Kairos: A Scalable Serving System for Physical AI

Yinwei Dai
Princeton University

Ganesh Ananthanarayanan
Microsoft

Landon Cox
Microsoft

Xenofon Foukas
Microsoft

Bozidar Radunovic
Microsoft

Ravi Netravali
Princeton University

arXiv:2605.11381v1 [cs.RO] 12 May 2026

Abstract

Physical AI is experiencing rapid growth with frontier foundation models increasing its capabilities across general environments. Physical AI tasks are characterized by inference properties that are markedly different from digital AI. They consist of multiple rounds of inference and action execution, generating a chunk of actions in each inference round, and asynchronously interleaving inference and execution. This makes existing digital AI serving systems unsuited for physical AI; a shortcoming that is critical for enabling their wide adoption, considering their size and the scale of the robot fleets they have to serve. To fill this gap, we design Kairos, the first multi-robot serving system that makes the generate-execute loop a first-class citizen, with active involvement in the execution phase. Across a wide range of physical AI models and robots, Kairos reduces the average end-to-end task latency by 31.8–66.5% over state-of-the-art digital AI serving practices, with gains scaling with the robot fleet size.

1 Introduction

Physical AI – intelligent systems that perceive and act on the real world – is undergoing rapid growth. For instance, robotic platforms ranging from dexterous manipulation arms to full-scale humanoids are being deployed across warehouses, factories, hospitals, and households [1, 2, 9, 35, 51], with demand pushing toward fleets of hundreds or thousands of robots coordinating on diverse tasks. Underpinning these deployments is a new generation of frontier foundation models, including Vision-Language-Action models (VLAs), Video Action Models (VAMs), and World Action Models (WAMs), that map real-world sensory observations to robot motor commands with unprecedented generality. As physical AI scales in the size of models and number of robots, offloading inference out of the robot, and consolidating robot inference onto shared serving infrastructure becomes essential for cost efficiency [44].

Serving physical AI at scale, however, surfaces a fundamental difference from digital AI (e.g., LLMs for text generation) that existing serving systems are not designed for. In digital AI, inference is *decoupled* from the world: a model generates tokens from a fixed context, and those tokens remain valid regardless of when they are consumed. Physical AI breaks this property. To meet their high control frequencies, physical AI models generate actions in *chunks*, whereby a single inference call produces N actions that the robot executes sequentially while the next inference round runs *asynchronously* in parallel, keeping the robot in motion until it completes its *task*. However, robots act in a dynamic, changing world, so the observations used for each inference – e.g., camera images, joint states – are snapshots that grow stale as time passes and actions are executed, making later actions in each chunk increasingly outdated. This interplay between inference and a changing physical world has two implications for how serving systems must evolve.

First, *serving systems for physical AI must manage a new accuracy-efficiency tradeoff.* The number of generated actions to execute before a new round of inference (i.e., the *execution horizon*) directly controls the tradeoff: a shorter horizon replans with fresher observations (and higher accuracy) at proportionally higher serving load, while a longer horizon lowers inference cost but risks executing stale actions (and lowering accuracy). The optimal execution horizon varies across physical AI tasks, robots, and inference rounds within a task as execution demands shift, e.g., free-space motion tolerates a long horizon, while contact-rich grasping does not. Yet today, execution horizons are set statically by developers, conservatively for the worst case. Our results show that this leaves 62%–87% of rounds safely capable of executing up to 2.8× more actions without accuracy loss, hence 2.8× lower inference load – substantial efficiency opportunities that no existing serving system captures.

Second, *serving systems must account for execution times when scheduling.* In digital AI, task progress is purely a function of inference times. In physical AI, tasks are multi-round and interleave inference with real-world execution phases of the robot that are significant – often dominant – in end-to-end latency, and inherently heterogeneous across tasks and robots that share infrastructure (0.3s–1.6s for each execution phase at 30Hz control based on the selected execution horizon). This matters because a task's true urgency for when it next needs an inference is determined by its execution duration, not its generation history. Yet existing schedulers, which reason only about accumulated generation time used for inference or queuing delays, are oblivious to this: a task with little generation time may be the longest-running in wall-clock terms if its execution dominates, causing schedulers to systematically prioritize the wrong tasks. Indeed, across our workloads, this misidentification affects 50-93% of tasks, inflating end-to-end latency across the fleet.

We present **Kairos**, the first serving system designed for physical AI. Kairos's central insight is that efficient physical

AI requires the serving system to be a first-class participant in the generate–execute loop, not just a passive inference provider, both within the rounds of each task and across tasks sharing infrastructure. Within each task, Kairos treats the execution horizon as a dynamic, per-round knob managed by the serving system, adapting it in each round to the robot's current execution context. To do so without additional inference cost, Kairos exploits the inference confidence of each action's final diffusion update in the chunk. The confidence serves as a fine-grained staleness indicator, identifying the boundary in each round of actions beyond which generated actions are no longer reliable. Across tasks, minimizing end-to-end latency requires reasoning about when each task needs its next inference, which is gated by whichever phase (generation or execution) currently dominates. Kairos addresses this with *wait ratio* – the fraction of a task's lifetime spent waiting for compute. The wait ratio unifies phase delays into a trackable priority signal to avoid the inversions that plague generation-only schedulers.

