Action Chunk Scheduling for Batched Robot Policy Serving

Rohan Bansal*, David He*, Nadun Ranawaka Arachchige, Zhenyang Chen, Soobum Kim, Kexin Rong†, Danfei Xu†
* Equal Contribution, † Equal Advising
Georgia Institute of Technology

Abstract: Deploying robot foundation models at scale is the next step towards realizing the potential of general-purpose robots. However, Vision-Language-Action (VLA) and other foundation models are computationally demanding, and on-device compute is constrained by power and space. In this paper, we introduce the problem of serving a robot policy to multiple robots from a remote GPU and formulate it as a scheduling problem. We build Armory, a serving system validated on fleets of both simulated and real robots. Our experiments show that naive scheduling heuristics perform well when all robots are the same, but fall short when robots consume action chunks at different rates, uncovering a mismatch between conventional batching methods and the closed-loop requirements of robot policy execution. To address this, we propose a scheduling algorithm that accounts for this heterogeneity and improves overall system throughput by up to 18% in real-world experiments. Additional details are available at https://gatech-rl2.github.io/actionchunkscheduling/.

Keywords: Robot Learning Systems, Robot Foundation Models

1 Introduction

Robot policies are becoming increasingly capable [1, 2, 3, 4, 5], and deploying them at scale turns policy inference into a *policy serving problem*. In contrast to other machine learning serving problems, robots have strict latency requirements: robots act in closed loop with the physical world and must react quickly to changes in the environment. While local inference avoids network delay, it can be difficult to run foundation-scale policies at real-time rates on power-constrained hardware [6]. Cloud serving is therefore a natural alternative: a powerful GPU can host larger models while amortizing overhead across many robots.

In this paper, we study the multi-tenancy problem for robot policy serving: *how should a single powerful GPU serve many robots through batched inference?* This setting is enabled by action chunking, where each policy query produces a sequence of future actions rather than a single action [7]. Since a robot can execute the current chunk while the next query is being processed, action chunking reduces the required inference frequency and makes remote serving practical. Recent works have proposed *asynchronous execution*, where inference overlaps with execution so that robots can continue execution while waiting for the next chunk [8, 9, 10]. However, these systems primarily address single-robot serving, where the GPU may remain underutilized between requests.

The core difficulty is that batching improves GPU efficiency at the cost of robot responsiveness. Larger batches amortize inference costs, but they also increase latency. Unlike in LLM serving [11], latency for robotics is not merely a service-level metric. It changes the closed-loop behavior of the robot. If a robot executes a chunk for too long, its actions become stale and it loses reactivity. If the next chunk arrives too late, the robot starves, leaving it with no valid action to execute. This tradeoff is especially difficult for heterogeneous fleets: dynamic tasks may require short execution horizons and frequent updates, while quasi-static tasks may tolerate longer horizons. A system that treats all robots uniformly can therefore underserve the robots whose tasks are most sensitive to delay.

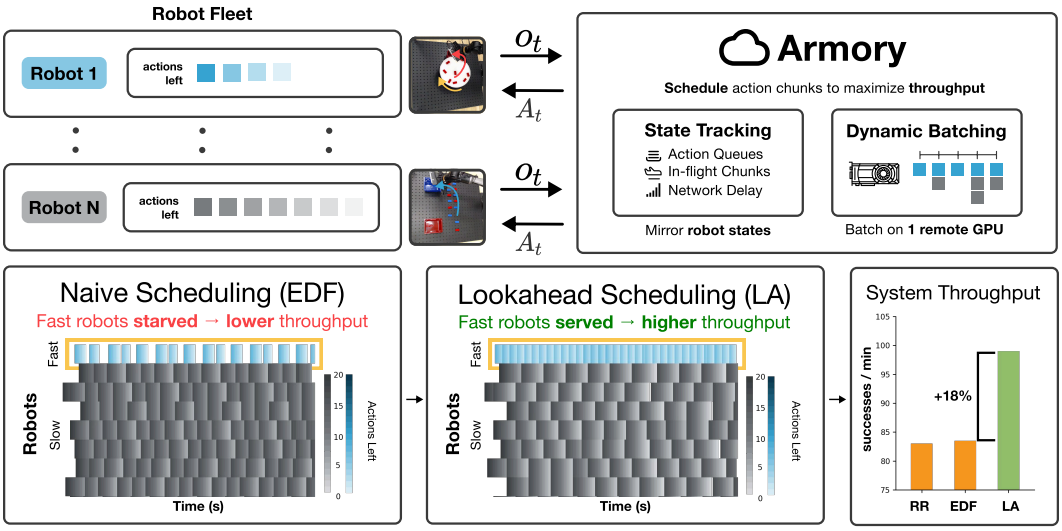


Figure 1: Armory is an end-to-end serving system for deploying robot foundation model policies from the cloud that tracks robot states and schedules action chunks to minimize robot starvation. Naive scheduling underserves fast robots completing highly dynamic tasks, so we propose Lookahead scheduling to improve fast robot service with minimal impact on other robots, significantly boosting system throughput under heterogeneous workloads.

Our key insight is that batched robot policy serving should be formulated as a *closed-loop scheduling problem*. The server should decide which robots to serve by planning how each batch affects future execution: which robots will continue acting, which will starve. We instantiate this idea in **Armory**, a serving system for batched robot policy inference. Armory maintains a server-side mirror of each robot's action queue, in-flight chunks, execution horizon, and communication delay. We formulate action chunk scheduling as a Markov Decision Process in which the server action is a batch of robots to serve and the reward is total executed robot time. Since exact planning is intractable, we propose Lookahead, a practical lookahead scheduler that simulates candidate batches forward and selects the batch with the best predicted reward normalized by inference time.

We evaluate Armory in both simulation and on a fleet of 10 real robots. Our experiments show that acceptable execution horizons are task-dependent: dynamic manipulation degrades sharply with stale chunks, while quasi-static manipulation is more tolerant. We also show that starvation is not merely a systems metric; it directly reduces downstream throughput, especially for dynamic tasks. Finally, we find that simple schedulers are competitive in homogeneous fleets, but heterogeneity-aware scheduling is needed when robots have different reactivity requirements. In real-world experiments, lookahead scheduling improves overall system throughput by up to 18% in heterogeneous settings while exposing a controllable tradeoff between serving fast and slow robots.

2 Preliminaries

Action Chunking. A robot policy processes an input observation o_t to produce an action chunk $A_t = [a_i, a_{i+H-1}]$, where H is the prediction horizon and i is the **action index** of the first action. During execution, a robot steps at a pre-defined control frequency f_c, capturing o_t and executing a_t. In *synchronous execution*, a robot executes a chunk to the end of its horizon before generating a new chunk. Generating a new chunk with the policy incurs non-zero delay, creating gaps between executed chunks that appear as pauses or discontinuous motion. To address this, many works adopt *asynchronous execution* [8, 9], where the inference for the next chunk starts while the current chunk is still executing. To ensure continuity, executed action indexes must strictly increment [12].

Execution Horizon. In practice, a robot only executes a prefix of the full prediction horizon H. The length of this prefix should be restricted to an interval $[H_{\text{min}}, H_{\text{max}}]$. If H_{min} is too small, frequent switching between chunks increases compounding errors [13, 14], and if it is too large, robots react too slowly to changes in the environment. Empirically, we find the optimal interval is task-dependent (Figure 3a), motivating the need to serve workloads with heterogeneous execution horizons.

