Action Chunk Scheduling for Batched Robot Policy Serving

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

***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 a fleet of 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://actionchunkscheduling.github.io.**

I. Introduction

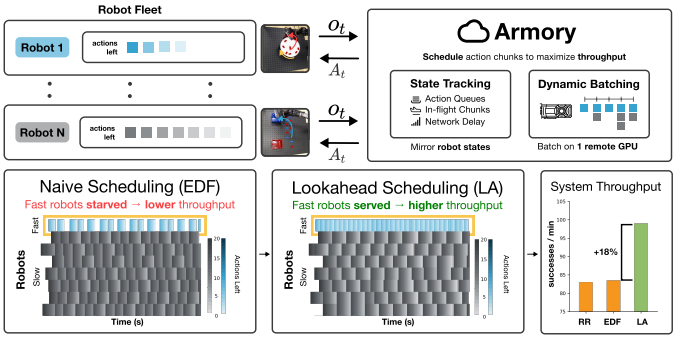


Fig. 1: Armory is an end-to-end serving system for deploying robot policies from the cloud that tracks robot states and schedules action chunks to minimize robot starvation. Naive scheduling underserves fast robots completing highly dynamic tasks, so we propose Lookahead scheduling to be able to perform a trade off for fast robot service.

Robot policies are becoming increasingly capable [37, 18, 4, 14, 3], and deploying them at scale turns policy inference into a *policy serving problem*. Local inference avoids network delay, yet it is constrained by power, thermals, payload, and cost, and large VLA policies can be difficult to run in real-time on local hardware [16]. Cloud serving is therefore a natural alternative: a shared GPU can host larger policies and amortize computation across many robots. In this paper, we study the question: *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 [34]; recent real-time VLA systems build on this idea [5, 28, 6], but primarily address single-robot serving, where the GPU may remain underutilized between requests. The core difficulty is that batching improves GPU throughput at the cost of robot responsiveness: 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, where dynamic tasks require short execution horizons and frequent updates, while quasi-static tasks may tolerate longer horizons.

Our key insight is that batched robot policy serving should be formulated as a *closed-loop scheduling problem*. We instantiate this idea in **Armory**, a serving system for batched robot policy inference, formulating 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 scheduler that simulates candidate batches forward and selects the batch with the best predicted reward normalized by inference time. We evaluate Armory on a fleet of 10 real robots, finding that simple schedulers are competitive in homogeneous fleets, but heterogeneity-aware scheduling is needed when robots have different reactivity requirements. Lookahead scheduling improves overall system throughput by up to 18% in heterogeneous settings while exposing a controllable tradeoff between serving fast and slow robots.

II. 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 predefined control frequency f_c, capturing o_t and executing a_t. To address gaps between executed chunks that appear as synchronous pauses, we adopt *async execution* [5, 28], where the inference for the next chunk starts while the current chunk is still executing. To ensure continuity, executed action indices must strictly increment [30], and we decouple action and observation indices to facilitate cloud streaming (Figure 2).

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 $[H_{\min}, H_{\max}]$. If $H_{\min}$

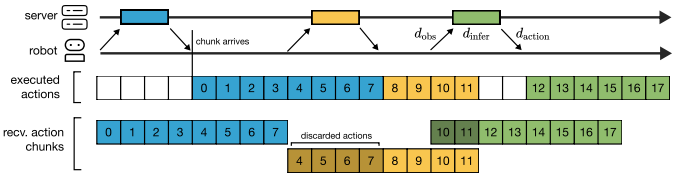


Fig. 2: **Execution Timeline.** A robot with $H_{\max} = 8$ consumes actions by incrementing the action index, pausing when it is starved. When starvation occurs while a chunk is still in inference/network, the chunk's execution will extend more than $H_{\max}$ steps after the observation was sent.

is too small, frequent switching between chunks increases compounding errors [7, 33], and if it is too large, robots lose reactivity. Empirically, we find the optimal interval is task-dependent (Figure 3a), motivating the need to serve workloads with heterogeneous execution horizons.

Delay. 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 delay d_{action}.

