ChatGPT

PPO Training Improvement Plan for Vidur Inference Scheduler

1. Reward Function Redesign

Motivation: The current hybrid reward (latency/throughput/imbalance) is too tightly clipped between –4 and +4, causing the agent's returns to **flatten around +0.79** and frequently hit a floor at –4.0. This indicates that beyond a certain performance, the reward signal provides no additional incentive for improvement. Latency penalties implemented via a steep logistic around a threshold may be saturating, so the agent isn't differentiating "bad" vs "terrible" latency scenarios, nor getting extra credit for exceptional latency reductions. This lack of gradient in both the high and low extremes likely contributed to the PPO policy converging below round-robin performance.

Impact on Learning: By **expanding the positive reward headroom** and reducing early saturation of penalties, the agent will receive more informative feedback for both improving latency and avoiding latency violations. A less clipped reward encourages exploration beyond the current plateau, as incremental latency improvements will translate into higher rewards (instead of capping out) and severe latency events will be distinguished (instead of all being lumped at –4). In scheduling tasks, using a well-shaped reward that emphasizes waiting time (latency) has been shown to drive better decisions [1]. Removing the artificial ceiling on good performance should incentivize the agent to outperform the baseline rather than satisfice at the threshold.

Implementation Guidance: Adjust the reward design in the environment configuration or code (e.g. the `vidur/env` reward computation module) to **widen the range and soften clipping**. For example:

- **Increase or remove reward clip limits:** If a `MAX_REWARD` of +4 and `MIN_REWARD` of –4 are defined, consider raising these (e.g. ±8) or eliminating hard clipping altogether, relying on normalization to keep values reasonable. This ensures truly superior latency (well below the SLO) can yield proportionally higher reward instead of topping out at +0.8.

- **Tune the logistic latency penalty:** Make the penalty curve shallower or centered further beyond the latency SLO so that latencies slightly above the threshold aren't immediately pegged to –4. For instance, if the logistic penalty currently achieves –4 at 110% of the latency target, stretch it to 130% or use a piecewise linear extension beyond the threshold. This **reduces penalty saturation**, allowing the agent to detect gradations of "bad" latency and learn to avoid the worst cases rather than giving up once over the limit.

- **Add reward for extra-low latency:** To **improve latency signal**, introduce a modest positive reward component for latency significantly under the target (e.g. negative of latency, or a bonus when P99 latency is below the SLO by a margin). This expands the positive range and encourages beating the threshold instead of just meeting it. Ensure this term is scaled so that achieving, say, 20% lower latency than round-robin yields a noticeably higher total reward (e.g. approaching +3 or +4) rather than staying near +0.8.

- **Rebalance throughput vs latency weights:** In the hybrid reward formula (latency, throughput, imbalance), increase the weight/scale on latency reduction and possibly cap the penalty for small throughput drops. This prioritizes latency-first behavior. For example, limit how much reward can be lost from a slight throughput dip (so long as throughput stays within acceptable range), while making each millisecond of latency saved count more. This change, combined with the expanded range, will make latency improvements the primary route to high rewards. It aligns with Vidur's goals since meeting strict latency SLOs is often a hard requirement [2].

Where to Modify: Update the JSON/YAML config for rewards (if exists) – e.g. fields like `latency_threshold`, `latency_penalty_slope`, or `reward_clipping` – to reflect the new scaling. In code, functions that compute the reward (for example, `calc_reward()` in the simulator or scheduler environment class) should be adjusted: use a smoother logistic or a conditional to avoid immediate saturation at threshold breach. Also adjust any normalization of reward (e.g. if value normalization assumes reward ∈ [–4,4], extend that range accordingly). By making these changes, the PPO agent will have a clearer, richer reward landscape to explore, pushing it toward **better-than-baseline latency** performance.

2. State Feature Improvements

Motivation: The current state observation may be lacking critical information about **queueing delays and load trends**, limiting the policy's foresight. Without features like per-replica queue wait time or recent load changes, the agent might only see instantaneous load or utilization, which doesn't capture how long requests have been waiting or if a particular replica is consistently lagging. This can lead to suboptimal scheduling (e.g. not prioritizing a replica that has an older request waiting). Additionally, raw metrics might not be normalized, making it harder for the neural network to generalize across different scales of workload.