We evaluate Kairos across six physical AI models spanning three architecturally distinct families (VLAs, WAMs, and VAMs), five simulation benchmarks (LIBERO, Meta-World, Isaac Lab, RoboTwin 2.0, SIMPLER), and real-robot experiments on a bimanual SO-101 platform. Compared to FIFO scheduling in existing LLM serving systems (e.g., vLLM [27]) and fairness-based scheduling in multi-round agent serving systems (e.g., Autellix [37]), Kairos reduces average end-to-end task latency by 31.8–66.5% at peak load in the online serving setting. When serving a dedicated fleet of robots that collectively execute a fixed set of tasks offline, Kairos's average latency reductions grow from 20.4% at 10 robots to 42.8% at 100 concurrent robots. We will open source Kairos post publication.

In summary, Kairos makes the following contributions. **(1)** We quantify the impact of the key parameter, execution horizon, controlling the accuracy-efficiency tradeoff (§2) and adapt it dynamically for resource savings (§3.1). **(2)** We build a scheduling system for a fleet of robots designed for the workload pattern of generation-execution rounds (§3.2). **(3)** We implement and evaluate our system with real robots to demonstrate feasibility and gains (§4, §5).

2 Background and Motivation

This section provides a primer on physical AI, introduces the state-of-the-art foundation models for physical AI and their shared properties (§2.1), and then examines why existing serving systems are suboptimal and the optimization opportunities that they leave untapped (§2.2 and §2.3).

Transformer-based foundation models have moved beyond the digital world [10, 18, 41, 56, 67] to transforming the physical world. Physical AI encompasses intelligent systems that perceive the real world and act on it. With a growing ecosystem of robotic platforms, e.g., dexterous manipulator arms, mobile bases, and full-scale humanoids [1–3], physical AI is poised to shape many domains, including warehouses [9], factories [51], homes [1, 2], and healthcare [35].

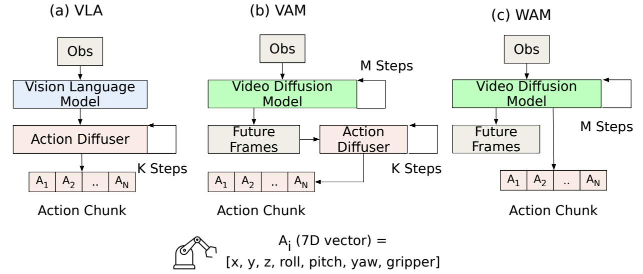


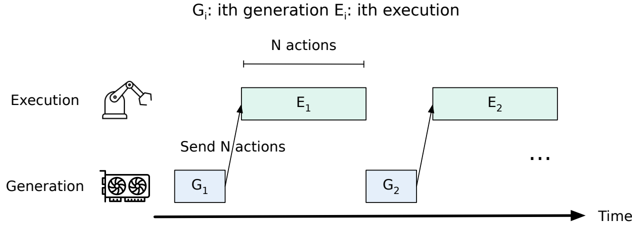
Figure 1. The state of the art physical AI models.

2.1 Physical AI Models

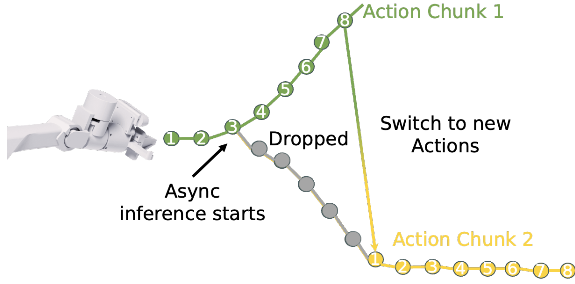
Foundation models for physical AI map *observational inputs* to *actions*. Example observational inputs include camera images, robot states, and task instructions. Actions, on the other hand, consist of low-level motor commands, such as gripper signals or joint positions, to advance the robot towards the instructed goal. Figure 1 shows the family of physical AI models for robotic manipulation. *Vision-Language-Action models* (VLAs; 1(a)) [12, 22, 26, 40, 48] feed observations through a pre-trained vision-language backbone and decode actions for the robot to execute from its joint visual-semantic representation. *Video Action Models* (VAMs; 1(b)) [31, 38, 42, 55] replace the vision language backbone with a video diffusion model that first predicts future visual frames, then decodes actions based on the predicted frames. *World Action Models* (WAMs; 1(c)) [25, 58, 60, 62] uses a video diffusion model to generate future frames as well as actions simultaneously. Our work targets this family of frontier physical AI models (VLAs, VAMs, WAMs); we next explain their key properties and differences compared to LLMs used in digital AI.

1) Action chunking. Modern robots operate at control frequences of 30 Hz (or higher), thus making the typical paradigm of generating output tokens autoregressively one after the other, to be impractical for robotic AI. With robots demanding a new action every ∼33ms, and given the inference times of hundreds of millseconds, generating each action with a separate inference would cause the robot to pause after every action, leading to jerkiness and loss in utility. State-of-the-art physical AI models instead employ *action chunking*: a single inference call produces a chunk of N actions at the same time using diffusion.[1] Its ability to enable high-frequency control of robots has made diffusion the dominant action-generating paradigm for all of the models in the physical AI family (Figure 1).

[1]We use "diffusion" broadly to include both denoising diffusion [15], which iteratively removes noise from a corrupted sample, and flow matching [33], which integrates a learned velocity field; both use a fixed number of steps to produce the full action chunk.



(a) Physical AI model serving is inherently multi-round: each round, the model generates a chunk of N actions for robot execution. This generation–execution cycle repeats until the task completes.



(b) Async inference overlaps the next inference with current execution, so a fresh action chunk is ready when the robot exhausts its actions. Actions in the overlap region (grey) are discarded to align with execution due to inference delay.

Figure 2. Physical AI model inference paradigm.

