting exploration), then gradually become greedier as learning solidifies. If a dynamic approach is preferred, implement a temperature scheduler that adjusts temperature to maintain policy entropy around a target value [5] [6]. For instance, target an entropy of ~`1.3` bits (for 4 actions, that' s about 70% of maximum entropy); if actual entropy falls below target, boost temperature by a small delta (and vice versa).
- **Increase entropy bonus:** Raise PPO' s entropy regularization weight **c_2** to encourage exploration. Concretely, if c_2 was, say, 0.005, try **c_2 = 0.02** (4× higher) at the start [7]. This makes the policy loss favor higher entropy (more random action selection), counteracting any early exploitation. Monitor the average entropy of the action distribution; it should stay higher initially. You can later **anneal c_2 back down** to the original value (e.g., over 100k steps) once the policy has learned to balance loads, to allow convergence.
- **Exploration noise strategies:** Consider injecting additional exploration if needed. For example, implement an **ε-greedy twist** during early training: with a small probability (ε, e.g. 5%), choose a random replica for a request instead of following the policy output. This can be phased out after some episodes. Another idea is to randomize initial policy network weights such that the initial action distribution is nearly uniform (so the agent doesn' t start with a bias for certain replicas).
- **Monitor and adapt:** Continuously monitor metrics like action distribution entropy and replica usage counts. If you notice the policy' s entropy still collapsing prematurely (temperature staying at 1.0 with low entropy), either **increase the target entropy/temperature** or further boost the entropy bonus. The key is to keep exploration high until the reward signals (from the improved

reward function) clearly differentiate a better load-balanced strategy. Only then should the agent be allowed (via gradually lowered temperature or entropy weight) to exploit what it has learned.

State Modeling Enhancements

The state representation (310-dimensional vector) should be examined to ensure it provides the agent with clear, useful information for load balancing decisions. In particular, since the goal is to balance load across 4 homogeneous replicas, the state should explicitly encode the **current load on each replica** and any other features that correlate with good balancing. If not already present, include features such as: the number of pending requests or current queue length for each replica, the recent throughput or utilization of each replica, and possibly the relative difference between replicas (e.g., the range or standard deviation of queue lengths). By adding these focused features, the agent can easily "see" when one server is more loaded than others and take corrective action. This aligns with best practices in RL for load balancing – key performance indicators (KPIs) used in the reward (like throughput or load) should also appear in the state, so the agent can observe the effect of its actions [8]. For example, if we use a load variance term in the reward, provide the per-replica loads in the state vector; the agent can then learn how its actions influence this variance over time.

Another consideration is the **scaling and relevance** of state features. With 310 dimensions, it's possible some features are irrelevant or noisy. We should identify if any part of the state can be simplified. For instance, if the state includes detailed per-request features or long histories that aren't directly improving decisions, we might compress or remove them. Focus on aggregate statistics that matter for scheduling (like average processing time, network delay, etc., if those vary). Normalize all features to comparable ranges (e.g., 0 to 1 or z-score) so that one large-magnitude feature doesn't drown out the others during neural network training. Homogeneous replicas mean that an ideal policy is symmetric with respect to the replicas, so the state could even be structured or ordered in a way that doesn't break that symmetry (though this can be complex, it's worth ensuring we aren't giving an arbitrary index an outsized importance). In practice, simply making sure each replica's features are arranged consistently (and perhaps the network architecture can share weights for processing each replica's sub-state) will help the policy generalize the concept of "any replica that's overloaded vs underloaded" rather than treating them differently.

Actionable changes:

- **Include per-replica load features:** Augment the state vector with a 4-length subvector representing each replica's current load (e.g., number of queued requests or CPU utilization of each server). These values should be updated each time step and normalized (for example, divide queue length by some max capacity or use utilization 0–1). This gives the policy immediate visibility into load imbalance. If the state already contains these indirectly, ensure they are clearly discernible. You might also include the running mean or last few values of these loads to indicate trends.
- **Feature engineering for balance:** Add a **summary statistic** of load distribution to the state. For example, compute the **load difference** (max minus min load among the 4 replicas) or the **coefficient of variation** of the loads and include that as a state feature. This is a one-number summary of imbalance the agent can easily use. While the raw loads let the neural network derive this, an explicit feature can speed up learning. It directly correlates with the reward's balancing objective.
- **Prune or compress high-dimensional parts:** If some of the 310 dimensions are, say, detailed telemetry or embeddings that the agent isn't using effectively, consider reducing them. For instance, if there are 50 features per replica about low-level hardware counters that don't impact scheduling at the request level, those could be removed or combined into an aggregate. The goal

is to remove distractors. Perform a feature importance analysis or simply experiment by removing certain blocks of the state vector to see if performance improves.