Impact on Learning: Introducing features such as **per-replica queueing delay, temporal trends, and normalized loads** will enrich the agent's understanding of the system state, leading to more informed decisions. For example, if the state includes how long the oldest request in each replica's queue has been waiting, the agent can infer urgency and avoid sending new traffic to an overloaded replica. Indeed, prior RL schedulers incorporate such cues: one design fed the scheduler the queue length and the wait time of the first job in each queue, which helped the agent reduce latency by focusing on backlogged queues [3]. Temporal trend features (like the increase or decrease in queue length over the last few scheduling cycles) would allow the agent to anticipate surges or idleness, rather than reacting myopically. Normalizing inputs (e.g. dividing queue lengths or service rates by a peak value or capacity) ensures that features scale comparably (0 to 1 range), improving network training stability and generalization to different cluster sizes or loads.

Implementation Guidance: Enhance the observation space in the Vidur simulator environment (likely in the `get_state()` or `observation_builder` code):

- **Queueing delay:** Add a feature for each replica representing the **waiting time of the oldest request** (or average wait time) in that replica's queue. This could be computed as current time minus the arrival time of the first request in queue (0 if empty). Also include **queue length** (number of requests waiting) if not already present. These features directly signal how busy a replica is beyond just current processing load [3]. For implementation, if there's a data structure tracking queued requests per replica, retrieve its length and the oldest element's age and append these to the state vector for that replica.

- **Temporal load trend:** Include one or two features that capture recent history. For example, **exponential moving average of latency or load** and/or the **delta** from last timestep's value. A concrete approach is to maintain a running average of each replica's utilization or queue length and supply the difference between the current value and this average. Another approach is a **history vector** (e.g. the last N latencies or actions) fed into the RNN, but since the actor is already GRU-based (handling temporal sequence), providing the trend as part of state is simpler. Implement this by updating the state assembly code to compute these on the fly: e.g., store the last round's queue lengths in the env object, subtract from current to get a trend indicator per replica.

- **Normalized loads and times:** Apply normalization to raw features like number of tokens in service, queue lengths, or time values. For instance, if typical max queue length is 100 requests, divide the queue length feature by 100 (capping at 1.0). If latency is measured in milliseconds, divide by the SLO (latency threshold) so that the threshold corresponds to 1.0 in the feature space. Normalize per-replica loads by capacity (e.g., fraction of max throughput). This can be configured via a normalization layer or simply computed before feeding to the network (if a JSON config defines feature scaling, set those). Normalized features prevent large scale differences from overshadowing others and let the agent apply what it learns on one scale to another scenario.

- **State schema changes:** If the state is represented as a concatenation of per-replica features, the length will increase with these new entries. Ensure the actor-critic network input size is adjusted accordingly (e.g. if adding two new features per replica, input layer dimensions should reflect that). In a config file (like `state_features.json` or similar if present), list the new features (e.g. `"queue_wait_ms"`, `"queue_length"`, `"load_trend"`) so they are logged and exported to TensorBoard/CSV as well. This documentation helps verify during training that these features are non-zero and evolving as expected.

By improving the state representation with richer features, the agent can learn a **more nuanced scheduling policy**. It will understand not just how many requests are at each replica, but how desperate each queue is (long waits) and where the system is heading (increasing or decreasing load). These additions should directly contribute to latency reduction (e.g. avoiding sending work to a replica with long queueing delay) and better **generalization** to different traffic patterns, since normalized trend-aware features let the policy adapt to workloads of varying intensity.

3. Network Architecture Updates

Motivation: The existing actor (3-layer GRU, 320 hidden) and critic (4-layer GRU, 384 hidden) architectures may limit expressiveness in representing per-replica differences and their interactions. A pure recurrent model might be treating the multi-replica state as a flat sequence or relying on the GRU to learn implicit relationships, which is challenging when scheduling decisions require understanding **comparisons across replicas (e.g. which one is least loaded)**. Additionally, the critic network is heavier than the actor and could be prone to overfitting or slowing down training. There is an opportunity to introduce **cross-replica attention mechanisms** so the policy can attend to the most critical replica states, and to streamline the value function with a simpler head that leverages shared features from the actor. Modern RL schedulers have begun using attention layers to capture system-wide patterns, as these mechanisms help the model focus on critical signals and relationships [4].