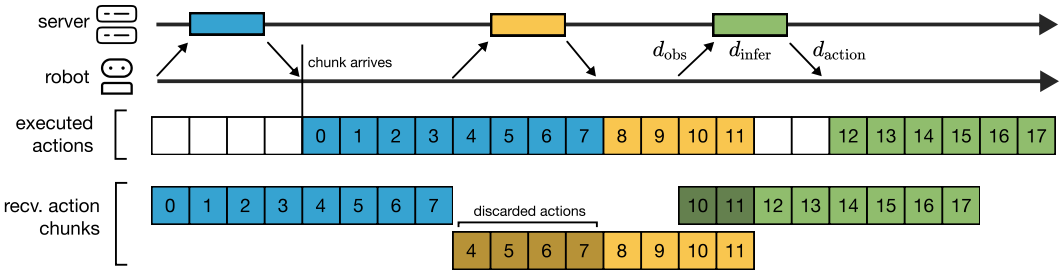


Figure 2: **Execution Timeline.** A robot with $H_{\text{max}} = 8$ consumes actions by incrementing the action index, pausing when it is starved. When starvation occurs while a chunk is still in inference/network, the chunk's execution will extend more than H_{max} steps after the observation was sent.

Delay. In the real-world, there exists some delay d from the time the robot captures observation o_t to the time it executes an action from the corresponding chunk A_t. We decompose d into three main components: the observation to server delay d_{obs}, the policy inference delay d_{infer}, and the server to action rollout delay d_{action}. In cloud serving, d_{obs} and d_{action} are dominated by network round-trip time between the robot and the policy server; in on-device or ideal local setups they are effectively zero. d_{infer} is non-negligible in either setting and typically spans multiple control steps.

Starvation. The primary objective of robot policy serving is minimizing the number of control steps where a robot has no actions to execute, which we refer to as a *starvation*. Starvations are harmful because they slow robot execution, induce jerky motions, and impair policy performance. As with the execution horizon, we hypothesize the cost of starvation to be task-dependent and validate this empirically in Figure 3b. These task-dependent responses create heterogeneity in inference demands when serving different tasks concurrently, posing challenges we address in Section 3.

Batching. Batching on a GPU significantly increases inference throughput, measured in requests per second, since the cost of loading model weights is amortized across the batch. However, increasing the batch size increases d_{infer}, delaying the arrival of every generated chunk. A good serving system must use *dynamic batching* to carefully balance the tradeoff between throughput and latency.

3 Scheduling for Batched Robot Policy Serving

Armory treats batched policy serving as a control-aware scheduling problem. Action chunk execution depends on when chunks are generated by the server and when they are received by the robot. In the following section, we describe the complex dynamics of action chunk generation and execution, which Armory models with a server-side software mirror.

3.1 Batched Policy Serving as an MDP

We formulate batched robot policy serving as a Markov Decision Process. Robots are indexed by $j \in 1, \ldots, N$, and scheduling epochs are indexed by k. At epoch k, the server observes state s_k and selects a non-empty batch $B_k \subseteq 1, \ldots, N$ of robots to serve.

Server-side execution state. For each robot j, Armory tracks three quantities:

$$i_j \quad \text{action index of the latest executed robot step} \tag{1}$$

$$\hat{i}_j \quad \text{action index of the latest robot step received by the server} \tag{2}$$

$$\mathcal{Q}_j \quad \text{queue of generated action chunks} \tag{3}$$

Inside $\mathcal{Q}_j$, an action chunk for robot j is written as

$$c = (i_{\text{start}}, h_j, t_{\text{arr}}), \tag{4}$$

where i_{start} is the action index corresponding to the first action in the chunk, h_j is the robot's execution horizon H_{max}, and t_{arr} is the time the chunk arrives at the robot. $\mathcal{Q}_j$ stores all chunks for a robot, including chunks that have been generated by the server but have not yet arrived at the robot.

The full serving state is

$$s_k = \big(t_k, i_j, \hat{i}_j, \mathcal{Q}_j\big), \tag{5}$$

where t_k is the wall-clock time at the start of the scheduling epoch.

Transition. Let per-batch-size inference latencies be $\tilde{d}_{\text{infer}(b)}$ for batch size b, and stochastic per-robot networking latencies $\tilde{d}_{\text{obs},j}, \tilde{d}_{\text{action},j}$. Given state s_k and batch B_k, inference completes at

$$t_{k+1} = t_k + \tilde{d}_{\text{infer}(|B_k|)}. \tag{6}$$

For each served robot $j \in B_k$, the resulting action chunk added to $\mathcal{Q}_j$ is

$$c^j_{\text{new}} = \big(\hat{i}_j, h_j, t_{k+1} + \tilde{d}_{\text{action},j}\big). \tag{7}$$

During the same interval, all robots tick forward at f_c. For each control tick $\tau \in [t_k, t_{k+1}]$, robot j advances by one action index if there is a chunk in $\mathcal{Q}_j$ that has arrived and covers the current index:

$$i_j \leftarrow i_j + 1 \quad \text{if } \exists c \in \mathcal{Q}_j : t_{\text{arr}} \le \tau \wedge i_j \in [i_c, i_c + h_c). \tag{8}$$

The server also updates $\hat{i}_j$ with the latest control step that has arrived at the server by t_{k+1}:

$$\hat{i}_j \leftarrow \max i : t_i + \tilde{d}_{\text{obs},j} \le t_{k+1}, \tag{9}$$

where t_i is the wall-clock time when action index i was reached on the robot.

Reward. Let $\Delta i_j(s_k, B_k)$ denote the number of non-starved control steps robot j executes during the transition induced by batch B_k. The per-epoch reward is the total executed robot time:

$$R(s_k, B_k) = \frac{1}{f_c} \sum_{j=1}^{N} w_j \Delta i_j(s_k, B_k). \tag{10}$$

where w_j is an optional task-dependent weight. With $w_j = 1$ for all robots, the objective treats all executed robot time equally. Larger w_j values prioritize robots whose tasks require fresher actions, such as dynamic tasks with shorter execution horizons or higher starvation sensitivity.

A scheduling policy π maps serving states to batches. The objective is

$$\pi^\star = \arg\max_{\pi} \sum_{k} \mathbb{E}\big[R(s_k, B_k)\big]. \tag{11}$$

3.2 Scheduling

Exact planning in this MDP is intractable because future batch choices create a branching tree over robot execution states and networking latencies are stochastic. Thus, we propose **Lookahead**, a scheduling algorithm that plans a few steps ahead using the server-side software mirror.

At each scheduling epoch, Lookahead enumerates all potential schedules of L epochs. A schedule S is a sequence of states and actions:

$$S = (s_0, B_0),\ (s_1, B_1),\ \ldots,\ (s_{L-1}, B_{L-1}). \tag{12}$$

For each schedule, it rolls the simulation forward and calculates a score $\text{Score}(S)$. Lookahead then dispatches the first batch B_0 from the highest scoring schedule. In this work, we consider a score that normalizes the cumulative reward by the time spent on the GPU:

$$\text{Score}(S) = \sum_{\ell=0}^{L-1} \frac{R(s_\ell, B_\ell)}{d_{\text{infer}}(|B_\ell|)}. \tag{13}$$

We compare Lookahead against two standard scheduling primitives. **Round Robin (RR)** cycles through robots in batches of maximum size b, serving robots fairly but ignoring urgency. **Earliest Deadline First (EDF)** schedules the b robots predicted to run out of executable actions soonest. EDF captures imminent starvation, but treats all starvation events as equally costly and does not model how the chosen batch affects future serving states. Lookahead instead evaluates batches by their induced execution trajectory, allowing the scheduler to trade off GPU efficiency, starvation avoidance, and task-dependent reactivity.

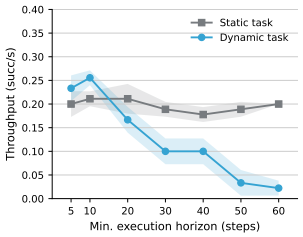
3.3 Armory

We implement Armory, an end-to-end serving engine for batched robot policy inference, designed around the software mirror of the MDP described in 3.1. To prioritize mirror accuracy, Armory uses a push-based communication architecture, where each robot sends the captured observation o_t before executing action a_t. The server tracks the latest received action index $\hat{i}_j$, queued chunks $\mathcal{Q}_j$. Latencies d_{infer} are profiled on startup and d_{obs} and d_{action} are continuously estimated for each robot.

