ChatGPT

Generalizing the PPO Warmstart Setup

To avoid hardcoding environment-specific values and make the training pipeline more general, you should refactor how state and action dimensions are determined and how environment parameters are handled. Currently, the state dimension is explicitly calculated for a specific scenario (e.g. 4 replicas with certain per-replica features, yielding `state_dim = 4 × 47 + 10 = 198`). This ties the model to the **Vidur** environment configuration. For better generality:

- **Dynamic State Dimensions:** Instead of fixing `state_dim` in the config, derive it from the environment's observation space or configuration at runtime. For example, in the Vidur code the global state vector is built by concatenating each replica's state and some global metrics [1]. The length of this state depends on the number of replicas and the `max_queue_requests_per_replica` parameter (which defaults to 4 in Vidur's config) [2]. By querying the environment (or a config file) for these values, you can set the network input size dynamically. This way, if you run on a different framework (e.g. a **vLLM** setup with a different number of replicas or features), the code adapts without manual changes. Consider using an environment wrapper or initialization routine that computes `state_dim` and `action_dim` from the environment's observation and action spaces.

- **Abstract Environment Interface:** Ensure your training code interacts with the environment through a consistent interface. Whether using Vidur (a simulated CPU scheduling environment) or vLLM (a GPU-based serving environment), define a common set of methods to get state, apply actions, and retrieve rewards. This might involve writing an adapter for vLLM so that, for example, metrics like throughput or latency are exposed in the same format as Vidur. By not relying on Vidur-specific classes or global variables, the **PPO+warmstart** pipeline becomes framework-agnostic.

- **Configurable Parameters:** Parameters like number of replicas, QPS, etc., should be driven by a config file or command-line arguments rather than being fixed in the code. Your current script already uses variables (e.g. `NUM_REPLICAS=4`, `QPS=3.0`, etc.) – to generalize further, you might have the script detect or be told the environment settings. For instance, in **vLLM** on a GPU, you might have fewer replicas but higher QPS or different request patterns. The training script can accept these as inputs or infer them (for example, reading `cluster_config` in Vidur to get replica count). The goal is to eliminate any **hardcoded constants** tied to one setup. That includes the state construction details – if the state vector includes per-replica request features, the count of those features (`max_queue_requests`) should be a parameter you can adjust or compute, not an assumed "5 requests per replica" etc.

By making these adjustments, the PPO training system can **generalize to different deployment scenarios**. Importantly, you can still **tune it for Vidur** to achieve superior performance there, but the tuning (network sizes, hyperparameters) should not assume Vidur's exact configuration. For example, instead of assuming exactly 4 actions, consider reading the environment's action space (which might be the number of replicas or scheduling choices available). In summary, refactor to use environment introspection and configuration-driven design so the pipeline works out-of-the-box on both the Vidur simulator and a real vLLM GPU environment.

Diagnosing the Performance Issue vs Baselines

You mentioned that after training, the learned policy "does not outperform either round-robin or random." This indicates that despite the complex warm-start and PPO training, the policy's final performance plateaued at or below the level of these simple baseline schedulers. There are a few likely reasons and diagnostics for this outcome:

- **Reward Function Constraints:** The most critical factor appears to be the reward design. The new reward shaping scheme you used is effectively capping positive returns. In fact, you observed that **any positive return gets clipped around 0.79**, whereas negative returns can reach the floor of –4.0. This asymmetric clipping severely limits the agent's ability to see improvement beyond a certain point. In reinforcement learning, if the reward signal for better performance is truncated, the agent might not differentiate between a "good" action and an "even better" action [3]. It will learn to achieve the clipped reward and then have no incentive (in terms of reward feedback) to improve further. This could explain why the policy "flattens" in performance below Round Robin – it literally isn't rewarded for surpassing what Round Robin achieves. The decision to scale and clip rewards (to [-4, 4] in your config) was likely made for stability, but overly aggressive clipping can mask important differences [4]. Engstrom et al. (2020) have shown that reward scaling significantly affects performance, and that while some clipping is common (e.g. clipping Atari rewards to [-10,10]), there's **no clear evidence that clipping after scaling helps learning** [4]. In your case, the clipping threshold (~0.79 scaled) is too low, effectively acting as a performance ceiling.

- **Legacy Penalties Dominating:** You mentioned a "legacy penalty" in the reward. From the config, there are terms like load balance penalty, latency penalties, etc. If these penalty terms are too heavy, the agent might be playing it safe to avoid negative rewards instead of pushing for higher positive rewards. For instance, if the reward function heavily penalizes latency beyond 6.0 seconds (with `latency_penalty_scale=0.85`) or imbalance (`balance_penalty_weight=0.2` and `load_balance_penalty=1.0`), the agent might prefer a conservative strategy (like Round Robin) that avoids worst-case penalties but also never achieves the best throughput. **Round Robin** is known for fairness and predictable behavior, which might avoid spikes in latency (thus avoiding big negative hits) [5]. Your RL policy, being cautious due to the penalty, might be essentially mimicking a Round Robin-like allocation to stay out of trouble. Meanwhile, the **Random** policy, by chance, might sometimes exploit short-term advantages (or at least, in a stochastic sense, it might not consistently hit the worst-case). If the RL agent's behavior cloning initialization included data from Round Robin and Random, it could also be averaging those behaviors initially, and strong KL regularization in early PPO phases would keep it close to that mix. In summary, the reward design might be **over-penalizing bad outcomes** relative to rewarding good outcomes. This imbalance makes the agent risk-averse and content with sub-baseline performance that avoids penalties.