Impact on Learning: Upgrading the architecture will **improve the policy's decision quality and generalization**. A cross-replica attention module allows the agent to explicitly consider interactions (for example, it can learn an attention weight to heavily focus on a replica with a huge queue delay versus one that's idle, effectively implementing a "least outstanding requests" strategy when appropriate). This should lead to more intelligent scheduling than a sequence-processing GRU alone. Enhancing per-replica expressiveness (e.g. via a small feed-forward network per replica) means the model can capture non-linear features combinations (like a replica with moderate load but growing queue might be more critical than one with high load but short queue) before combining information. Meanwhile, a **lightweight critic head** (fewer layers or shared backbone) will reduce the complexity of value estimation, potentially yielding more stable advantage estimates and faster convergence. Sharing representations between actor and critic can also regularize the value function – for instance, using a common encoder for both networks has been shown to improve sample efficiency [5].

Implementation Guidance: Modify the model definition (likely in the training pipeline's model configuration or in a file such as `vidur/rl/model.py` if it exists) as follows:

- **Add cross-replica attention:** Introduce a self-attention layer that operates on the set of per-replica embeddings at each time step. Concretely, after the state features for all replicas are collected for the current scheduling decision, pass them through an attention module (e.g. a Transformer encoder block with a single layer and a few heads). This will produce a context-aware embedding for each replica or a pooled embedding for the whole system. For example, use PyTorch's `nn.MultiheadAttention` to allow the model to weigh information from different replicas when computing the next action. The **key outcome** is that the policy can, in one forward pass, compare all replicas' states and pinpoint which one should be scheduled (the action typically being an index of a replica to serve next). Academic work on HPC job scheduling found that combining multi-head attention with gating improved the scheduler's ability to retain essential signals and capture long-range dependencies [4], validating this approach. In practice, you might wrap this in a small module (e.g. `CrossReplicaAttention` class) and insert it before the GRU or between GRU layers. Ensure the dimensions match (e.g. if GRU input was of size equal to feature count × replicas, now each replica's feature vector goes into attention which outputs same dimension vectors, then aggregate or flatten for GRU).

- **Improve per-replica expressiveness:** Before feeding data into the attention or recurrent layer, use a **feed-forward sub-network per replica**. This could be a simple two-layer MLP (with ReLU) that processes the feature vector of each replica independently, projecting it into a higher-dimensional embedding (perhaps 64 or 128). This allows the model to compute a richer representation of each replica's state (capturing interactions among that replica's features like load, queue, etc.) locally before the global attention considers interactions between replicas. In implementation, this could mean adding a `nn.Linear(input_dim, embed_dim)` + activation for each replica's features (you can do this efficiently by treating the batch of replicas as a matrix and using one Linear on all). The GRU (or attention) would then take these embeddings as input instead of raw features.

- **Lightweight critic head:** Simplify the critic network by sharing the early layers with the actor and reducing the depth of the critic-specific layers. One approach is to use a **shared encoder** (e.g. the entire GRU+attention stack) and have two output heads: one for the policy (actor) and one for value. The heads can be as small as a single linear layer (for value, mapping the shared representation to a scalar). This way, the critic doesn't have its own 4-layer GRU, but rather reuses the representation learned by the actor, which both reduces computation and acts as a regularizer [5]. If completely sharing the GRU is not desired, another option is to cut down the critic to, say, 1 GRU layer (or a simple MLP) that takes the output of the actor's last layer as input. In config terms, you might set `critic_hidden_size: 256` and `critic_layers: 1` (down from 384/4), and toggle a flag for shared weights if available (e.g. `"share_encoder": true`). Also continue using value normalization (keep that module on) since it has been aiding stability, but verify it still works with the new reward scale.

- **Test architecture with small experiments:** After implementing, run a few short simulations to ensure that the attention mechanism is working (e.g. log the attention weights or verify that the model can still overfit a simple scenario). Check that adding attention doesn't break the RNN's hidden state logic (you may need to handle how the GRU hidden state interacts with the attention output each step – one approach is to do attention at each timestep independently on the current state, then feed a pooled result into the GRU along with the previous hidden state). Given the additional complexity, monitor training for any instability (if divergence occurs, you might need to reduce learning rate or initialization scale for the new layers).