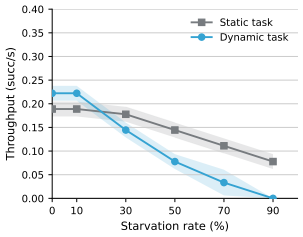
Armory separates network handling, scheduling, and GPU inference into independent processes communicating through shared memory, so that CPU-side work never blocks the GPU. The scheduler maintains the execution mirror and is work-conserving: whenever the GPU is idle and at least one useful request exists, it dispatches a batch. More implementation details are in Appendix A.9.

4 Experiments

In this section, we describe the evaluation of our suite of scheduling algorithms in real-world and simulation experiments under different levels of robot heterogeneity. We find that under a homogeneous workload, simpler algorithms may suffice, while under heterogeneous workloads, there is a tradeoff between overall system starvation and throughput. For all evaluations, we use $\pi_{0.5}$ [4] as a representative robotics foundation model.



(a) Execution horizon.



(b) Starvation rate.

Figure 3: Dynamic task throughput is more sensitive to stale execution horizons and starvation than that of quasi-static tasks.

4.1 Setup

Workload and Heterogeneity. Not all robots are equal: some must be more reactive than others. We capture this through two robot classes that differ in execution horizon. A *fast* robot runs what we call a "dynamic" task with a shorter H_{max}, since policy performance degrades quickly when chunks are stale while manipulating moving objects; a *slow* robot tolerates longer horizons for quasi-static manipulation (Figure 3a), running what we call "static" tasks. Dynamic tasks are sensitive to loss of reactivity, so starvation also hurts fast robots disproportionately (Figure 3b). Note that *fast* and *slow* refer to the speed at which a robot exhausts its chunks as a result of its horizon, and **not** its movement/step speed; robots that exhaust chunks faster are more reactive for this reason.

We vary heterogeneity across three configurations: *one-fast* (one fast robot, rest slow), *half-fast* (half fast, half slow), and *all-fast* (all robots fast, none slow). These span the heterogeneity spectrum and stress the scheduler in distinct ways. *All-fast* is a homogeneous setting that isolates the effect of fleet size from heterogeneity. *Half-fast* maximizes scheduling tension: fast and slow robots compete equally for batch slots, so bias toward one class visibly starves the other. *One-fast* tests whether the scheduler can protect a single sensitive robot from many less urgent ones; a realistic analog is a robot that may become highly dynamic for a short period (e.g., striking a match) before resuming slower operation.

Server Specifications. For both simulation and real-world experiments, the cloud server is colocated on the network (round-trip delay $\sim$ 5–10 ms) and uses a single L40S GPU. We show in Appendix A.6 that serving remains (1) practical with network delay up to 50 ms (a typical US coast-to-coast round trip), beyond which added delay simply acts as a stronger starvation signal, and (2) robust to network jitter from mild to extreme.

Simulation. Our cloud server runs a $\pi_{0.5}$-LIBERO checkpoint ($H = 10$, $f_c = 20$) on a node separate from the simulation clients. Both classes set $H_{\text{min}} = 1$; *fast* robots use $H_{\text{max}} = 6$ and *slow* robots use $H_{\text{max}} = 10$. We develop a LIBERO [15] evaluation harness that steps each robot independently in real time. Each robot is assigned one of two randomly sampled LIBERO-10 tasks and completes as many episodes as possible in a 5-minute window (30-second time-out per episode).

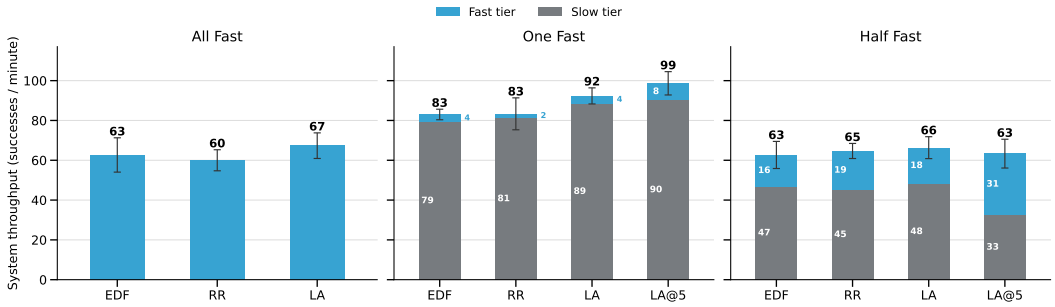


Figure 4: We compare system throughput in legos-per-minute across the three heterogeneous scenarios in a real-world setting. We observe that LA@5 significantly improves fast-tier throughput in the one-fast case by almost $2\times$ over EDF, and provides a small boost to slow-tier as well, increasing system throughput by approximately 18% over baselines. In the half-fast case, we can maintain system throughput but trade off towards fast robots, boosting their share of system throughput significantly. Counterintuitively, there are some cases when throughput increases despite an increase in starvation, which we analyze in Appendix A.8.

Heterogeneity is applied per task, where task 1 robots are *fast*, task 2 robots are *slow* (excluding all-fast, where all tasks are fast). We sweep 3 seeds and fleet size from 2 to 10 across all 3 configurations.

Real World. We deploy 10 AgileX PiPER arms with top and wrist cameras (Intel RealSense D435i), served by a $\pi_{0.5}$ checkpoint finetuned at $H = 20$ and $f_c = 30$. Both classes use $H_{\text{min}} = 5$ to prevent excessively multimodal motions; *fast* robots use $H_{\text{max}} = 10$ and *slow* robots use $H_{\text{max}} = 20$. We fine-tune on two tasks: static brick sorting (sorting colored bricks into bins) and dynamic turntable sorting (picking moving bricks off a turntable into a mug). Fast robots run the turntable task and slow robots run bin sorting, except in the homogeneous setting where all run bin sorting. We evaluate 3 seeds per scheduler per configuration, similar to sim.

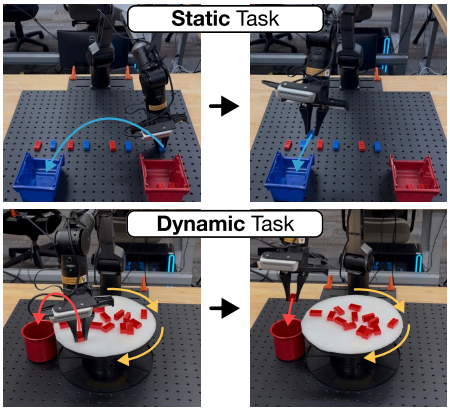


Figure 5: Real-world task setup. Top is the static brick-into-bin sorting task, and bottom is the dynamic brick-into-mug turntable task.

Schedulers. For all experiments, we analyze the Round-Robin, Earliest-Deadline-First, and Lookahead schedulers described in 3.2. In simulation, we set $b = 3$, and in real, we set $b = 3$ for the homogeneous and $b = 5$ for the heterogeneous setting. These choices favor RR and EDF; we sweep batch sizes to find the best-performing ones for these two schedulers to make a fair comparison. See Appendix A.4 for detailed ablations on batch size.

To study the tradeoff ability of Lookahead, we set $w_j = 1$ for all slow robots and compare three weightings $w_j = 1, 3, 5$ for the fast robots, each denoted as LA@w_j. Intuitively, $w = 1$ treats all robots equally, $w = 3$ moderately prioritizes robots with shorter horizons and higher starvation sensitivity, and $w = 5$ is an aggressive setting for deployments where protecting fast robots is worth sacrificing slow-tier service. Although the formulation permits multi-step receding-horizon planning, we find that a one-step model-based score is sufficient for heterogeneity-aware scheduling (Appendix A.5), so we use the simplest instantiation of the framework with $L = 1$.

Metrics. We report two primary metrics: *starvation rate*, the fraction of control steps where a robot has no action to execute, and *throughput*, the number of successes a robot produces in a unit time. In simulation, throughput is the number of successful task completions each robot achieves per minute of wall-clock time. In the real world, throughput is the number of Lego pieces each robot sorts during a fixed 60-second window (also per-minute).