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 III.

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. Our system uses *dynamic batching* to carefully balance the tradeoff between throughput and latency.

III. Scheduling for Batched Robot Policy Serving

Armory treats batched policy serving as a control-aware scheduling problem: chunk execution depends on when chunks are generated by the server and received by the robot, dynamics that Armory models with a server-side software mirror.

A. 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 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. Armory keeps a server-side mirror of robot execution. The state is

$$s_k = \big(t_k, \{i_j, \hat{i}_j, \mathcal{Q}_j\}_{j=1}^{N}\big), \tag{1}$$

where t_k is the wall-clock time at the start of the epoch, i_j and $\hat{i}_j$ are the action indices of the latest executed robot step and the latest robot step received by the server, and $\mathcal{Q}_j$ is robot j's queue of generated action chunks (including chunks generated by the server but not yet arrived at the robot). Each chunk is $c = (i_{\text{start}}, h_j, t_{\text{arr}})$: the action index of its first action, the execution horizon, and the time it arrives at the robot.

Transition. The inference latency $\tilde{d}_{\text{infer}(b)}$ (a function of batch size b) and the per-robot observation and action latencies $\tilde{d}_{\text{obs},j}, \tilde{d}_{\text{action},j}$ are stochastic. Given s_k and B_k, inference completes at $t_{k+1} = t_k + \tilde{d}_{\text{infer}(|B_k|)}$, and each served robot $j \in B_k$ appends a chunk $c^j_{\text{new}} = (\hat{i}_j, h_j, t_{k+1} + \tilde{d}_{\text{action},j})$ to $\mathcal{Q}_j$. Meanwhile every robot ticks at control frequency f_c: at each tick $\tau \in [t_k, t_{k+1}]$, robot j advances if some arrived chunk covers its current index, and otherwise starves,

$$i_j \leftarrow i_j + 1 \quad \text{if } \exists\, c \in \mathcal{Q}_j : \; t_{\text{arr}} \le \tau \wedge i_j \in [i_c, i_c + h_c). \tag{2}$$

The server also refreshes $\hat{i}_j \leftarrow \max\{i : t_i + \tilde{d}_{\text{obs},j} \le t_{k+1}\}$, where t_i is the wall-clock time the robot reached index i.

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

$$R(s_k, B_k) = \frac{1}{f_c} \sum_{j=1}^{N} w_j \, \Delta i_j(s_k, B_k), \tag{3}$$

where w_j is an optional task-dependent weight: $w_j = 1$ treats all executed robot time equally, while larger w_j prioritize robots that need fresher actions, such as dynamic tasks with shorter execution horizons. A scheduling policy π maps states to batches, and the objective is

$$\pi^\star = \arg\max_{\pi} \sum_{k} \mathbb{E}\big[R(s_k, B_k)\big]. \tag{4}$$

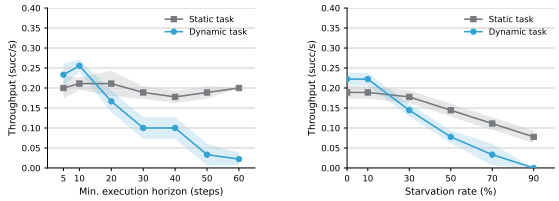
B. Scheduling

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 $S = (s_0, B_0), (s_1, B_1), \ldots, (s_{L-1}, B_{L-1})$ is a sequence of states and actions. 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:

$$\text{Score}(S) = \sum_{\ell=0}^{L-1} \frac{R(s_\ell, B_\ell)}{d_{\text{infer}}(|B_\ell|)}. \tag{5}$$

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



(a) Execution horizon. (b) Starvation rate.

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

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.

C. Armory

We implement Armory, an end-to-end serving engine built around the software mirror of III-A, using a push-based architecture (each robot sends o_t before executing a_t) to keep the mirror accurate; d_{infer} profiling, online estimation of $d_{\text{obs}}, d_{\text{action}}$, and more are detailed in §A9.