By implementing these architecture updates, the agent's policy network becomes **more powerful in describing the scheduling problem**. It will explicitly learn to **"attend" to the most loaded or lagging replica**, akin to how a least-outstanding-requests (LOR) heuristic would, but in a learned optimal way. This, combined with a simpler critic, should improve convergence and help the policy surpass round-robin by making more globally optimal scheduling decisions.

4. PPO Training Strategy Upgrades

Motivation: While the pipeline (behavior cloning + PPO) is generally sound, there are several training strategy enhancements that could address the observed convergence issues. The current PPO config (clip 0.15, entropy_coef 0.02, γ=0.95, λ=0.9, etc.) might be leaving performance on the table in terms of exploration and stability. For instance, the entropy coefficient is fixed – the entropy was ~1.35 during training, indicating the policy may still be fairly stochastic. As training progresses, we might want to **decrease entropy** to allow the policy to converge to a deterministic, low-latency strategy. Also, the PPO update could be made more adaptive: using a **KL divergence target** would prevent the policy from drifting too slowly or too quickly. The fact that KL stayed <0.06 suggests PPO might have been overly cautious (perhaps not exploring far from the behavior clone policy). Furthermore, introducing a **curriculum** (progressively increasing difficulty or objectives) can prevent early training from getting "stuck" in a local optimum. These techniques are known to improve learning effectiveness – for example, curriculum learning in scheduling tasks has led to significantly better final results (agents trained with curricula achieved ~3% shorter job completion times than without) [6].

Impact on Learning: Upgrading the training strategy will likely yield a **more robust and better-performing policy**. Adaptive entropy scheduling ensures the agent explores sufficiently early on (preventing premature convergence) and exploits more later (fine-tuning for latency optimization). A curriculum approach could, for example, start the agent on an easier scenario or partial objective and then ramp up, which helps avoid the agent being overwhelmed by complexity initially. This typically results in higher rewards and better generalization [6]. Tuning GAE (λ) and discount (γ) can reduce variance in advantage estimates – a slightly higher λ (closer to 1) can improve long-horizon credit assignment, which is useful if latency penalties occur with some delay. KL stabilization (either via an explicit penalty or an early stopping criterion) will keep policy updates in a safe range: it prevents both over-updating (which can collapse the policy) and under-updating (which wastes samples). Overall, these modifications will make PPO training **more stable and sample-efficient**, helping the agent reach a performance above the heuristic baselines.

Implementation Guidance: Update the training script `train_ppo_warmstart_optimized.sh` and associated config (likely a JSON of hyperparameters) to incorporate the following:

- **Adaptive Entropy Coefficient:** Implement a schedule or adaptation mechanism for the entropy bonus (`entropy_coef`). For example, start with the current 0.02 for the first N training iterations to encourage exploration, then gradually reduce it (e.g. linearly or exponentially decay to 0.0 by the end of training). This can be done by updating the optimizer loop to multiply the entropy coefficient by a factor each epoch or by using a schedule in the config (if supported, e.g. `"entropy_schedule": [[0, 0.02], [40000, 0.0]]` meaning at step 0 it's 0.02, at step 40000 it's 0). An alternative is to use a **target entropy** approach (as in Soft Actor-Critic): monitor the policy entropy and if it remains high in later stages, manually decrease the coefficient more to push the policy to become more deterministic. The aim is to allow convergence to a consistent strategy once enough exploration has occurred. This **refines the exploration-exploitation balance**, preventing the agent from dithering with random actions when it could be reliably hitting latency targets.