4.2 Evaluations

Weighted scheduling exposes a controllable throughput tradeoff. In simulation, we sweep fleet size over the three scenarios. In the homogeneous all-fast setting, all methods perform comparably;

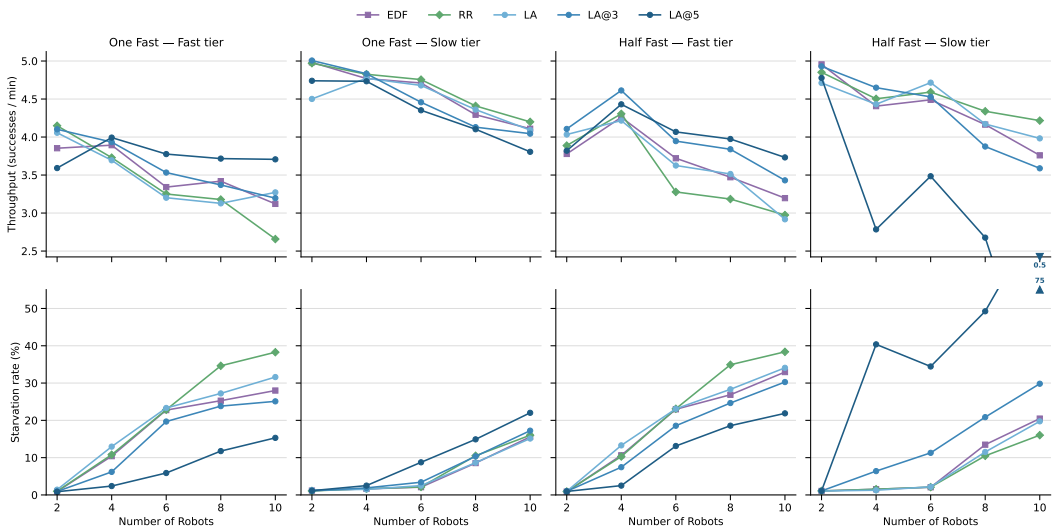


Figure 6: LIBERO-10 simulation results. For one-fast, LA@5 is able to maintain a consistently higher throughput and lower starvation for fast-tier robots without sacrificing slow-tier performance. For half-fast, LA@3 is the most useful operating point: it improves fast-tier service while keeping the slow-tier close to the fixed-priority baselines. LA@5 shows the boundary of the tradeoff, where further prioritizing fast robots begins to significantly sacrifice slow-tier performance.

we delegate these results to Appendix A.7 and focus on the more revealing heterogeneous settings. When fast and slow robots share a GPU, the scheduler must choose between maximizing average service and protecting robots that exhaust chunks faster. Figure 6 decomposes throughput and starvation by robot tier. In the one-fast setting, weighting the fast robot higher allows Lookahead to serve it before it starves, improving fast-tier throughput while preserving slow-tier performance thanks to longer action buffers. In the half-fast setting, the same weighting creates a sharper tradeoff: moderate weighting improves fast-tier throughput with limited slow-tier degradation, while aggressive weighting over-serves fast robots at the expense of slow ones. This is desirable for an operator-facing scheduler: the weight parameter controls throughput allocation between robot classes, and in some scenarios (namely one-fast), *we can gain throughput for free*. Appendix A.8 reports per-tier metrics as fleet size increases.

Scheduling gains transfer from simulation to real hardware. On a fleet of ten PiPER arms (Figure 4), the homogeneous setting again shows all schedulers performing comparably. In heterogeneous deployments, Lookahead provides a meaningful control surface: in the one-fast setting, Lookahead at $w = 5$ boosts fast-tier throughput by $4\times$ over RR and $2\times$ over EDF while preserving slow-tier throughput. In the half-fast setting, the same weighting improves fast-tier throughput significantly at the cost of slow-tier service, exposing a sharper tradeoff that operators can tune to their priorities. Across both settings, the main benefit of Lookahead is not universal dominance over fixed baselines, but the ability to use a model of action-buffer dynamics to select a desired operating point on the throughput/starvation tradeoff.

5 Related Work

Robot Policy Inference. Robot foundation models are rapidly scaling: early VLA systems [1] showed that large VLM backbones could be adapted to robotic control, and more recent models pair even larger backbones with broader cross-embodiment datasets [2, 3, 4, 16, 5, 17, 18]. This trend intensifies as the field expands to video prediction, with state-of-the-art models taking up to 7 seconds per inference [19]. On the systems side, [6] profiles the bottlenecks of VLA inference, [20] performs inference-time verification, and [21] pushes latency down through kernel-level optimization, while [22] modifies policy formulation to synthesize actions faster. Closer to our setting, several methods improve latency tolerance by overlapping inference with execution through real-time chunking [8, 10], reasoning over future state during async inference [9], accelerating rollout [23],

or combining action scheduling with controller tuning [24]. These efforts, however, are orthogonal: they make individual policies faster or more latency-tolerant, whereas we schedule shared compute across multiple robots and are architecture-agnostic, benefiting from any per-model speedup.

ML Serving Systems. Early work on low-latency prediction serving and pipeline provisioning established modular serving abstractions, cost-aware configuration, and latency-SLO management for generic inference workloads [25, 26, 27]. More recent systems for transformer and LLM serving introduce iteration-level scheduling, efficient memory management, prefix caching, chunked prefills, and disaggregated execution to improve throughput and tail latency [28, 29, 30, 11, 31]. While many of these systems may optimize for latency, they do not serve models where the latency directly affects model performance. Robot policy serving is uniquely challenging in that delayed action chunks can directly lead to downstream execution failures.

Real-Time Scheduling. Classical real-time scheduling studies how recurring tasks can be ordered to satisfy timing constraints, with foundational formulations dating back to 1973 [32]. In adjacent systems settings, [33] introduces plan-ahead rescheduling for deadline-aware jobs in dynamic heterogeneous clusters, while [34] studies heterogeneity-aware scheduling for shared accelerator clusters running machine learning workloads. These works provide useful abstractions for deadlines, resource contention, and heterogeneous compute allocation. However, they do not address robot policy serving, where inference requests are coupled to closed-loop control, network, and action-chunk starvation; our formulation adapts the receding-horizon idea to this setting, where the cost of a scheduling decision depends on the physical state of each robot.

Two recent works also target multi-robot serving. Kairos [35] minimizes end-to-end task latency through variable execution horizon and dynamically prioritizing action generation and execution. ROSA [36] maximizes action throughput in robot factories through a static provisioning schedule for known workloads, and further assumes synchronous execution. Both treat latency as a target, and neither models the closed-loop consequence of a late chunk, i.e. the downstream starvation and execution cost it induces. This gap matters heavily under heterogeneity, the differing rates at which robots exhaust chunks and starve: our work folds starvation and closed-loop control into an MDP that allows us to schedule against each robot's execution state and protect different classes of robots.

6 Conclusion

In this work, we introduce multi-tenant robot policy serving as a closed-loop scheduling problem. By formulating the server's batching decisions as an MDP over robot action queues, execution horizons, and communication delays, we showed that policy serving has structure that generic batching strategies leave on the table. Our experiments in simulation and on a fleet of ten real robots confirm three findings. First, the cost of latency in robot serving is task-dependent: dynamic manipulation degrades sharply with stale chunks and starvation, while quasi-static tasks are more tolerant. Second, when all robots share the same timing requirements, simple schedulers like Round Robin and Earliest Deadline First perform comparably to more sophisticated alternatives. Third, when robots have heterogeneous reactivity demands, a Lookahead scheduler that simulates action chunking dynamics can improve system throughput by up to 18% in the real world by selectively protecting latency-sensitive robots, with tunable weights that let operators choose which robots to prioritize.