- **Normalize and scale:** Ensure all state inputs are normalized. For example, if one feature is an absolute timestamp or a raw request ID (which could be large integers), that should be removed or turned into a cyclical feature if needed. Key features like queue lengths or throughput should be scaled to a consistent range (0 to 1 or -1 to 1). This helps the neural network learn more effectively. If the state contains heterogeneous data (some binary flags, some continuous metrics), you may even want to separate them in the network (different input processing streams), but at minimum scale continuous features to a similar scale.
- **Leverage symmetry:** Since all replicas are identical in capacity, consider structuring the state and policy to treat them uniformly. For example, you could sort replica-specific features by load or randomize their order between episodes to prevent the policy from always focusing on “replica 1” due to position. A more involved idea is using a **shared subnet** that processes each replica’s feature vector and outputs a per-replica score, and then the policy decides where to send the next request based on those (essentially implementing a form of attention or permutation invariance). However, if that’s too complex to implement, simply be aware of symmetry – ensure that nothing in the state explicitly breaks the symmetry (like an ID that is always 1 for the first replica, unless needed). This way the agent learns general load balancing behavior applicable to any replica, rather than memorizing an index.

PPO Algorithm and Hyperparameter Tuning

In addition to reward and state changes, we should fine-tune the PPO training parameters to stabilize learning under the new setup. One key aspect is **loss function balancing** – PPO’s total loss includes the policy surrogate loss, the value function loss, and an entropy bonus (with coefficients often named c_1 for value and c_2 for entropy) [9] [7]. If the value loss was weighted too high (c_1 too large) in the original runs, the agent might have focused on predicting values (which were mostly zero due to sparse rewards) at the expense of improving the policy. Now that we’ll have denser rewards, we can adjust c_1 and c_2 to appropriate levels. For example, consider **decreasing the value loss weight c_1** (if it was 1.0, drop to 0.5) so that the policy gradient gets relatively more emphasis. This prevents the critic from dominating updates – important now because the value function will need to adapt to the new reward signals, but we don’t want it to learn too slowly and hold back the policy. Meanwhile, as noted, **keep the entropy bonus weight c_2 higher at first** (as recommended above for exploration) and potentially return it to a smaller value once the average entropy naturally stays sufficient [7].

Next, refine the **GAE (Generalized Advantage Estimation) parameters**. GAE uses a discount (γ) and a smoothing factor (λ) to compute advantages. The choice of λ trades off bias and variance in advantage estimates [10]. With the new reward providing more immediate feedback each step, we might not need a very high λ. A slightly lower λ (e.g. **reduce from 0.95 to 0.90**) will make advantage estimates rely a bit more on immediate rewards rather than long-horizon returns, which can reduce variance when rewards are dense. This can stabilize training in early phases. Conversely, if we find that some aspect of performance (like final tail latency or long-term throughput) is only seen at episode end, we should ensure γ is high (close to 0.99) and λ around 0.95 so that those rewards propagate backwards properly. In summary, we expect the reward tweaks to make the task more Markovian (each step’s reward tells the agent something useful), so leaning slightly toward lower λ can be beneficial. We will monitor training: if advantage estimates are noisy, we lower λ a bit; if the agent struggles to connect an action to a delayed outcome, we raise λ.

We should also examine other PPO hyperparameters: **learning rate, batch size, and clip ratio**. With a better reward signal, it may be safe to use a slightly larger learning rate to speed up convergence – for instance, if it was 3e-4, one could try 5e-4 – but do so cautiously and monitor for instability (spikes in value loss or oscillating reward). The **clip ratio** (ϵ in PPO, often 0.2) can usually remain at 0.2; however, if we observe that updates are too cautious (policy not improving much per epoch), we could experiment with 0.25. Generally 0.2 is a good default, but the main point is to **not constrain the policy too heavily now that better rewards are in place**. If a KL divergence cap was being used (some implementations stop updates early if KL > some threshold), consider relaxing it slightly, since initially the policy may need to move a bit farther from the old policy to start balancing loads correctly. We will also maintain **advantage normalization** each update (ensuring the advantages have mean 0, std ~1) – this is typically done and should remain, especially as reward scales change.