- **Curriculum Learning:** Design a curriculum either in terms of environment difficulty or reward shaping over time. One idea is to begin PPO fine-tuning on a lighter load or a subset of scenarios. For instance, for the first 10,000 requests, run the simulator at a lower QPS or with fewer concurrent requests (making it easier to achieve good latency), then gradually increase to the full workload by 40,000 requests. This way the agent first masters basic scheduling, then faces busier conditions. Another curriculum dimension could be the reward: start with a reward function that is a bit more forgiving (e.g. latency penalty threshold slightly higher or clipped at –2 instead of –4) so the agent learns to improve latency without getting too many -4 signals early on, and then tighten the penalties after it has learned a reasonable policy. Implementation-wise, you could script this by updating env parameters partway through training – e.g., if using JSON configs, prepare multiple configurations (`env_config_easy.json`, `env_config_hard.json`) and have the training script switch or interpolate between them after certain epochs. Monitor performance on the fly; you should see that as difficulty increases, the agent's performance dips then recovers, indicating it's learning to handle the tougher scenario. Research has shown that such curricula yield better final policies in scheduling domains [6], as the agent avoids bad local optima that it might fall into if it started directly on the hardest task.

- **Generalized Advantage Estimator (GAE) Tuning:** Consider increasing λ from 0.90 to 0.95 (closer to 1.0) to put more weight on longer-horizon advantages. In a scheduling environment, some rewards (like avoiding latency violations) might only come after several steps of good decisions, so a higher λ can credit a sequence of actions leading to success. However, keep γ at 0.95 or at most 0.99; since requests have a limited lifetime in the simulator, extremely long horizons aren't necessary. Test the effect: a higher λ might increase variance, so if training becomes unstable, you can dial it back or use value function clipping. In the config JSON, this is simply `"gae_lambda": 0.95`. The **impact** is typically smoother learning curves – advantages will be a bit more accurate, helping the policy update in the right direction.

- **KL Divergence Stabilization:** Activate an **adaptive KL penalty or early stopping** for PPO updates. PPO's clipped objective indirectly controls KL, but we can explicitly ensure KL stays in a desired range (e.g. target KL = 0.1 per epoch). One method is to add a KL term to the loss: if the library supports it, set `"use_kl_regularization": true` and `"target_kl": 0.1`. This will penalize the loss when KL > target, effectively slowing down updates that change the policy too much [7]. Alternatively, implement manual monitoring: after each PPO epoch, check the KL divergence (already being logged ~0.06). If KL is below, say, 0.05 for several epochs, you might decrease the clipping ratio or increase the learning rate a bit to encourage faster learning; if KL spikes above

0.15, you can early-stop further mini-batch updates in that epoch or reduce the learning rate. This heuristic keeps the policy update size "just right" [8] – not too small (which would converge slowly or not at all) and not too large (which could collapse the policy). In practice, enabling KL regularization in PPO algorithms helps maintain training stability and prevents overshooting the optimum.

- **Gradual Behavior Cloning fade-out:** Since you warm-start with behavior cloning (BC), consider if a form of **mixed training** could help initially – e.g., continue to include a small supervised loss on the demonstration data for the first few PPO iterations to keep the policy from drifting wildly. This is a minor point (not explicitly asked), but sometimes used in RL finetuning. If interested, implement by mixing an LfD (learning from demonstration) loss with the PPO loss early on, weighted by a factor that decays to 0 after a certain number of updates.

All these strategy changes should be reflected in the training script or configuration. Document them in the experiment log (perhaps note in the CSV output when curriculum switches happen, etc.). The expected outcome is that PPO will train more **effectively** – you should observe higher peak rewards (beyond +0.79, if reward redesign is done) and the policy maintaining low entropy and low latency by the end of training. The policy should also be less likely to regress below round-robin, thanks to the stronger guidance from adaptive KL and curricula.

5. Rollout and Evaluation Enhancements

Motivation: Even with improvements in model and training, how we collect experience and evaluate the policy can significantly affect performance. The current setup might be using a single simulator instance for rollouts, which can lead to highly correlated samples and slow data throughput. **Parallelizing rollouts** will speed up training and provide a more diverse set of experiences per iteration. Additionally, the evaluation metrics should align with the latency-first objective: currently average reward was monitored, but we need to explicitly track **tail latency metrics (P90, P99)** and other safety-critical indicators to ensure the agent truly outperforms round-robin in the way we care about (latency SLO adherence). **Safety monitoring** during training is also prudent – we want to catch any policies that might exploit the simulator in undesirable ways (for example, by starving certain requests to game the average latency) or any episodes where latency spikes to unacceptably high values. These enhancements will give a clearer picture of the policy's performance and guard against regressions or unsafe scheduling behaviors.