7 Limitations

In recent years, many works have proposed inference optimizations for large transformer models. Notable techniques include continuous batching [28], PagedAttention [29], chunked prefill [11], and prefill-decode disaggregation [31, 37]. State-of-the-art serving systems also typically implement distributed inference to support larger models. In this work, we assumed the simplest inference model (batching on a single GPU). We hypothesize there is more room for improvement by co-designing action chunk scheduling together with these inference optimizations.

Acknowledgments

The real-world experiments presented in this work were conducted in the Advanced Robotic Manipulation (ARM) Lab at Georgia Tech.

References

[1] B. Zitkovich, T. Yu, S. Xu, P. Xu, T. Xiao, F. Xia, J. Wu, P. Wohlhart, S. Welker, A. Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183. PMLR, 2023.

[2] M. J. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, G. Lam, P. Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

[3] K. Black, N. Brown, D. Driess, A. Esmail, M. Equi, C. Finn, N. Fusai, L. Groom, K. Hausman, B. Ichter, et al.
pi_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

[4] P. Intelligence, K. Black, N. Brown, J. Darpinian, K. Dhabalia, D. Driess, A. Esmail, M. Equi, C. Finn, N. Fusai, et al.
$pi_{0.5}$: A vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.

[5] J. Bjorck, F. Castañeda, N. Cherniadev, X. Da, R. Ding, L. Fan, Y. Fang, D. Fox, F. Hu, S. Huang, et al. Gr00t n1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.

[6] W. Jiang, J. Clemons, K. Sankaralingam, and C. Kozyrakis. How fast can i run my vla? demystifying vla inference performance with vla-perf. *arXiv preprint arXiv:2602.18397*, 2026.

[7] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn. Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv preprint arXiv:2304.13705*, 2023.

[8] K. Black, M. Y. Galliker, and S. Levine. Real-time execution of action chunking flow policies. *arXiv preprint arXiv:2506.07339*, 2025.

[9] J. Tang, Y. Sun, Y. Zhao, S. Yang, Y. Lin, Z. Zhang, J. Hou, Y. Lu, Z. Liu, and S. Han. Vlash: Real-time vlas via future-state-aware asynchronous inference. *arXiv preprint arXiv:2512.01031*, 2025.

[10] K. Black, A. Z. Ren, M. Equi, and S. Levine. Training-time action conditioning for efficient real-time chunking. *arXiv preprint arXiv:2512.05964*, 2025.

[11] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[12] J. Vial. Distributed real-time chunking, Mar 2026. URL https://jackvial.com/posts/distributed-real-time-chunking.html.

[13] C. Chi, Z. Xu, S. Feng, E. Cousineau, Y. Du, B. Burchfiel, R. Tedrake, and S. Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 44(10-11):1684–1704, 2025.

[14] T. T. Zhang, D. Pfrommer, C. Pan, N. Matni, and M. Simchowitz. Action chunking and exploratory data collection yield exponential improvements in behavior cloning for continuous control. *arXiv preprint arXiv:2507.09061*, 2025.

[15] B. Liu, Y. Zhu, C. Gao, Y. Feng, Q. Liu, Y. Zhu, and P. Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36:44776–44791, 2023.

[16] M. J. Kim, Y. Gao, T.-Y. Lin, Y.-C. Lin, Y. Ge, G. Lam, P. Liang, S. Song, M.-Y. Liu, C. Finn, et al. Cosmos policy: Fine-tuning video models for visuomotor control and planning. *arXiv preprint arXiv:2601.16163*, 2026.

[17] A. O'Neill, A. Rehman, A. Maddukuri, A. Gupta, A. Padalkar, A. Lee, A. Pooley, A. Gupta, A. Mandlekar, A. Jain, et al. Open x-embodiment: Robotic learning datasets and rt-x models: Open x-embodiment collaboration 0. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6892–6903. IEEE, 2024.

[18] A. Khazatsky, K. Pertsch, S. Nair, A. Balakrishna, S. Dasari, S. Karamcheti, S. Nasiriany, M. K. Srirama, L. Y. Chen, K. Ellis, et al. Droid: A large-scale in-the-wild robot manipulation dataset. *arXiv preprint arXiv:2403.12945*, 2024.

[19] S. Ye, Y. Ge, K. Zheng, S. Gao, S. Yu, G. Kurian, S. Indupuru, Y. L. Tan, C. Zhu, J. Xiang, et al. World action models are zero-shot policies. *arXiv preprint arXiv:2602.15922*, 2026.

[20] J. Kwok, C. Agia, R. Sinha, M. Foutter, S. Li, I. Stoica, A. Mirhoseini, and M. Pavone. Robomonkey: Scaling test-time sampling and verification for vision-language-action models. *arXiv preprint arXiv:2506.17811*, 2025.

[21] Y. Ma, Y. Zhou, Y. Yang, T. Wang, and H. Fan. Running vlas at real-time speed. *arXiv preprint arXiv:2510.26742*, 2025.

[22] S. H. Høeg, Y. Du, and O. Egeland. Streaming diffusion policy: Fast policy synthesis with variable noise diffusion models. *arXiv preprint arXiv:2406.04806*, 2024.

[23] L. Guo, Z. Xue, Z. Xu, and H. Xu. Demospeedup: Accelerating visuomotor policies via entropy-guided demonstration acceleration. In *Proceedings of The 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pages 599–609. PMLR, 2025.

[24] N. R. Arachchige, Z. Chen, W. Jung, W. C. Shin, R. Bansal, P. Barroso, Y. H. He, Y. C. Lin, B. Joffe, S. Kousik, et al. Sail: Faster-than-demonstration execution of imitation learning policies. *arXiv preprint arXiv:2506.11948*, 2025.

[25] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[26] D. Crankshaw, G.-E. Sela, C. Zumar, X. Mo, J. E. Gonzalez, I. Stoica, and A. Tumanov. Inferline: Ml prediction pipeline provisioning and management for tight latency objectives. *arXiv preprint arXiv:1812.01776*, 2018.

[27] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, 2020.

[28] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*, pages 521–538, 2022.

[29] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[30] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[31] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[32] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[33] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems*. Association for Computing Machinery, 2016.

[34] D. Narayanan, R. Rao, S. Kandula, A. Seshadri, S. Roberts, P. Chaudhary, J. Gu, J. Gonzalez, A. Harlap, A. Krishnamurthy, et al. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, 2020.

[35] Y. Dai, G. Ananthanarayanan, L. Cox, X. Foukas, B. Radunovic, and R. Netravali. Kairos: A scalable serving system for physical ai. *arXiv preprint arXiv:2605.11381*, 2026.

[36] W. Jiang, J. Clemons, R. O'Flaherty, H. Hadfield, A. Degirmenci, S. Song, Y. Narang, and C. Kozyrakis. Rosa: A robotics foundation model serving system for robot factories. *arXiv preprint arXiv:2607.01088*, 2026.

[37] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

A Appendix

Contents

A.1 Scheduler Pseudocode 14
A.2 Other GPUs 14
A.3 Additional robot foundation models 15
A.4 Choosing the right batch size 15
A.5 Lookahead L 16
A.6 Effect of network on cloud serving 16
A.7 Simulation system metrics 18
A.8 Real world per-tier metrics 19
A.9 Implementation Details 20

A.1 Scheduler Pseudocode

EDF and RR plan the next batch only when the GPU becomes available. LA, in contrast, searches continuously while the GPU is busy and dispatches its best batch when the GPU becomes available.

Earliest Deadline First. EDF greedily fills the batch with the candidates closest to starvation.

```
def edf(robots: list[Robot]) -> list[Robot]:
  sorted_by_deadline = sorted(robots, key=lambda robot: robot.deadline)
  return sorted_by_deadline[:min(max_batch_size, len(robots))]
```

`robot.deadline` is computed from the mirror as the wall-clock time of the last control step covered by any chunk in $\mathcal{Q}_j$.