IV. Experiments

We evaluate our schedulers in real-world experiments under varying robot heterogeneity, using $\pi_{0.5}$ [14] as a representative robotics foundation model.

A. Setup

Workload and Heterogeneity. Robots differ in how reactive they must be, which we capture with two classes that differ in execution horizon. A *fast* robot runs a *dynamic* task with a shorter H_{max}, since performance degrades quickly when chunks go stale while manipulating moving objects; a *slow* robot runs a *static*, quasi-static task that tolerates longer horizons (Figure 3a). Because dynamic tasks are sensitive to loss of reactivity, starvation also hurts fast robots disproportionately (Figure 3b). Here *fast*/*slow* refer to how quickly a robot exhausts its chunks given its horizon (and thus how reactive it is), **not** its movement/step speed. We vary heterogeneity across three configurations: *one-fast* (one fast robot, rest slow), *half-fast* (half fast, half slow), and *all-fast* (all fast, none slow), which span the spectrum and stress the scheduler in distinct ways. *All-fast* is homogeneous, isolating fleet-size effects from heterogeneity. *Half-fast* maximizes scheduling tension: fast and slow robots compete equally for batch slots, so any bias toward one class visibly starves the other. *One-fast* tests whether the scheduler can protect a single sensitive robot among many less urgent ones—e.g., a robot that turns highly dynamic for a short period (striking a match) before resuming slower operation.

Real-world setup. We serve 10 AgileX PiPER arms (top and wrist Intel RealSense D435i cameras) from a single colocated L40S GPU (round-trip delay ∼5–10 ms; robustness to delay and jitter in §A6), using a $\pi_{0.5}$ checkpoint finetuned at $H = 20$, $f_c = 30$. Both classes use $H_{\text{min}} = 5$, with $H_{\text{max}} = 10$ for fast and 20 for slow robots. We fine-tune two tasks, static brick sorting (colored bricks into bins) and dynamic turntable sorting (moving bricks off a turntable into a mug), running the turntable task on fast robots and bin sorting on slow robots, with 3 seeds per configuration. Due to resource constraints, all robots do brick sorting with $H_{\text{max}} = 10$ for *all-fast*.

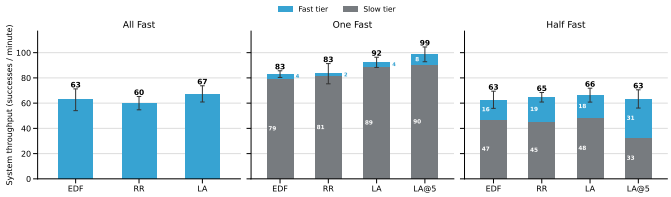


Fig. 4: Real-world system throughput in legos-per-minute across the three scenarios. LA@5 significantly improves fast-tier throughput in the one-fast case by 2.27× over EDF, and also provides a boost to slow-tier to increase system throughput by approximately 18% over baselines. In the half-fast case, we can trade off towards fast robots while maintaining system throughput.

Schedulers. We compare Round-Robin, Earliest-Deadline-First, and Lookahead (III-B) at the best-performing batch size per baseline scheduler ($b = 3$ homogeneous, $b = 5$ heterogeneous; sweep in §A4). For Lookahead we fix $w_j = 1$ for slow robots and vary $w_j \in \{1, 3, 5\}$ for fast robots (LA@w_j): larger w more aggressively prioritizes short-horizon, starvation-sensitive robots at the expense of slow-tier service. Although we permit multi-step planning, ($L = 1$) suffices for heterogeneity-aware scheduling (§A5).

Metrics. Our two metrics are *starvation rate* (the fraction of control steps where a robot has no action to execute) and *throughput* (successes per unit time: Lego pieces sorted per 60-second window).

B. Evaluations

Weighted scheduling exposes a controllable throughput tradeoff. On the ten-arm fleet (Figure 4), all methods perform comparably in the all-fast (homogeneous) setting.