Impact on Learning and Performance: Using parallel environment instances will **accelerate training and improve policy robustness**, as it's known to increase sample efficiency by decorrelating observations [9]. The PPO agent will learn from multiple independent rollout trajectories at once, which often leads to more generalized strategies (not overfitting to a particular sequence of random events from a single environment). Comprehensive evaluation metrics (especially tracking tail latencies) will ensure that improvements in the average latency don't come at the expense of occasional very bad outliers. In practice, a scheduler must keep 99th-percentile latency under a threshold [2]; by monitoring these, we can verify the policy meets production-level criteria. Safety monitoring will impact learning by avoiding or penalizing catastrophic decisions. If the agent ever produces a scheduling pattern that leads to, say, unbounded queue growth or extreme delays (beyond a defined safety limit), intervening can prevent those outliers from skewing training (and obviously would be important in a real deployment). Overall, these changes make the training/evaluation loop **more aligned with real-world performance goals**.

Implementation Guidance:

- **Parallel Rollouts:** Modify the training pipeline to use N simulator instances in parallel (e.g. 4 or 8) for experience collection. If the codebase is based on an existing RL library, use their VecEnv or parallel Sampler utilities. For example, with Stable Baselines, you would do `vec_env = make_vec_env(VidurEnv, n_envs=8, env_kwargs=...)` and then pass `vec_env` to PPO. If using a custom training script, you might spawn multiple processes using Python' s `multiprocessing` or `ray` to run `vidur.simulate()` concurrently. Each parallel worker should collect `rollout_len` transitions, and then these can be concatenated into one batch for PPO update. Ensure that the **minibatch size and epoch settings** account for the increased batch size (e.g. if 8 envs × 128 steps = 1024 transitions per update, you might increase minibatch size or number of SGD epochs slightly). Empirically, parallel environments will **increase the throughput** of steps (so you can finish 40,000 requests of experience faster) and produce more i.i.d. samples [9], which tends to stabilize training. This is because each batch will contain a mix of different trajectories/events, reducing the risk that a fluke event in one environment steers the policy update too strongly. From a code perspective, you may need to adjust how the environment is reset and stepped. For instance, in `train_ppo_warmstart_optimized.sh`, instead of a single loop calling `env.step()`, you' d maintain a vector of env states and step all simultaneously (or in chunks). The TensorBoard logging should also aggregate across these.

- **Tail-Tracking Metrics:** Augment the evaluation and logging to include **tail latency statistics**, such as 90th and 99th percentile latency per episode (or per fixed number of requests). The Vidur simulator already supports detailed metrics; in fact, Vidur' s own analysis uses TTFT-P90 and TBT-P99 as key metrics [2]. You can leverage these by computing them from the reward components or directly from the environment' s metrics. Implementation steps: if the environment returns a list of request completion times or a histogram of latencies at the end of an episode, compute the P90/P99 and log them (e.g. using TensorBoard' s scalar log). If not readily available, modify the environment to record each request' s latency. For example, every time a request finishes, log its end-to-end latency to an array; when an episode ends (or every 1000 steps), calculate the percentiles and reset the array. Add these values to the CSV export as new columns "latency_p90" and "latency_p99". During training, monitor these – the aim is to see these tail metrics improve and stay below round-robin' s levels. In the final evaluation of the trained policy versus round-robin, emphasize not just average latency but also tail latency reduction. By explicitly tracking this, the training process might even be adjusted if needed (for example, you could introduce an additional reward shaping that gives a small bonus for keeping P99 below threshold, but this must be done carefully to avoid non-stationarity in reward).

- **Throughput and Imbalance Metrics:** Alongside latency, ensure throughput is still logged (since we want competitive throughput). Track requests completed per second (or tokens/sec) and maybe average batch size. These should remain close to baseline values. Also monitor any **imbalance/fairness metric** (if defined, e.g. standard deviation of utilization across replicas) to ensure the policy isn' t overloading one server exclusively. Logging these will help confirm that our latency-first policy isn' t doing something pathological like sacrificing all throughput.