Round Robin. RR cycles through all robots, advancing a pointer across epochs.

```
i = 0
def round_robin(robots: list[Robot]) -> list[Robot]:
  batch = []
  for _ in range(min(max_batch_size, len(robots))):
      batch.append(robots[i])
      i = (i + 1) % len(robots)
  return batch
```

Lookahead. LA simulates forward and chooses the first batch from the best scoring schedule.

```
def lookahead(robots: list[Robot]) -> list[Robot]:
    best_schedule, best_score = [], 0
    # (schedule, simulated state)
    frontier = deque([([], robots)])

    while frontier and has_slack():
        schedule, state = frontier.popleft()
        for batch in candidate_batches(state):
            next_state = simulate(state, batch)
            new_schedule = schedule + [batch]
            score = score_schedule(robots, next_state, new_schedule)
            if score > best_score:
                best_schedule, best_score = new_schedule, score
            if len(new_schedule) < max_depth:
                frontier.append((new_schedule, next_state))

    return best_schedule[0]
```

`has_slack` is true while the current time is more than a few milliseconds before the GPU's next anticipated dispatch time; `candidate_batches(s)` enumerates batches of size $1, \ldots, B_{\max}$; `simulate(s, b)` uses the mirror to roll the MDP transition of §3.1 to the next epoch; and `score_schedule` calculates Score(S) using the mirror.

In our implementation, `candidate_batches(s)` heuristically prunes batches to reduce the search space and `score_schedule` calculates the reward as the total executed robot time inside a 1-second window starting from the schedule's starting time.

A.2 Other GPUs

We cloud-serve all of our experiments on an L40S datacenter GPU, but it is possible to serve on all other GPU types as well. In this section, we do some preliminary analysis on inference latency between an L40S and an H100. Table 1 details the inference latencies as a function of batch size on both GPUs with the $\pi_{0.5}$ model. We note that because $\pi_{0.5}$ leans more compute-bound than memory-bound, an H100 is able to considerably reduce inference latency and allow for smaller deltas between consecutive batch sizes. In practice, the schedulers will be able to schedule larger batches at lower inference cost, supporting larger numbers of robots.

Table 1: $\pi_{0.5}$ inference latency (ms) as a function of batch size on L40S and H100 GPUs.

Batch size	L40S (ms)	H100 (ms)
1	73.0	42.3
2	109.8	56.0
3	142.4	66.1
4	177.5	79.3
5	211.1	87.2

A.3 Additional robot foundation models

Armory is not limited to $\pi_{0.5}$ as the only model choice; it is possible to configure backends for other popular VLA models, such as NVIDIA's GR00T-N1 [5]. In this section, we do some preliminary analysis on the inference profile of GR00T. Table 2 details the inference profiles as a function of batch size. We see that GR00T is more memory-bound than $\pi_{0.5}$, allowing for smaller deltas between inference times for consecutive batch sizes. Similar to the insight from Section A.2, the schedulers will be able to support larger numbers of robots by scheduling larger batches at lower inference cost.

Table 2: Inference latency (ms) as a function of batch size on $\pi_{0.5}$ and GR00T N1.7 on an L40S GPU.

Batch size	$\pi_{0.5}$	GR00T N1.7
1	73.0	71.8
2	109.8	76.5
3	142.4	83.5
4	177.5	91.8
5	211.1	100.9

A.4 Choosing the right batch size

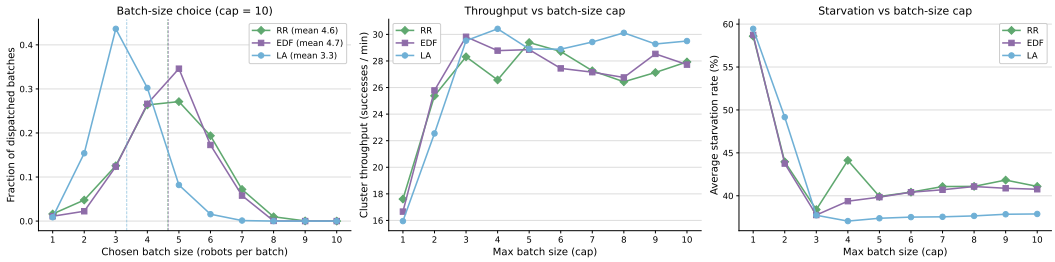


Figure 7: How does the max batch size affect scheduler performance? We ablate this on the all-fast scenario. Lookahead performs most effectively by choosing smaller, more efficient batches, keeping starvation lower as the batch-size cap grows.

In our work, we posit that naively using the maximum batch size is not always the best decision, and that many scenarios require dynamic batching. Lookahead can take a maximum batch size cap and choose the best batch of robots within that cap to serve, while more naive heuristic schedulers like EDF and Round-Robin batch as many robots together as they can at each decision step.

The reason this matters is that batching is a tradeoff. Larger batches amortize inference across more robots, but inference latency grows with batch size (Table 2): on an L40S, a batch of 5 takes 211ms versus 73ms for a single robot, while a batch of 3 takes only 142ms and already captures most of the per-robot amortization. A scheduler that always fills the batch therefore serves more robots per call but makes every call slower and less frequent, so robots left out of the current batch wait longer and starve. Choosing the right batch size means balancing this amortization against the latency penalty.

We ablate over the maximum batch size cap so as to give the naive methods the best shot at efficient scheduling. We run this study on the all-fast scenario because homogeneous deadlines are the most batching-friendly setting: it is where fixed max-batching is least penalized, making it the strongest case for the naive heuristics.

Figure 7 shows the result. The left panel reports the distribution of chosen batch sizes at a cap of 10: EDF and Round-Robin gravitate to large batches (mean 4.7 robots per call), whereas Lookahead prefers smaller, more efficient batches (mean 3.3, median 3). The center and right panels show that at small caps, all schedulers are forced into small batches and behave similarly; as the cap grows, the naive schedulers exploit it by over-batching, and their starvation rate climbs. Lookahead is largely insensitive to the cap (because it already selects efficient batches, raising the cap does not tempt it into over-batching), holding starvation and throughput flat. This amounts to Lookahead delivering lower starvation and higher throughput than the naive methods.

We found that in simulation, with our setup of 10 robots, $b = 3$ is the best parameter that gives the naive schedulers their strongest configuration while keeping per-call latency low enough to avoid widespread starvation, and we use it as the batch-size cap in all main experiments. In the real world, we found $b = 3$ to be best for all-fast, and $b = 5$ to be best for one-fast and half-fast.

A.5 Lookahead L

For Lookahead, the L parameter determines the length of the schedules (in epochs) the scheduler considers in its search. We originally hypothesized that planning for longer would improve scheduler performance, but we found that this is not the case for our implementation. In Table 3, we ablate $L = 1, 2, 3$ across our experimental setups in LIBERO and find that increasing L slightly increases average starvation. We note that these results are specific to our implementation, and that changes to the search algorithm or reward function may show different results.

Table 3: Average starvation (%) slightly increases across lookahead depths $L \in \{1, 2, 3\}$.

		LA@1			**LA@3**			**LA@5**		
Config	N	L=1	L=2	L=3	L=1	L=2	L=3	L=1	L=2	L=3
All Fast	**2**	1.0	1.1	1.0	1.0	1.2	1.0	1.0	1.0	1.0
	4	11.0	11.1	10.7	10.9	11.2	11.0	10.9	10.8	10.7
	6	22.5	22.9	23.1	22.9	22.7	23.1	22.9	22.7	23.1
	8	30.4	30.7	31.3	30.6	30.7	31.4	30.5	30.9	31.3
	10	36.6	37.3	38.1	36.7	37.2	38.3	36.6	37.1	37.8
One Fast	**2**	1.0	1.0	1.1	1.0	1.0	1.0	1.1	0.9	1.1
	4	4.5	4.3	4.3	3.1	3.2	3.2	3.0	3.1	3.0
	6	6.4	6.3	6.4	6.3	6.8	7.3	9.0	9.3	10.0
	8	11.6	12.0	12.3	12.5	12.8	15.0	14.5	15.6	17.7
	10	17.6	18.1	18.4	18.5	19.1	21.0	21.4	22.7	24.2
Half Fast	**2**	1.0	1.0	1.1	1.1	1.0	1.1	1.0	1.1	1.0
	4	6.8	7.0	6.7	6.7	7.2	6.8	19.5	17.2	17.3
	6	13.7	13.6	13.9	14.9	15.5	16.6	22.2	22.7	23.8
	8	20.4	20.9	21.1	22.4	23.7	24.7	33.5	35.6	36.9
	10	27.2	27.4	27.6	29.1	31.3	31.4	47.4	48.9	48.9

A.6 Effect of network on cloud serving

A natural concern with cloud-based robot policy serving is whether network latency and jitter make the approach impractical. We ablate both factors in simulation with the all-fast, one-fast, half-fast configurations with $b = 3$.