When fast and slow robots share a GPU, Lookahead provides a control surface for trading average service against protecting robots that exhaust chunks faster: at $w = 5$ it boosts one-fast fast-tier throughput by 4× over RR and 2× over EDF while preserving slow-tier throughput, and in half-fast it shifts service toward fast robots at the cost of slow-tier. The benefit of Lookahead is thus not universal dominance over fixed baselines, but the ability to select an operating point on the throughput/starvation tradeoff, and in some cases (one-fast) we gain throughput for free.

V. Conclusion

We introduced multi-tenant robot policy serving as a closed-loop scheduling problem, formulating the server's batching decisions as an MDP over action queues, execution horizons, and communication delays. Our real-world experiments show that the cost of serving latency is task-dependent, that simple schedulers suffice for homogeneous fleets, and that under heterogeneous reactivity demands our Lookahead scheduler improves system throughput by up to 18% by selectively protecting latency-sensitive robots, with tunable weights for operator control.

VI. Limitations

In recent years, many works have proposed inference optimizations for large transformer models. Notable techniques include continuous batching [32], PagedAttention [21], chunked prefill [1], and prefill-decode disaggregation [36, 27]. 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.

Also, our implementation still has significant room for improvement. While the Lookahead scheduler can improve overall system throughput in some situations, it does not always outperform the baseline schedulers. Future work may develop schedulers that are optimal across all workloads.

VII. AI Usage Statement

All writing was done by humans. We minimally used AI (Claude/ChatGPT) to do spelling/grammar/syntax checks. Claude/Codex were cautiously used with heavy human input and postprocessing to help with the generation of some plots/visualizations.

VIII. Acknowledgements

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

References

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[2] Nadun Ranawaka Arachchige, Zhenyang Chen, Wonsuhk Jung, Woo Chul Shin, Rohan Bansal, Pierre Barroso, Yu Hang He, Yingyang Celine Lin, Benjamin Joffe, Shreyas Kousik, et al. Sail: Faster-than-demonstration execution of imitation learning policies. *arXiv preprint arXiv:2506.11948*, 2025.

[3] Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, et al. Gr00t n1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.

[4] Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. pi_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

[5] Kevin Black, Manuel Y Galliker, and Sergey Levine. Real-time execution of action chunking flow policies. *arXiv preprint arXiv:2506.07339*, 2025.

[6] Kevin Black, Allen Z Ren, Michael Equi, and Sergey Levine. Training-time action conditioning for efficient real-time chunking. *arXiv preprint arXiv:2512.05964*, 2025.

[7] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 44(10-11):1684–1704, 2025.

[8] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[9] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: Ml prediction pipeline provisioning and management for tight latency objectives. *arXiv preprint arXiv:1812.01776*, 2018.

[10] Yinwei Dai, Ganesh Ananthanarayanan, Landon Cox, Xenofon Foukas, Bozidar Radunovic, and Ravi Netravali. Kairos: A scalable serving system for physical ai. *arXiv preprint arXiv:2605.11381*, 2026.

[11] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan 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.

[12] Lingxiao Guo, Zhengrong Xue, Zijing Xu, and Huazhe 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.

[13] Sigmund H Høeg, Yilun Du, and Olav Egeland. Streaming diffusion policy: Fast policy synthesis with variable noise diffusion models. *arXiv preprint arXiv:2406.04806*, 2024.

[14] Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, et al. $pi_{0.5}$: A vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.

[15] Wenqi Jiang, Jason Clemons, Rowland O'Flaherty, Hugo Hadfield, Alperen Degirmenci, Shuran Song, Yashraj Narang, and Christos Kozyrakis. Rosa: A robotics foundation model serving system for robot factories. *arXiv preprint arXiv:2607.01088*, 2026.

[16] Wenqi Jiang, Jason Clemons, Karu Sankaralingam, and Christos Kozyrakis. How fast can i run my vla? demystifying vla inference performance with vla-perf. *arXiv preprint arXiv:2602.18397*, 2026.

[17] Alexander Khazatsky, Karl Pertsch, Suraj Nair, Ashwin Balakrishna, Sudeep Dasari, Siddharth Karamcheti, Soroush Nasiriany, Mohan Kumar Srirama, Lawrence Yunliang Chen, Kirsty Ellis, et al. Droid: A large-scale in-the-wild robot manipulation dataset. *arXiv*

preprint arXiv:2403.12945, 2024.