2) Asynchronous multi-round inference. A typical physical AI *task*, such as picking up a cup or setting a table cloth, requires multiple action chunks. With each chunk generated by an inference call, a physical AI task requires multiple *rounds* of inference in succession. A naive approach would generate a chunk of actions, and after the robot executes them, generate the next chunk with fresh observations (Figure 2a). Naturally, this leads to idling of the robot between inference rounds. To mitigate this, existing physical AI systems [11, 48, 49, 58, 60] apply *asynchronous inference*, which launches the next inference while the robot is still executing the current chunk (Figure 2b), thereby overlapping action generation with action execution to avoid (or reduce) idling.

2.2 Execution Horizon

As a direct consequence of the two points in (§2.1) – action chunking and multi-round inference – physical AI models suffer from *execution staleness*: all actions in a chunk are conditioned on a single observation captured at the start of each round, so later actions can become outdated and lead to degraded execution, e.g., collisions or missed grasping. This is unlike LLM based workloads in digital AI where generated tokens remain valid regardless of when they are consumed. To mitigate this staleness, robotic practitioners tune the *execution horizon, H*, which is a knob that controls

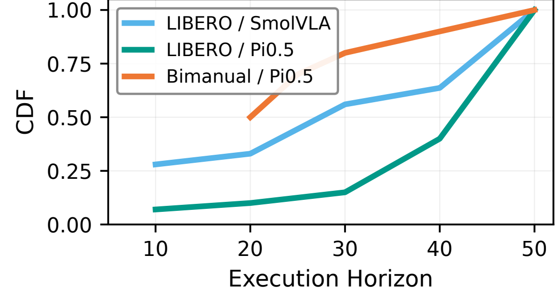


Figure 3. The best per-task execution horizon varies widely across tasks and workloads. Each curve shows the CDF of the best per-task H for a workload.

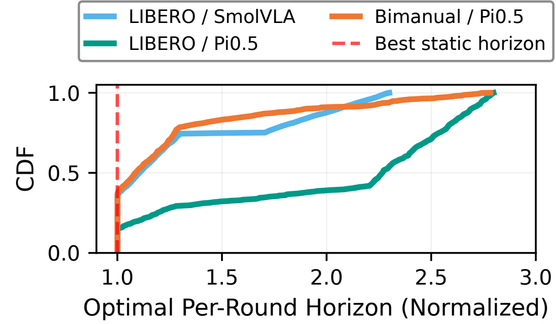


Figure 4. The best per-task execution horizon varies within a task. CDF of the optimal per-round optimal execution horizon, normalized by the best per-task horizon.

how many actions are executed from each chunk. The set of actions in the chunk beyond the first H are discarded.[2]

The execution horizon is a key parameter that allows *trading off accuracy with resource demand.* A shorter value of H generates actions more frequently with fresher observations, thus improving accuracy but also proportionally increasing inference load. A longer H, on the other hand, lowers inference cost but risks executing stale actions for the robot, which impacts its accuracy for complex tasks (e.g., grasping under contact uncertainty) that require fresher actions.

We quantify the variation across robotic tasks in the best value of H, i.e., the highest value of H that achieves the highest accuracy. By scanning for the highest possible value of H, we also minimize the inference load. We use representative workloads from our evaluation (Table 1), including simulated and real bimanual robot tasks. As Figure 3 shows, $\sim$ 40% of tasks see no drop in accuracy even for $H \geq 40$ out of a maximum of 50. In the simulator with the Pi0.5 model, less than 10% of tasks demand an execution horizon under 10. For the bimanual robot task, the minimum H value needed is only 20 while 10% of the tasks are fine with the horizon in excess of 40. The above results show not only a wide variation but also

[2] Since the number of actions generated in a chunk is already baked into the model architecture at training time and set large for stable convergence [15, 64], tuning the execution horizon is a practically simple approach compared to retraining the model for a new action chunk size of H.

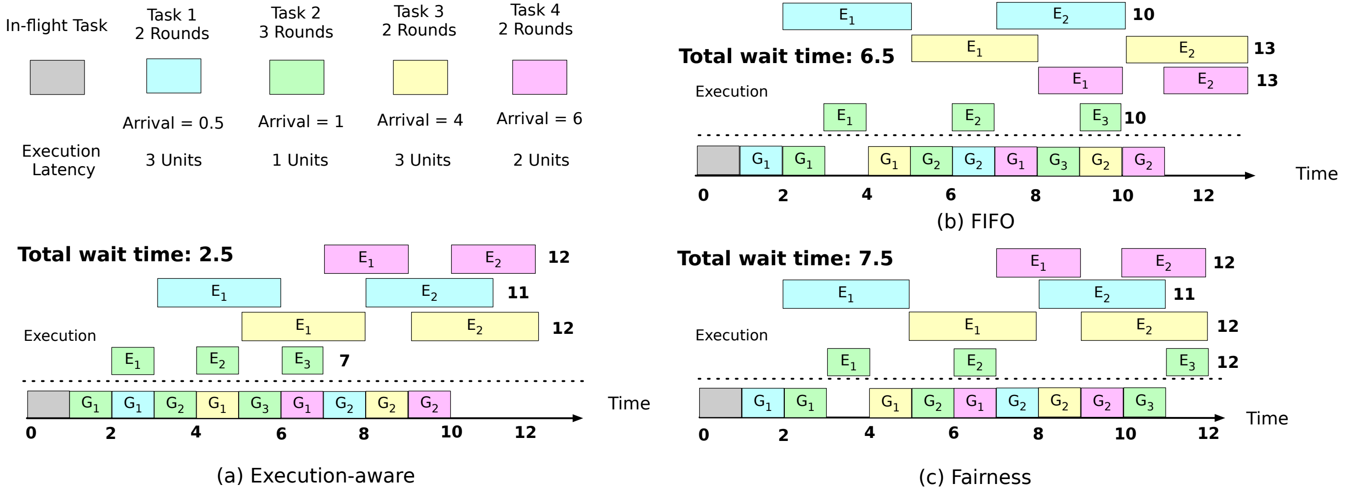


Figure 5. We use four physical AI tasks to illustrate that execution-unaware scheduling leads to suboptimal decisions. Task 2 (green) has longer total generation time but shorter end-to-end latency. FIFO ignores task-level information and only prioritizes request-level wait time, while Autellix [37] misclassifies it as a long-running task and prioritizes it. Execution-aware scheduling correctly prioritizes Task 2, reducing total wait time to 2.5 units compared to 6.5 (FIFO) and 7.5 (Fairness).