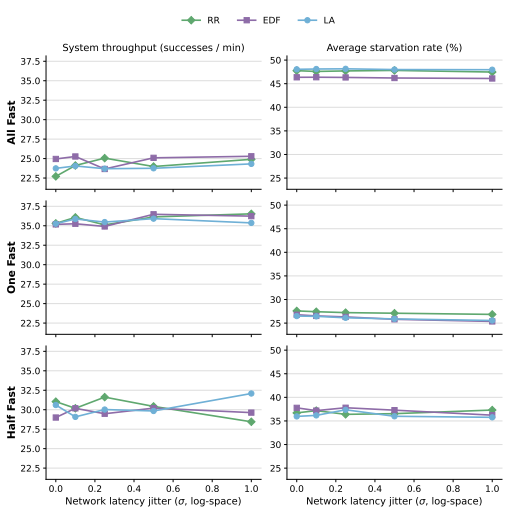
Modeling network deviation. We model each one-way link delay (the observation uplink d_{obs} and the action downlink d_{action}) as an independent draw from a log-normal distribution. For a link with target median delay m (ms) and jitter parameter σ, we sample

$$d \sim \text{Lognormal}(\mu,\, \sigma^2), \qquad \mu = \ln m, \tag{14}$$

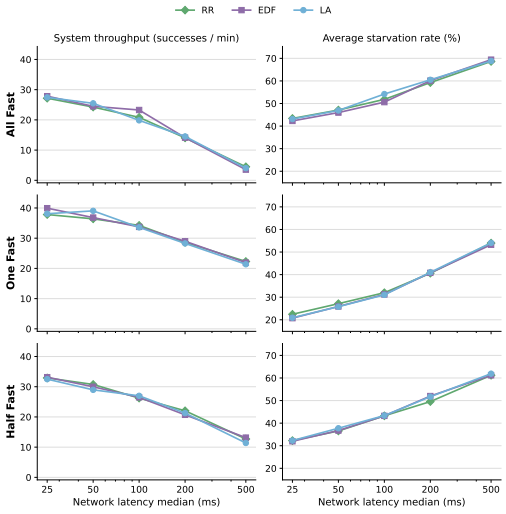
so that $\ln d \sim \mathcal{N}(\ln m,\, \sigma^2)$. We parameterize by the *median* rather than the mean because the median is invariant to the jitter: $\text{median}(d) = e^{\mu} = m$ for all σ, allowing us to sweep jitter while holding the typical latency fixed. A log-normal (rather than Gaussian) captures the heavy right tail characteristic of wide-area networks, where delays are strictly non-negative, most packets arrive near or below the median, and occasional congestion produces spikes.

Increasing σ widens this tail asymmetrically while leaving the median fixed. Setting $\sigma = 0$ recovers a deterministic delay of exactly m. We draw a fresh sample for every request, so each robot sees new uplink and downlink delays at every inference step.

Network jitter. In Figure 8a we study the effect of network jitter on cloud serving. We sweep the standard deviation σ of a log-normal jitter distribution applied to both d_{obs} and d_{action}, while setting each one-way median to *50ms* (standard US coast-to-coast transmission delay). Across the full range of jitter magnitudes, starvation rate and system throughput remain nearly flat for all three schedulers. This stability arises because Armory's EMA-based delay estimator absorbs moderate jitter, and the action queue provides a natural buffer against individual delayed packets. The result indicates that our cloud serving system is robust even under high network variance.



(a) Effect of network jitter variance ($m = 50$ms).



(b) Effect of median network latency ($\sigma = 0$).

Figure 8: Network ablation study showing how network delay affects cloud serving for $b = 3$.

Network median delay. In Figure 8b we study the effect of median network latency on cloud serving, sweeping the median delay applied to both d_{obs} and d_{action} from 25ms to 500ms with zero jitter. In contrast to the jitter sweep, median latency has a direct and monotonic effect: as latency grows, system throughput decreases and starvation rate increases for all three schedulers and across all robot configurations. The degradation is graceful at the latencies encountered in real deployments. We can see that performance remains high through 50-100ms, spanning typical intra-continental round trips. All three schedulers degrade in the same manner, indicating that generally, scheduling policy cannot compensate for raw transport delay. These results confirm that cloud serving remains practical across the latency regimes of real wide-area deployments, with performance degrading predictably only as latency approaches the extremes.

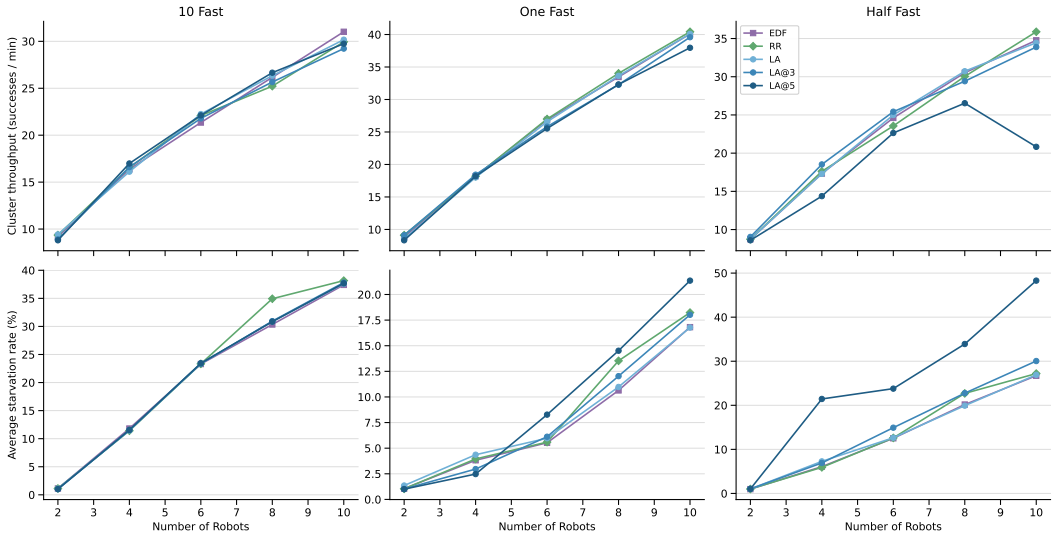


Figure 9: System throughput (successes/min) and average starvation as the cluster scales from N=2 to 10 robots, with batch-size cap b=3.

A.7 Simulation system metrics

System metrics. Figure 9 and Table 4 report system throughput and average starvation as we scale the cluster from N=2 to 10 robots across the three robot configurations. Throughput grows with N while starvation rises as more robots contend for the shared server, and at b=3 all schedulers achieve comparable *total* throughput and starvation. The core contribution of Lookahead is not in raising aggregate throughput, but in reallocating it across tiers, giving fast and slow robots a more favorable share of the system's throughput (Section 4.2).

Table 4: LIBERO: System throughput (successes/min) and average starvation (%, gray), max batch size = 3.