- **Demonstration and KL Effects:** Your training went through a warm-start with behavioral cloning from a mixture of policies (Round Robin, LOR, Random). While this is great for giving the agent a good initial policy, it may also bias it towards those strategies. The PPO phase used KL regularization (`kl_coef`, reference policy decay) to ensure the policy doesn't stray too rapidly from the pre-trained behavior. If that KL constraint was too tight for too long, the agent's exploration could be stifled. However, according to your config, the KL reference coefficient decays to 0 by 5000 steps, so eventually the policy was free to move. At that point, the primary limiter becomes the reward structure itself. Still, it's worth considering if the **behavior cloning stage** provided enough diversity in demonstrations. Mixing Random with Round Robin/LOR gives

coverage of different strategies, but it might also introduce a lot of suboptimal actions (random) that the agent then has to unlearn or avoid exploiting. If the agent converged "quickly and then flattens," it could mean it mostly latched onto a Round Robin-like strategy (since that's a stable local optimum under the clipped reward). Essentially, the agent might have converged to the safest strategy present in its demos and gotten stuck there because pushing beyond yields no visible reward gain (due to clipping).

- **Insufficient Positive Signal:** Another angle – how does Round Robin compare in raw reward versus the optimum? It could be that Round Robin already achieves close to the maximum reward under your shaping (0.79 out of a possible 0.79, if that's the cap). If so, the RL agent literally cannot exceed it because the reward function doesn't allow any policy to score higher. That would directly explain why it didn't outperform the baseline: by design of the reward, **nothing** can. Alternatively, if something could in theory reach closer to +4.0 reward (the upper clip), the agent never discovered those behaviors, likely because the combination of demonstration bias + strong penalties made those behaviors appear worse initially (or too risky). Without a clear reward gradient toward those better behaviors, PPO won't find them.

In summary, the current results do **not indicate a fundamental flaw in PPO or training stability** – rather, they highlight a reward design issue. The agent is stable and learning, but it's learning exactly what the reward function is encouraging (stay safe, avoid big negatives), and that inadvertently aligns with mediocre scheduling performance.

Reward Shaping Fixes and Next Steps

Since training is numerically stable (entropy ~1.35, KL < 0.06, no sign of divergence or oscillation), the solution lies in **adjusting the reward shaping to give the agent more headroom to improve**. Here's how you can address the issues:

- **Increase Positive Reward Headroom:** Loosen or raise the positive reward clip. Currently, positive returns are effectively capped at ~0.79. You might remove the clipping entirely on the high end or set the clip_range to a higher value (e.g. allow up to [+4.0, -4.0] or even more asymmetrically [+8, -4]). The OpenAI Baselines' `ClipRewardEnv` and VecNormalize clip are typically in the range [-10, 10] [4]; your clip at 4 is relatively tight. By allowing higher positive rewards, if the agent finds a way to exceed the current baseline (e.g. improve throughput or latency beyond what Round Robin achieves), it will actually see a higher return instead of hitting a ceiling. This provides **incentive for continued improvement**. Just be cautious to monitor training stability when you raise the cap – if rewards get too large, you might need to scale down the magnitude (but it sounds like even the absolute cap of 4.0 wasn't being reached on the positive side, so there's room).

- **Tune Down Penalties:** Dial back the "legacy" penalties that are suspect. For instance, consider reducing `load_balance_penalty` and the weighting on latency penalties. If the latency threshold (6.0) and penalty scale (0.85) currently slam the reward whenever latency goes above 6s, maybe you can soften this to not discourage the agent from occasionally letting a slightly longer request finish if it enables much higher throughput. You could increase the threshold or reduce the scale (so the penalty grows more slowly beyond the threshold). Also re-examine `balance_penalty_weight`: a weight of 0.2 might seem small, but if combined with other factors it could still bias the agent toward spreading load evenly at the cost of throughput. The idea is to **encourage exploration of more aggressive scheduling** by not immediately punishing the agent for, say, overloading one replica if that yields better overall throughput for a short period. Gradually reducing penalty weights and observing if the learned policy starts to outperform

Round Robin can guide you – you want to find a balance where the agent is kept in check from truly bad behavior, but not so much that it never discovers anything better than the status quo.