- **Safety Monitoring and Constraints:** Define safety limits and enforce them during training. For example, set a hard latency cutoff (maybe 2× the latency SLO) above which we consider the episode failed. The Medium article on Vidur-Search mentions constraining P99 scheduling delay under 5s for stability [10] – we can adopt similar logic. Implementation: if at any point P99 latency > X (or any single request latency > X), you could terminate the episode and give a large negative reward to strongly discourage that behavior. Alternatively, simply log a warning and count it. Another safety aspect is to ensure the agent doesn' t drop or ignore requests; if the simulation allows not scheduling certain requests, that's not acceptable (all requests must be eventually served), so the environment should already handle this, but worth verifying. During training, set up an automated check: for each training iteration, if any safety trigger fired (e.g. extreme latency, or maybe an internal metric like GPU memory overflow if that's modeled), print a message or store it. If such events become frequent, you might tweak the reward or policy constraints.

- **Evaluation Scenarios:** Beyond the training environment, test the trained PPO policy on various scenarios: different workload traces, different number of replicas, etc., to see how well it generalizes. This isn't a direct training change, but it's an important part of the plan to ensure the policy's superiority isn't narrow. For instance, run the Vidur simulator with the learned policy on a heavy workload and on a light workload, compare latency vs throughput trade-offs to round-robin and LOR. The expectation is that latency will be significantly improved at similar throughput. If it fails in some scenario, that might inform further training (perhaps adding that scenario into a curriculum or retraining with more diverse data).

Implementing these rollout and evaluation enhancements will make the training loop more **efficient and aligned with real goals**. With parallel rollouts, you should notice faster learning (more timesteps per second of wall-clock, and likely a more stable training curve). With tail metrics and safety checks, you will gain confidence that the new policy truly achieves **superior latency** without hidden trade-offs. All together, this comprehensive plan upgrades the PPO training pipeline to produce a Vidur scheduler that outperforms round-robin in latency (while keeping throughput high) and does so consistently, even under the varied conditions captured by the simulator.

Sources:

1. Kwon et al., "Improving End-To-End Latency Fairness Using an RL-Based Network Scheduler," Applied Sciences, 2023 – describes state and reward design for scheduling (queue lengths, delays) [3].

2. Huang et al., "Self-Attention Mechanisms in HPC Job Scheduling," 2023 – demonstrates using multi-head attention and shared actor-critic representations to improve scheduling decisions [4] [5].

3. OpenAI/Intel Coach PPO Documentation – notes on adaptive KL penalty in PPO for stable policy updates [8] [7].

4. Waubert de Puiseau et al., "Curriculum Learning in Job Shop Scheduling," CPSL 2023 – showed that curriculum training led to agents achieving ~3.2% shorter completion times than without curriculum [6].

5. Reddit RL discussion – explains that parallel environments greatly increase sample diversity and training speed by making transitions more IID [9].

6. Vidur paper (MLSys'24) and Medium article – emphasize importance of tail latency (P90/P99) SLOs in evaluating schedulers [2] and suggest constraining extreme latency outliers for stability [10].

---

[1] [4] [5] Self-Attention Mechanisms in HPC Job Scheduling: A Novel Framework Combining Gated Transformers and Enhanced PPO
https://www.mdpi.com/2076-3417/15/16/8928

[2] [10] Vidur: A Large-Scale Simulation Framework for LLM Inference Performance | by SACHIN KUMAR | Medium
https://medium.com/@techsachin/vidur-a-large-scale-simulation-framework-for-llm-inference-performance-1006909e6f36

[3] Improving End-To-End Latency Fairness Using a Reinforcement-Learning-Based Network Scheduler
https://www.mdpi.com/2076-3417/13/6/3397

[6] [2305.10192] Curriculum Learning in Job Shop Scheduling using Reinforcement Learning
https://arxiv.org/abs/2305.10192

[7] [8] Proximal Policy Optimization — Reinforcement Learning Coach 0.12.0 documentation
https://intellabs.github.io/coach/components/agents/policy_optimization/ppo.html

[9] HI , is there a difference between PPO with parallel environments and a multi agent PPO . What do they mean and how do they differ can anyone explain with example .Thanks : r/reinforcementlearning
https://www.reddit.com/r/reinforcementlearning/comments/halzoy/hi_is_there_a_difference_between_ppo_with/