Config	N	RR	EDF	LA w=1	LA w=3	LA w=5
All Fast	**2**	9.36 (1.1)	9.20 (1.1)	9.42 (1.1)	8.90 (1.0)	8.80 (1.0)
	4	16.55 (11.4)	16.37 (11.8)	16.11 (11.5)	16.60 (11.5)	16.99 (11.6)
	6	22.08 (23.3)	21.35 (23.3)	22.25 (23.4)	21.79 (23.4)	22.11 (23.4)
	8	25.22 (34.9)	26.17 (30.3)	26.34 (30.9)	25.67 (30.9)	26.66 (30.8)
	10	29.94 (38.2)	31.02 (37.4)	30.18 (37.6)	29.23 (37.8)	29.75 (37.7)
One Fast	**2**	9.12 (1.0)	8.83 (1.1)	8.56 (1.4)	9.11 (1.0)	8.33 (1.0)
	4	18.19 (3.9)	18.19 (3.8)	18.00 (4.4)	18.42 (3.0)	18.19 (2.5)
	6	26.99 (5.6)	26.87 (5.5)	26.57 (6.0)	25.82 (6.1)	25.54 (8.3)
	8	34.02 (13.5)	33.47 (10.6)	33.64 (11.0)	32.28 (12.0)	32.31 (14.5)
	10	40.42 (18.2)	40.10 (16.8)	40.09 (16.8)	39.59 (18.0)	37.96 (21.4)
Half Fast	**2**	8.74 (1.0)	8.73 (1.0)	8.75 (1.0)	9.04 (1.0)	8.60 (1.0)
	4	17.62 (5.9)	17.33 (6.1)	17.29 (7.3)	18.52 (6.9)	14.38 (21.4)
	6	23.56 (12.6)	24.61 (12.5)	25.00 (12.6)	25.42 (14.9)	22.64 (23.8)
	8	30.04 (22.7)	30.52 (20.2)	30.72 (19.9)	29.41 (22.8)	26.54 (33.9)
	10	35.88 (27.2)	34.76 (26.7)	34.45 (26.9)	33.89 (30.0)	20.83 (48.3)

A.8 Real world per-tier metrics

In Table 5, we list the throughputs and starvations from our real-world experiments. In many cases, we find that lower starvation correlates with higher throughputs, matching the results in Figure 3b. However, there are some cases where the Lookahead scheduler has both higher starvations *and* throughputs. This can be clearly seen in the **All Fast** configuration, where LA's throughput increases by more than 7% despite a 5.71 percentage-point increase in starvation when compared to EDF.

We suggest a few potential sources for this discrepancy. First, our metrics may be noisy due to small sample size. Setting up tasks for 10 robots in the real-world is very time-consuming, so we are only able to run 1-minute rollouts for 3 seeds. Second, real-world evaluations are also noisy. While we made our best effort at standardizing task and robot initializations, runs may still differ in their policy predictions as well as the exact scheduling decisions.

Lastly, our choice of *naive async* as our action chunking strategy may confound with increases in starvation. Prior works have noted that naive async creates discontinuous/jerky motions when execution starts from the middle of a chunk. Intuitively, this is because chunks predicted with naive async have no awareness of actions executed from previous chunks.

In Figure 10, we plot the distribution of the first executed indexes of chunks across the real-world experiments. The Lookahead schedulers bias towards scheduling a large proportion of chunks to start execution from the start of the chunk (first executed index 0), whereas other schedulers start execution from the middle. We hypothesize that this unintentionally helps robots under Lookahead by reducing the chances for naive async to create jerky motions. Predicting chunks with more sophisticated asynchronous strategies such as RTC [8] or VLASH [9] may lead to different results.

Table 5: Real-world: per-tier throughput (successes / min) and starvation rate (%) across scenarios.

		Throughput (successes/min)			**Starvation (%)**		
Config	**Scheduler**	**fast**	**slow**	**total**	**avg**	**fast**	**slow**
All Fast	**EDF**	6.27	–	62.67	35.94	35.94	–
	RR	6.00	–	60.00	44.46	44.46	–
	LA w=1	6.73	–	67.33	41.65	41.65	–
One Fast	**EDF**	3.67	8.81	83.00	13.37	43.27	10.05
	RR	2.00	9.04	83.33	12.90	43.31	9.52
	LA w=1	3.67	9.85	92.33	12.37	42.68	9.01
	LA w=5	8.33	10.04	98.67	15.18	13.10	15.41
Half Fast	**EDF**	3.13	9.40	62.67	24.93	42.09	7.78
	RR	3.87	9.07	64.67	24.66	41.26	8.05
	LA w=1	3.60	9.67	66.33	25.55	42.69	8.41
	LA w=5	6.13	6.53	63.33	39.42	32.72	46.13

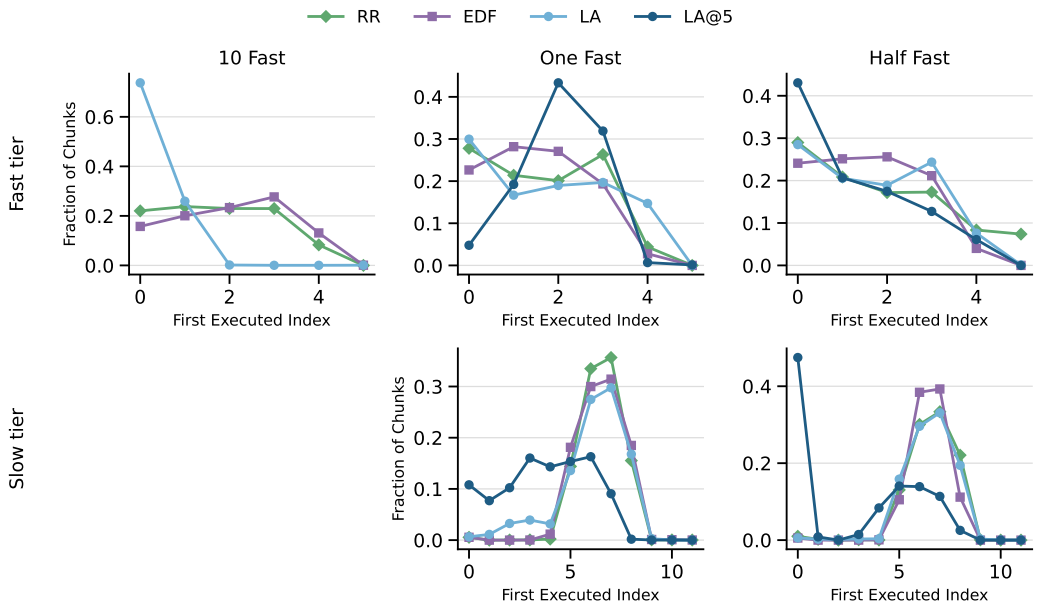


Figure 10: Lookahead schedulers bias towards scheduling chunks so that they are executed starting at index 0.

A.9 Implementation Details

Following standard practice in LLM serving engines, Armory is implemented as a collection of Python processes that communicate over shared memory.

The **frontend** process is responsible for communicating with robots over network. Robots initiate a WebSocket connection with the frontend and send the captured observation to the server on each control step. The frontend writes the raw data (images, prompt, metadata) to shared memory and sends the metadata to the scheduler and engine process.

The **scheduler** process is responsible for maintaining the software mirror of the robots and scheduling the next batches for the server to execute. The mirror tracks all the robots as well as all the chunks. For EDF and RR, the scheduler schedules the next batch when the GPU becomes available, while the Lookahead scheduler continually searches and queues the best schedule whenever the GPU becomes available again.

The **engine** process is a worker that is solely responsible for running model forward passes. On startup, it profiles d_{infer} for each batch size and sends it to the scheduler. Afterwards, it busy-waits until a batch arrives from the scheduler. For each batch it consumes, it directly reads the latest request in shared memory. If the latest request is identical to the last served request, or the robot has not executed $H_{\min}$ actions since the last served request, the request will be dropped from the batch. For the $\pi_{0.5}$ forward pass, Armory uses the official JAX implementation [4].