an opportunity to pick high values of H (which will reduce inference load) without impacting accuracy for certain tasks.

Further, the best value of H not only varies across tasks, but also *within the rounds of the same task*. The analysis above sets the same execution horizon across all the rounds of a task, which is reflective of the state-of-the-art practice in robotics deployments, for reasons of simplicity. Such a conservative value of H typically reflects the most complex section (round) of a robotic task, but not all sections are equally complex. Intuitively, moving to a soda can is easier than the actual step of grasping the can (that requires the very short horizon), but moving to the can may take longer for a robot to execute because of the distance it has to traverse. This means that most physical AI serving deployments today run inference too frequently most of the time.

We quantify the magnitude of such a conservative choice and the opportunity for improvement in Figure 4. We first collect the action traces using the task-optimal execution horizon set by Figure 3. We then implement a simple offline algorithm that finds the round-optimal execution horizon for each round in the recorded traces. For each recorded round, we start with the task-optimal horizon. We increase it in small steps, we rerun the inference and measure the difference between the newly obtained trajectory and the pre-recorded trajectory of the robot (using cosine similarity). The round-optimal execution horizon is the largest execution horizon for which the cosine similarity remains above a threshold (≥ 0.9, tuned to preserve task accuracy).

Figure 4 plots the CDF of the ratios of the round-optimal versus task-optimal horizons for all rounds across all tasks. As shown, half the rounds can increase their horizon by $\geq 2\times$ while preserving accuracy. Even for the real bimanual setup, 20% of rounds can increase H by $1.5\times - 2.8\times$. These present a significant opportunity to achieve inference efficiency (by having less frequent inference rounds) without impacting accuracy, which existing serving systems do not exploit.

2.3 Execution-aware scheduling

As physical AI scales in the size of models and number of robots, the serving infrastructure for robot AI models is becoming a consolidated theater for multiplexing inference for multiple robots [44]. Building upon §2.2 that discussed the resource optimization opportunities for a single robot's inference, we now discuss the scheduling opportunities in inference for multiple simultaneous physical AI models.

The latency of a physical AI task includes not only generation time (inference) but also execution, and as discussed, a task consists of multiple rounds of actions and inferences. Execution time can often be dominant depending on the execution horizon; at 30 Hz, execution time is 300ms at H=10 but shoots to 1.6 s at H=50. This can shift a task from being generation-dominant to execution-dominant. However, existing serving frameworks are *unaware* of execution times. They schedule shared GPU resources in FIFO order of individual inference requests [16, 27, 66], or for fairness (e.g., least attained service) to equalize GPU time for generation among tasks and to mitigate starvation [37]. This leads to sub-optimal latency of physical AI tasks.

Figure 5 presents an illustrative example of the above execution-unaware schedulers – FIFO and fairness – with a scheduler that is aware of execution times; the generation time is assumed to be the same for each inference. We ignore network latency and assume synchronous inference for simplicity. We consider four physical AI tasks, each with multiple rounds of generation (inference) and execution (actions), and each of the tasks has different execution times for

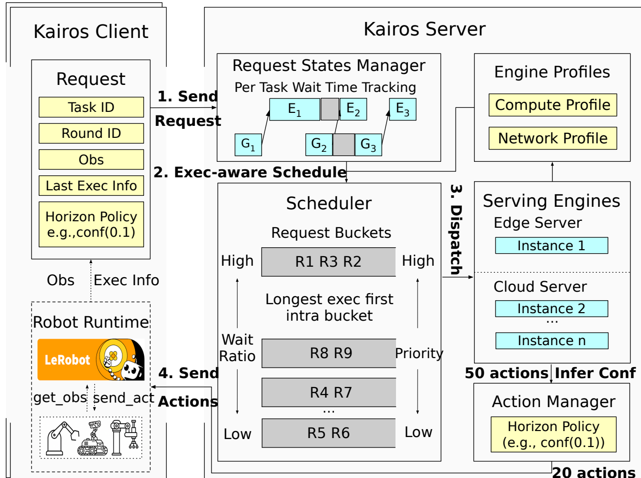


Figure 6. System architecture.

its respective execution horizons, keeping with the observations in §2.2. Focusing on Task 2's latency (green), we can see the benefit of execution awareness. FIFO and fairness schedulers prioritize tasks with longer execution times ahead of Task 2, do not focus on the end-to-end task latency, and end up starving shorter tasks. Task 2's latency is 7 units with an execution-aware scheduler, compared to 10 and 12 with FIFO and fairness schedulers. Further, even the average task completion time is lower for the execution-aware scheduler. This is a result of the average time spent waiting by each robot to be only 0.6 units with the execution-aware scheduler, compared to 1.6 and 1.8 for the FIFO and fairness schedulers. Note that under synchronous inference, wait time equals queuing delay; we discuss the asynchronous case in §3.2.