Actionable changes:

- **Adjust loss weights:** Set the PPO hyperparameters to give more balanced importance to each term. For example, **c_1 (value function loss coefficient)** down from 1.0 to **0.5**, and maintain **c_2 (entropy bonus coefficient)** at an elevated level (e.g. 0.02 as suggested earlier, then maybe down to 0.005 later) [7] . This means the policy gradient has relatively more say in the updates, and the entropy bonus keeps exploration going. The value function will still learn, but we accept a bit more bias in value estimates initially in favor of faster policy improvement.
- **GAE λ tuning:** If currently λ = 0.95 or 0.99, try **λ = 0.90** for a while (with γ still high, e.g. 0.99). A lower λ will shorten the advantage horizon, reducing variance from long-term rewards [11] . This can help given that rewards are now more frequent; the agent doesn' t need to consider extremely long futures for credit assignment. Monitor the results: if training becomes more stable (advantages less noisy, learning curves smoother), keep the lower λ. If we notice the agent ignoring longer-term effects (like perhaps it greedily balances load at each step but misses some end-of-episode goal), we can nudge λ back up to 0.95. The idea is to find a sweet spot where the advantages are informative but not too noisy.
- **Learning rate and epochs:** Increase the PPO actor/critic learning rate slightly to accelerate adaptation to the new rewards. For instance, if it was 3e-4, test **5e-4**. Because our reward is denser and less volatile now, the policy can handle a bit larger steps. Pair this with sufficient training epochs per iteration (e.g., 4–5 epochs over each batch of trajectories) to fully utilize the new feedback. We should also ensure the batch size (number of time steps collected before an update) is large enough to capture diverse scheduling scenarios – if it was small (say 2048 time steps), consider doubling it (4096) so that each update sees a variety of request patterns, which helps gradient estimates.
- **PPO clip/regularization:** Keep the **clip ratio at 0.2** initially. If policy updates seem very conservative (little change in policy across iterations) even after reward fix, you can try **ϵ = 0.25** to allow slightly larger updates. However, do this carefully – watch the KL divergence between old and new policies. The goal is to let the policy move to a better distribution (perhaps from the imbalanced [22,37,29,40] toward [32,32,32,32]) faster, now that we trust the reward signal. If using a KL penalty instead of hard clip, lower the penalty coefficient or increase the target KL to avoid premature stopping of updates.
- **Reward normalization:** Given the reward function has been changed (and possibly has larger magnitude now, especially with the linear terms), implement **reward scaling or normalization**. For example, use a running estimate of reward mean and standard deviation to normalize rewards (or advantages) to a reasonable range each update [10] . This prevents large reward values from causing too big policy updates. Many PPO implementations do advantage normalization by default; ensure that is on. This way, whether a step yields reward 0.5 or 5.0, the training sees a comparable advantage after normalization, keeping training stable.
- **Evaluation and adjustment:** Continuously evaluate the policy' s performance against the baseline (e.g., round-robin). As training progresses with these new settings, look at the distribution of requests per replica. If one replica still consistently gets significantly more, it indicates the policy might be stuck or reward weighting might still be off – for instance, then

increase `load_balance_penalty` further or add an even stronger negative reward for imbalance. Also check the **temperature controller output** during training: we expect initially >1.0, and gradually trending down. If it stays at 1.0 early on, that means exploration is still lacking – in that case, consider explicitly raising the minimum temperature or increasing the entropy target. Conversely, later in training, if temperature doesn’ t drop below 1.0, ensure that the mechanism to anneal it (or reduce entropy weight) is functioning so the policy can converge. Fine-tune these hyperparameters as needed, guided by training curves and evaluation metrics, to reach a well-balanced scheduling policy.

---

1 2 8 Reinforcement learning for communication load balancing: approaches and challenges
https://www.frontiersin.org/journals/computer-science/articles/10.3389/fcomp.2023.1156064/pdf

3 5 6 Enhancing Efficiency and Exploration in Reinforcement Learning for LLMs
https://arxiv.org/html/2505.18573v1

4 7 9 PPO — Intuitive guide to state-of-the-art Reinforcement Learning | by Brian Pulfer | Medium
https://medium.com/@brianpulfer/ppo-intuitive-guide-to-state-of-the-art-reinforcement-learning-410a41cb675b

10 11 Annonated Algorithm Visualization
https://opendilab.github.io/PPOxFamily/gae.html