- **Adjust Throughput Target or Reward Scaling:** If `throughput_target=1.85` is acting like a cap (perhaps the reward gives diminishing returns as it approaches 1.85 QPS), you might increase this target or change how the reward scales near it. For example, if currently reaching 100% of that target yields 0.79 reward, maybe allow the reward to keep increasing beyond that target (or raise the target to a higher number so the agent effectively sees it can go higher). Similarly, the `scale_factor: 1.5` in reward scaling could be revisited. That factor together with clip_range yielded the 0.79 cap (the exact math isn't in your summary, but likely the raw environment return was around ~0.5 and scaled up). If you lower the scale_factor, you might reduce the chance of hitting the clip so soon. Alternatively, remove that linear scaling and do a more nuanced normalization (perhaps learn a normalization or use a moving average baseline for advantage calculation – though you do have GAE and value function which handle some of this).

- **Monitor Reward Distribution:** It's helpful to log the distribution of raw rewards (pre-scaling and clipping) during training. If you consistently see that rewards hit the +0.79 ceiling and flatline, that's confirmation that the agent is constrained. After making adjustments, check that the agent's episodic returns have room to grow. Ideally, you want the baseline (Round Robin) to score somewhat lower in reward than the maximum possible, leaving the RL agent a gradient to climb. In other words, **engineer the reward so that baseline ≠ ceiling**. If Round Robin currently gets ~0.7 reward on average, maybe the max under new shaping could be, say, 2.0 or more, so the agent could in theory double the reward by a much better strategy. This gap will encourage the agent to search for that better strategy.

- **Gradual Relaxation:** Since your training pipeline is stable, you can afford to gradually apply these changes and see their effect. For example, you might first double the positive clip range and run training – if you start seeing the policy surpass baseline in reward (and hopefully in real performance metrics), you know you're on the right track. If training stays stable, you can then also reduce penalties, etc. Because you have strong monitoring (TensorBoard, KL tracking, etc.), you'll notice if any change destabilizes training (e.g., entropy collapsing or KL shooting up). So far, all indicators show PPO is well-behaved; thus, these reward tweaks shouldn't introduce instability if done in moderation.

- **Reevaluate Baseline Demos:** As a side note, once reward shaping is fixed, consider if you still need the random policy in demonstrations. A mixed-policy demonstration was used to give a wide coverage of behavior. If the RL policy is consistently underperforming even Random, it's worth asking if including Random in the warm-start might be injecting too much noisy behavior. Perhaps focusing demos on Round Robin and a slightly smarter heuristic (instead of pure random) could yield a better starting policy. However, this is secondary compared to reward issues – even a perfect demo won't help if the reward won't allow surpassing a threshold.

In conclusion, **there's no sign of intrinsic PPO issues** (no collapse, no oscillation, which aligns with PPO's design to maintain stability [6]). The agent is basically doing what it was told to do by the reward structure. To get it to beat Round Robin, you must **change the "goalposts"**: allow higher positive rewards and ease up on the constraints that are keeping it playing safe. This kind of reward engineering is a normal part of applied RL development – it often takes a few iterations to get it right [7]. Once you make these adjustments, you should observe the policy begin to explore strategies that yield higher throughput (or lower latency) than Round Robin, because it will finally be rewarded for doing so. Keep an eye on the actual performance metrics (throughput, latency) as well, not just the shaped reward, to ensure that improvements in reward correspond to real gains. With a more permissive reward function

and the strong foundation you've built (warm-start + advanced PPO features), the agent has a much better chance to demonstrate **superior performance on Vidur** and generalize to other settings.

Sources:

1. Vidur scheduler state construction – the global state vector is built by concatenating per-replica states (length depends on number of replicas and requests considered) plus global metrics [1] [2]. This shows how state dimensions are tied to environment settings.
2. Stable Baselines3 documentation – emphasizes that reward engineering often requires several iterations to achieve desired behavior [7].
3. ICLR Blog: 37 Implementation Details of PPO – notes that clipping rewards can make training stable but at the cost of obscuring performance differences, and there's no clear evidence that clipping after scaling is beneficial [3] [4]. This supports the need to adjust or remove the strict reward clipping in your setup.
4. General Round Robin scheduling knowledge – Round Robin's fairness can avoid extreme bad cases, making it a strong baseline in some scenarios [5], which your agent must overcome by being allowed to take smarter risks (via a refined reward function).

---

[1] [2] random_global_scheduler_with_state.py
https://github.com/pkucaoyuan/Vidur/blob/f843bdeebbb616cbd688b4179ca651a23e650b7f/vidur/scheduler/global_scheduler/random_global_scheduler_with_state.py

[3] [4] The 37 Implementation Details of Proximal Policy Optimization · The ICLR Blog Track
https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/

[5] The efficiency-fairness balance of Round Robin scheduling
https://www.sciencedirect.com/science/article/pii/S0167637721001619

[6] [7] Reinforcement Learning Tips and Tricks — Stable Baselines3 2.7.1a3 documentation
https://stable-baselines3.readthedocs.io/en/master/guide/rl_tips.html