3 Kairos: System Design

We design Kairos, a serving system tailored for physical AI workloads, with the guiding principle of being *execution-aware*: it optimizes the number of actions executed per round (execution horizon) to balance accuracy and efficiency, and incorporate execution-phase information into scheduling to minimize end-to-end task latency. Specifically, Kairos introduces two techniques: (1) exposing model inference confidence as a fine-grained, per-round knob to dynamically select the execution horizon, achieving a better accuracy-efficiency trade-off than any static horizon (§3.1), and (2) an execution-aware scheduler that leverages both generation and execution phase information to minimize end-to-end latency across concurrent robotic requests (§3.2).

Figure 6 overviews Kairos's architecture. Kairos operates in a client-server model. Each client is co-located with a robot runtime and communicates with the server in a request-response loop: every round, the client sends a new observation along with metadata about its previous execution phase (step 1 in Figure 6), and the server returns an action chunk (step 4) whose length is determined dynamically by Kairos's execution horizon policy (§3.1). Kairos by default exposes a diffusion confidence threshold as a user-configurable knob, but provides a flexible interface for users to configure their own execution horizon policy. On the server side, Kairos maintains each client task's lifecycle state (e.g., generation and execution timestamps of past rounds) to obtain per-task cumulative wait times, which feed into an execution-aware scheduler (step 2) that prioritizes requests for inference based on the ratio of wait times over its lifetime (considering both generation and execution (§3.2). The scheduler dispatches requests to a pool of serving engine instances (step 3) spanning edge and cloud tiers. We detail the client-server interface and server components in §4.

3.1 Dynamic Horizon with Inference Confidence

Recall from §2.2 that the optimal execution horizon H varies across tasks, and even across rounds within a single task, yet has to be identified against a reference trajectory that is unavailable at inference time. Prior work [28] approximates such a reference trajectory via self-consistency: generating multiple candidate action chunks to agree on one. However, this approach significantly inflates inference cost, often even undermining the efficiency gains that a dynamic execution horizon provides. In contrast, we make the following observation: the diffusion process of action generation itself provides a confidence signal for each action *at no extra cost.*

Physical AI models generate the entire action chunk through K iterative refinement steps. At each step, the model produces an update for *every action* in the chunk. For example, with a 7-dimensional action space and a chunk size of N=50, each step produces a 50×7 matrix of updates — one 7D update vector per action. This process follows a well-established coarse-to-fine trajectory in which the early steps produce large and coarse corrections, and later steps contribute diminishing refinements. Prior work [30, 59] exploits this diminishing-update structure to accelerate inference by skipping refinement steps for output regions (e.g., image patches) deemed confident once their values stabilize. Our key observation is that, after K steps, not every action in the chunk has converged confidently: some actions exhibit diminishing updates throughout the refinement steps, while others still incur substantial updates at the final step. An action still being significantly revised at the last step is one that the model is uncertain about. Kairos uses this as a per-action confidence indicator — not to skip denoising steps, but to determine which actions in the generated chunk can be safely executed and where to re-plan.

Concretely, after each generation round, Kairos exposes the per-action per-step diffusion updates $\mathbf{U}$ to an execution horizon policy module. The policy maps $\mathbf{U}$ to the execution horizon H for the current round. This module is general: a static horizon policy simply ignores $\mathbf{U}$ and returns a fixed H, while dynamic policies can exploit $\mathbf{U}$ to adapt H each round based on the confidence of the predicted actions. Any method

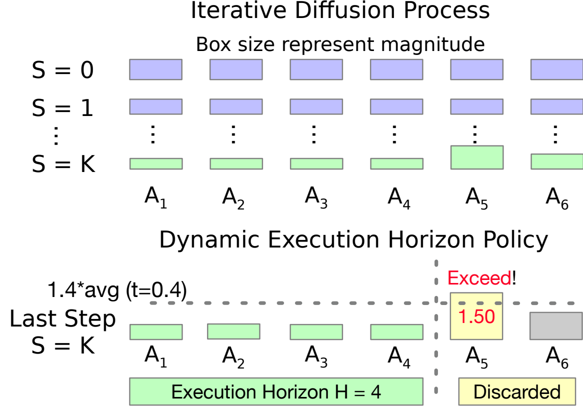


Figure 7. Diffusion confidence and dynamic horizon selection. Box sizes show per-action update magnitudes at each step. The horizon policy scans from A_1, comparing each action's final update magnitude against $(1+t)$ times its mean over earlier steps. Here A_5 exceeds the threshold, setting H=4 and discarding A_5–A_6.

that maps the intermediate diffusion updates or other intermediate inference states to the execution horizon decision can be plugged in, and this is configurable by the user, as shown in the horizon policy field in Figure 6.

As a default instantiation of this module, Kairos provides a threshold-based policy that uses the diffusion updates $\mathbf{U}$ to examine each action's degree of convergence with a threshold t as the tuning knob. Concretely, for each action in a generated chunk, we compare the magnitude of its final diffusion step update against the mean of its earlier updates. We use the final step as the reference because it is the update that directly produces the final action, making its magnitude a direct indicator of uncertainty. Walking sequentially from A_1, we stop at the first action A_i whose final update exceeds $(1+t)$ times that mean and set the execution horizon to H=i−1, retaining only the converged prefix $A_1, \ldots, A_{i-1}$. This is necessary because the robot executes actions in order: an unconverged action at position i compromises the trajectory from that point onward, regardless of whether later actions appear individually converged.