[18] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

[19] Moo Jin Kim, Yihuai Gao, Tsung-Yi Lin, Yen-Chen Lin, Yunhao Ge, Grace Lam, Percy Liang, Shuran Song, Ming-Yu Liu, Chelsea Finn, et al. Cosmos policy: Fine-tuning video models for visuomotor control and planning. *arXiv preprint arXiv:2601.16163*, 2026.

[20] Jacky Kwok, Christopher Agia, Rohan Sinha, Matt Foutter, Shulu Li, Ion Stoica, Azalia Mirhoseini, and Marco Pavone. Robomonkey: Scaling test-time sampling and verification for vision-language-action models. *arXiv preprint arXiv:2506.17811*, 2025.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion 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.

[22] Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36:44776–44791, 2023.

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

[24] Yunchao Ma, Yizhuang Zhou, Yunhuan Yang, Tiancai Wang, and Haoqiang Fan. Running vlas at real-time speed. *arXiv preprint arXiv:2510.26742*, 2025.

[25] Deepak Narayanan, Ratul Rao, Srikanth Kandula, Aakanksha Seshadri, Sasha Roberts, Parul Chaudhary, Jason Gu, Joseph Gonzalez, Aaron Harlap, Arvind 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.

[26] Abby O'Neill, Abdul Rehman, Abhiram Maddukuri, Abhishek Gupta, Abhishek Padalkar, Abraham Lee, Acorn Pooley, Agrim Gupta, Ajay Mandlekar, Ajinkya 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.

[27] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo 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.

[28] Jiaming Tang, Yufei Sun, Yilong Zhao, Shang Yang, Yujun Lin, Zhuoyang Zhang, James Hou, Yao Lu, Zhijian Liu, and Song Han. Vlash: Real-time vlas via future-state-aware asynchronous inference. *arXiv preprint arXiv:2512.01031*, 2025.

[29] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory 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.

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

[31] Seonghyeon Ye, Yunhao Ge, Kaiyuan Zheng, Shenyuan Gao, Sihyun Yu, George Kurian, Suneel Indupuru, You Liang Tan, Chuning Zhu, Jiannan Xiang, et al. World action models are zero-shot policies. *arXiv preprint arXiv:2602.15922*, 2026.

[32] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon 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.

[33] Thomas T Zhang, Daniel Pfrommer, Chaoyi Pan, Nikolai Matni, and Max Simchowitz. Action chunking and exploratory data collection yield exponential improvements in behavior cloning for continuous control. *arXiv preprint arXiv:2507.09061*, 2025.

[34] Tony Z Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv preprint arXiv:2304.13705*, 2023.

[35] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[36] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao 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.

[37] Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan 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.

APPENDIX

Contents

A Appendix 6
- A1 Related Work 6
- A2 Other GPUs 6
- A3 Additional robot foundation models 6
- A4 Choosing the right batch size 6
- A5 Lookahead L 7
- A6 Effect of network on cloud serving 7
- A7 Simulation metrics 8
- A8 Real world per-tier metrics . 8
- A9 Implementation Details . . . 9

A. Appendix

1) Related Work: **Robot Policy Inference.** Robot foundation models are rapidly scaling: early VLA systems [37] showed that large VLM backbones could be adapted to robotic control, and more recent models pair even larger backbones with broader cross-embodiment datasets [18, 4, 14, 19, 3, 26, 17]. This trend intensifies as the field expands to video prediction, with state-of-the-art models taking up to 7 seconds per inference [31]. On the systems side, [16] profiles the bottlenecks of VLA inference, [20] performs inference-time verification, and [24] pushes latency down through kernel-level optimization, while [13] 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 [5, 6], reasoning over future state during async inference [28], accelerating rollout [12], or combining action scheduling with controller tuning [2]. 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. More recently, Kairos [10] and ROSA [15] have proposed systems for serving multiple robots, but make different assumptions about the serving problem such as adaptive execution horizons and multi-GPU pools.