As shown in Figure 7, actions A_1–A_4 exhibit small final updates, indicating convergence, while A_5 is the first whose final update is 50% higher than its earlier mean, exceeding the t=40% threshold; so, Kairos retains A_1–A_4 and sets H=4 for this round. Kairos sets $H = \max(H_{\text{thresh}}, H_{\text{min}})$, where H_{min} is the smallest static horizon in the range we evaluate, ensuring a minimum number of actions per round. We observe that earlier actions are more likely to converge than later ones, with $H_{\text{thresh}} > H_{\text{min}}$ in the majority of cases. We show in §5.2 that using diffusion confidence as a tuning knob Pareto-dominates a static horizon knob across all our workloads.

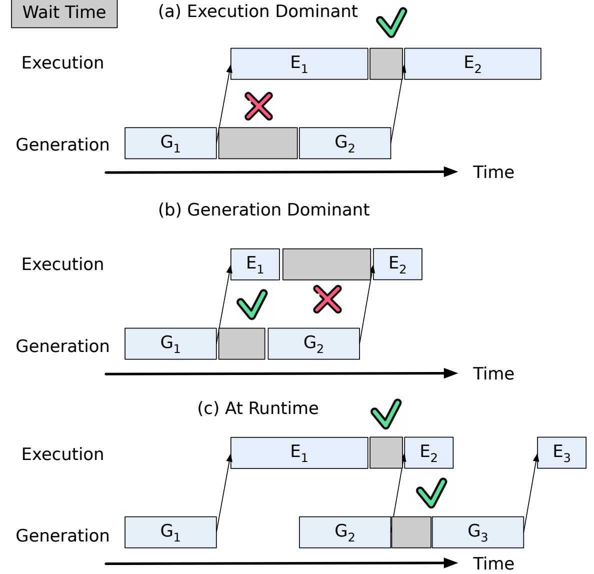


Figure 8. Wait time measurement. (a) Execution-dominated round: the optimal schedule overlaps G_2 with ongoing E_1 fully while starting G_2 as late as possible, so W between rounds 1 and 2 is the gap between E_1 ending and E_2 starting. (b) Generation-dominated round: W is the gap between G_1 ending and G_2 starting, since optimally G_2 begins right after G_1 with a fresh observation. (c) Kairos identifies the dominant phase per round and measures W accordingly.

3.2 Execution-aware Scheduler

To minimize end-to-end task latency when multiple physical AI tasks share a serving platform, the scheduler must reason about both generation and execution phases. Existing schedulers minimize wait time to achieve this, but define it solely in terms of queuing delays at the generation phase: FIFO greedily minimizes per-request queuing delay, while fairness-oriented policies such as least attained service first [20, 39] and Autellix [37] prioritize requests from tasks with lower accumulated generation time, reducing queuing delays for shorter tasks. However, these approaches ignore execution phases, which constitute a significant portion of end-to-end task latency and vary widely across tasks and across rounds (§3.1). Moreover, with asynchronous generation and execution (§2.1), queuing delay at the generation side does not directly translate to added wait time for a task—if the robot is still busy executing, the queued generation request is not yet needed to be scheduled. The right metric is the delay relative to *when the next action chunk is actually needed*. We first formalize wait time for physical AI tasks, then present Kairos's scheduling policy.

3.2.1 Wait time of physical AI tasks.

To quantify the wait time of each task, we compare each task's actual progress against its optimal progress—the latency it would achieve by starting generation as late as possible while still maximally overlapping generation and execution. We treat each

generation–execution round as a primitive: round i consists of a generation phase G_i followed by an execution phase E_i, and the total wait time is the sum of the wait times between all consecutive rounds. For two consecutive rounds, the optimal timeline—and therefore where wait time is measured—differs depending on whether the first round is generation-dominated or execution-dominated. Consider the first round being execution-dominated (Figure 8a): E_1 dominates G_1, and since diffusion inference latency remains constant, E_1 also dominates G_2. Therefore, the optimal schedule leverages async inference to start G_2 of the second round at the latest time possible, while fully overlapping with the ongoing E_1. The wait time between the two rounds is therefore the gap between E_1 ending and E_2 starting on the execution side. Recording the idle gap between G_1 and G_2 on the generation side would be incorrect: because execution dominates, a gap between G_1 and G_2 exists even under the optimal schedule. Similarly, when the first round is generation-dominated (Figure 8b), measuring the wait time on the execution side would be incorrect: even if G_2 of the second round starts as early as possible, a gap between E_1 and E_2 still exists because generation dominates. The earliest start time of G_2 possible could be right after G_1 with a fresh observation. The wait time is instead the gap between G_1 ending and G_2 starting on the generation side. As Figure 8c illustrates, Kairos therefore identifies the dominant phase each round and measures the wait time on that side.

To compute per-round wait times, Kairos maintains a per-task state record storing the sequence of generation intervals $(G_0, G_1, \ldots)$ and execution intervals $(E_0, E_1, \ldots)$. With asynchronous generation, a task's round i generation request may be sent while round $i{-}1$'s execution is still ongoing. The client piggybacks E_{i-1}'s start timestamp and the number of remaining actions on the generation request; since the robot executes actions at a fixed frequency, Kairos computes E_{i-1}'s end time from the request timestamp and the remaining action count. In this way, Kairos incrementally builds the complete generation and execution history of each task at runtime, with minimal bookkeeping overhead on both the client and server. From the per-round generation and execution history, Kairos computes each task's accumulated wait time and uses it to assign scheduling priorities, aiming to minimize wait time and therefore end-to-end latency.

3.2.2 Scheduling policy. Given per-round wait times, the schedule decides how to prioritize pending generation requests with the goal of minimizing wait times across concurrent tasks. Intuitively, given per-task wait times, a natural approach is to prioritize by accumulated wait (similar to least attached service), but this biases toward long-running tasks that naturally accumulate more wait, preventing shorter ones from being scheduled. Instead, we borrow insights from Highest Response Ratio Next [21], and normalize wait time by each task's elapsed lifetime to prioritize high *wait ratio*

Algorithm 1: Execution-aware scheduling

Input: Pending inference requests $\mathcal{R}$ from physical AI tasks, buckets B, aging interval A, edge profile $\mathcal{P}_e$ with capacity N_e, cloud profile $\mathcal{P}_c$ with capacity N_c
Result: Dispatch assignments

```
// Phase 1: Compute wait ratio and bin requests
1  foreach r ∈ R do
2      foreach round j with recorded G_j, E_j do
3          if |G_j| ≥ |E_j| then W_j ← G_{j+1}.start − G_j.end else
               W_j ← E_{j+1}.start − E_j.end
4      end
5      w_r ← Σ_j W_j/(t_now − t^r_start); assign to b_r ← ⌊w_r · B⌋
6      if r.skipped ≥ A then assign to
           b_r ← min(B − 1, b_r + ⌊r.skipped/A⌋)
7  end
// Phase 2: Within-bucket request ordering
8  S ← [ ]
9  for b = B−1 to 0 do
10     ê_r ← |E^r_last| · (1 + r.skipped) for each r ∈ bucket b; sort
           by ê_r desc; append to S
11 end
// Phase 3: Shortest estimated delay guided
   placement
12 S_e ← S[1 : N_e]; S_c ← [ ]
13 foreach r ∈ S[N_e+1 :] do
14     if cloud latency < edge delay and |S_c| < N_c then
           append r to S_c
15 end
16 foreach r ∈ S_e ∪ S_c with stale obs do
       r.obs ← fetch_obs(r.client)
17 Dispatch S_e to edge, S_c to cloud
18 foreach r ∈ R do
19     if r ∈ S_e ∪ S_c then r.skipped ← 0 else
           r.skipped ← r.skipped + 1
20 end
```

requests to minimize wait time while improving fairness.

$$wr = \frac{\sum_j W_j}{t_{\text{now}} - t_{\text{start}}} \tag{1}$$

where W_j is the wait time for round j. A high wait ratio indicates that a task has been disproportionately delayed and should be prioritized; a low one indicates the client has been well-served relative to its lifetime. Algorithm 1 summarizes the full scheduling policy; we describe the details below.

Scheduling by continuous wait-ratio values can degrade into round-robin when many tasks have similar ratios [37]. The scheduler, therefore, uses a two-level hierarchical priority (Algorithm 1). The scheduler uses a two-level hierarchical priority. At the coarse level, it discretizes the $[0, 1]$ wait-ratio range into B (default 10) equal-width buckets, bins each generation request by its task wait ratio, and processes buckets from highest to lowest. Within each bucket, ties are broken

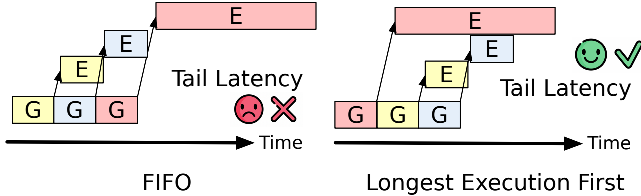


Figure 9. Within-bucket ordering by estimated execution latency. Left: FIFO ordering can leave long-execution tasks to the end. Right: longest execution first reduces the tail latency while keeping the average latency.

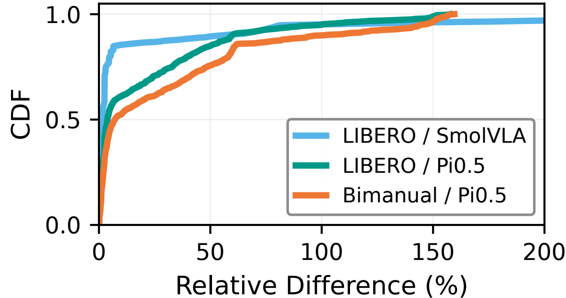


Figure 10. CDF of the relative difference between consecutive-round execution latencies. Across workloads, 52.3–85.4% of rounds have less than 10% deviation.

by estimated execution latency (descending). The reason is that scheduling execution-dominant tasks first prevents long execution phases from starving, which can reduce tail latency while maintaining the average latency (Figure 9, right vs. left). However, this ordering requires knowing the execution duration before actions are generated. To estimate the current round's execution latency, our insight is that physical execution is continuous—a robot's motion phase changes gradually across rounds—so the last observed execution duration is a reliable predictor of the current round. Figure 10 validates this: 52.3–85.4% of consecutive rounds differ by less than 10%; only 4.9–10.1% of rounds exhibit a relative difference exceeding 100%, which corresponds to the cases with sharp transition boundaries where they shift between execution and generation dominant.

To prevent starvation, the scheduler tracks the number of consecutive rounds in which a task's generation request is not selected; after every A skipped rounds, the request is promoted by one bucket, guaranteeing progress regardless of its current wait ratio (line 7). However, bucket promotion alone is insufficient for generation-dominant tasks: even in the highest-priority bucket, a short execution duration would always lose the within-bucket ordering. The scheduler thus scales the estimated execution latency by an aging factor $(1+\mathit{skipped})$, preventing requests from generation-dominant tasks to be indefinitely starved (lines 8–10). Once a task's generation is scheduled, its skip counter is reset to 0 (line 19).