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 [8, 9, 11]. 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 [32, 21, 35, 1, 36]. 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 [23]. In adjacent systems settings, [29] introduces plan-ahead rescheduling for deadline-aware jobs in dynamic heterogeneous clusters, while [25] 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.

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 I 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 I: $\pi_{0.5}$ 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

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-N1.7 [3]. In this section, we do some preliminary analysis on the inference profile of GR00T. Table II 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 A2, the schedulers will be able to support larger numbers of robots by scheduling larger batches at lower inference cost.

TABLE II: 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

4) Choosing the right batch size: In our work, we posit that naively using the maximum batch size is not always the best

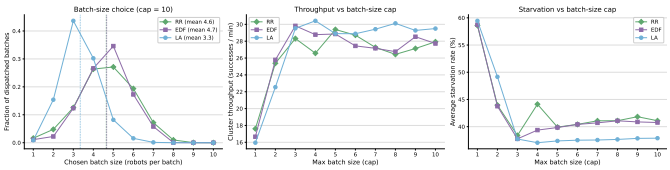
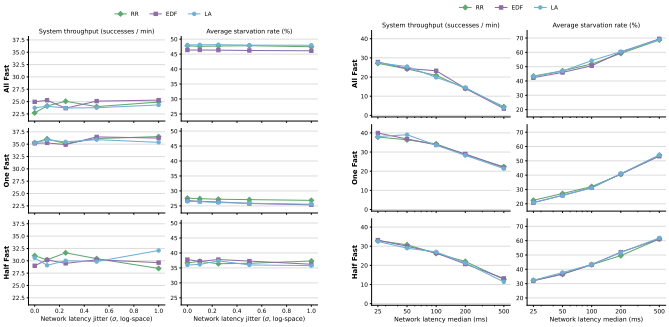


Fig. 5: How does the max batch size affect scheduler performance? We ablate this on the all-fast scenario. Lookahead performs most effectively by choosing smaller, more efficient batches, keeping starvation lower as the batch-size cap grows.



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

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

Fig. 6: Network ablation study showing how network delay affects cloud serving for $b = 3$.

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 II): 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 5 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 real, with our setup of 10 robots, $b = 3$ is the best parameter for all-fast, and $b = 5$ is the best parameter for one-fast and half-fast. This 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 simulation (§A7), we found $b = 3$ to be best globally.

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 III, 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.

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

$$d \sim \text{Lognormal}(\mu, \sigma^2), \qquad \mu = \ln m, \tag{6}$$

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 6a 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.

Network median delay. In Figure 6b 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.

7) Simulation metrics: **Setup.** 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 [22] 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). 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.

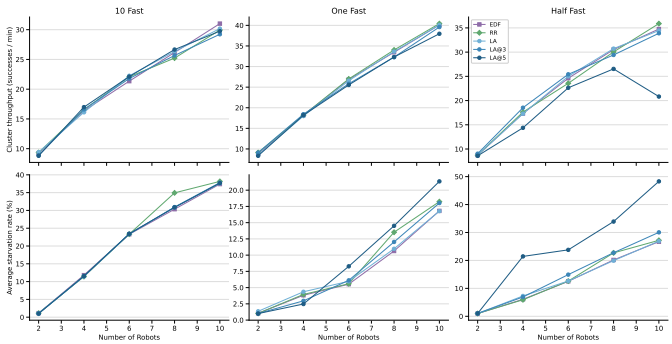


Fig. 7: System throughput (successes/min) and average starvation as the cluster scales from N=2 to 10 robots, with batch-size cap b=3.

Metrics. Figure 7 and Table IV report system throughput and average starvation as we scale from N=2 to 10 robots, and Figure 8 contains per-tier results. Throughput grows with N while starvation rises as more robots contend, and all schedulers achieve comparable *total* throughput and starvation. The core contribution of Lookahead is not in raising aggregate throughput, but in reallocating across tiers, giving fast/slow robots a more favorable share of the system's throughput.

8) Real world per-tier metrics: In Table V, we list the throughputs and starvations from our real-world experiments.

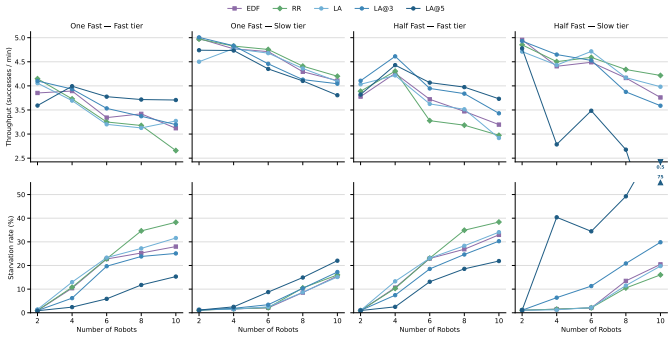


Fig. 8: LIBERO-10 simulation results. For one-fast, LA@5 is able to maintain a consistently higher throughput and lower starvation for fast-tier robots without sacrificing slow-tier performance. For half-fast, LA@3 is the most useful operating point: it improves fast-tier service while keeping the slow-tier 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.

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.

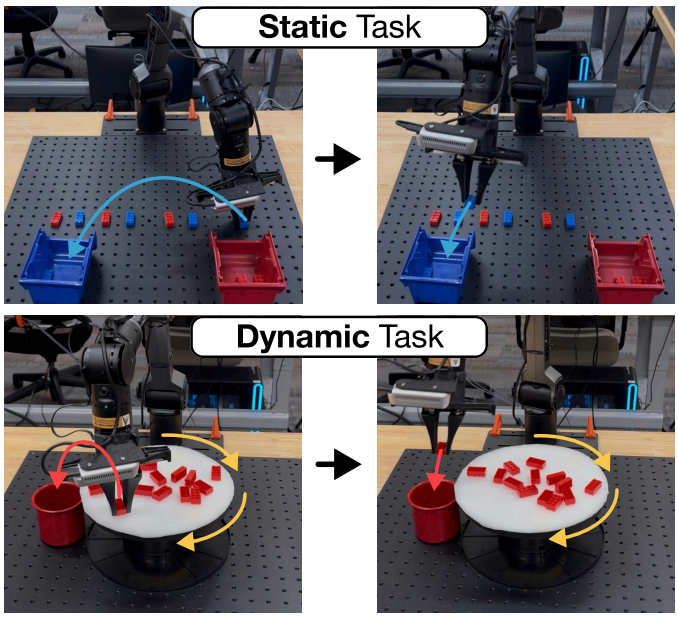


Fig. 9: Real-world task setup. Top is the static brick-into-bin sorting task, and bottom is the dynamic brick-into-mug turntable task.

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.

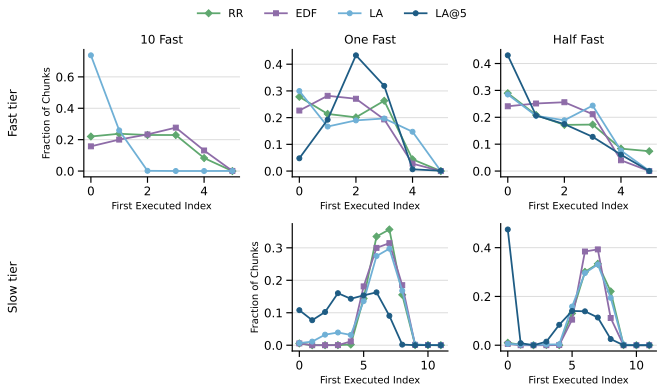


Fig. 10: Lookahead schedulers bias towards scheduling chunks so that they are executed starting at index 0.

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 [5] or VLASH [28] may lead to different results.

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, simulating the arrival and consumption of 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 [14].

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

Config	N	LA@1			LA@3			LA@5		
		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

TABLE IV: 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)

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

Config	Scheduler	Throughput (successes/min)			Starvation (%)		
		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