Finally, before dispatching to the serving engine, if a request's observation has grown stale due to queuing, the scheduler re-fetches it from the client to ensure an up-to-date observation to incorporate any observation changes during its queuing delay(line 16). The scheduler then batches dispatched requests up to the maximum batch size, determined by the saturation point of the latency—batch size profile ($\mathcal{P}_e$ or $\mathcal{P}_c$) beyond which additional batching increases latency without improving throughput. Once dispatched, a request's skip counter is reset to zero (line 19).

3.2.3 Support hybrid edge and cloud serving. Physical AI tasks prefer co-located edge servers to avoid excessive network latency to the cloud [24]. However, edge capacity is inherently limited: as fleet size grows or bursty workloads spike demand, edge queues deepen, and latency degrades. If done carefully, a hybrid edge–cloud deployment can absorb these overflow bursts on cloud GPUs, trading network round-trip cost for shorter queuing delay and faster computing. Many physical AI models are stateless—conditioning only on the current observation and task description—and provide a more flexible placement opportunity compared with LLM serving, so requests can be freely placed on any available instance. Unlike autoregressive LLM inference where output length varies unpredictably, physical ai models run a fixed number of diffusion iterations, enabling Kairos to profile latency-vs-batch-size curves ($\mathcal{P}_e$, $\mathcal{P}_c$) offline for each serving tier. The scheduler prefers to dispatch to the edge server and offloads to the cloud only when the estimated cloud latency—including network round-trip—is lower than the expected queuing delay at the edge (lines 12–15). For models that maintain history across rounds [25, 47, 60], this estimation additionally includes historical data transfer cost.

4 Implementation

To our knowledge, no existing serving system supports online serving of concurrent physical AI tasks. We reimplement state-of-the-art scheduling techniques from LLM and multi-round agent serving systems as baselines (§5.1). We build Kairos in 3.7K lines of Python atop LeRobot [13], a popular open-source framework for physical AI, reusing its abstractions for simulated and real robots. We next describe the client-server interface in Kairos's design (§3).

Client. The Kairos client integrates with LeRobot's robot runtime through a lightweight shim that exposes two functions: `get_obs` for capturing the current observation and execution information, and `send_act` for streaming actions to the robot controller. The client communicates with the server via gRPC, attaching a unique *Task ID*, a monotonically increasing *Round ID*, the current observation, and *Last Exec Info* reporting the start and end timestamps of the previous execution phase to each request. Kairos uses the diffusion-confidence-based execution horizon policy (§3.1) by default, but exposes a flexible *Horizon Policy* field for user customization. The Last Exec Info, Round ID, and Horizon Policy fields are what distinguish Kairos's interface from

a standard physical-AI inference-serving API such as LeRobot's: they grant the server continuous visibility into the physical-world phase of each request's lifecycle, enabling the execution-aware scheduling described in §3.2.

Server. The server is built as an asynchronous event loop. Models are compiled with `torch.compile` to optimize inference latency. The *Execution-aware Scheduler* assigns priorities and dispatches requests to engine instances based on *Engine Profiles*, which record per-instance compute (batch size vs. latency) and network latencies from offline profiling. Dispatched requests are dynamically batched (up to the engine's specified max batch size) to maximize throughput. The *Action Manager* sits on the return path: after an engine produces an action chunk, it applies the client's horizon policy and returns the trimmed action sequence to the client.

5 Evaluation

We evaluate Kairos across a wide range of physical AI workloads, models, varying loads, and different deployment setups. Our key findings are:

- **Diffusion confidence Pareto-dominates static horizon as an efficiency–accuracy knob**, yielding up to 2.67× longer execution horizons at matched accuracy or up to 30% accuracy gains at matched horizons (§5.2).
- **Kairos scales to increasing online serving loads**, reducing average latency by 31.8–66.5% over both baselines at peak load (up to 88.4% P25, 52.0% P95) (§5.3).
- **Kairos scales with robot fleet size**, with growing average latency reductions of 20.4% (10 robots) to 42.8% (100 robots)
- **Kairos generalizes to diverse serving setups**, reducing average latency by 36.9–47.7% over edge-only and 51.9–67.9% over cloud-only in a hybrid edge–cloud setup (§5.3).

5.1 Experimental Setup

Physical AI workloads. For simulation workloads, we use three common benchmarks: (1) **LIBERO** [34], a suite of 40 language-conditioned manipulation tasks across four categories: spatial reasoning, object manipulation, procedural knowledge and long-horizon planning on a simulated Franka Panda [63] arm; (2) **Meta-World** [61], 50 manipulation tasks on a simulated Sawyer [46] arm divided in easy, medium and hard difficulty categories; and (3) **NVIDIA Isaac Lab** [5], a GPU-accelerated robot learning framework built on Isaac Sim [6] for high-fidelity robotic simulation, where we evaluate on a simulated Fourier GR1 [4] humanoid robot across 32 manipulation tasks spanning two categories in different environments. (4) **RoboTwin 2.0** [14], a scalable bimanual manipulation benchmark with structured domain randomization across 50 dual-arm tasks spanning multiple robot embodiments; and (5) **SIMPLER** [32], a real-to-sim evaluation suite that mirrors common real-robot setups and exhibits strong correlation with real-world policy performance. For real-robot experiments, we deploy on a customized **bimanual SO-101** [7] setup, where one arm picks up an object and hands it to the other arm, which then places it into a box. We construct tasks by repeating this handoff-and-place procedure 20 times with varied initial object placements.

Models. We pair these workloads with 6 models—four VLAs (Pi0.5 [22], SmolVLA [48], GR00T N1.5 [40], XVLA [65]), one WAM (Fast-WAM [62]), and one VAM (mimic-video [42])—to evaluate Kairos on architecturally distinct model families. Table 1 summarizes the workload model pairs.

Testbed. All serving experiments run on an edge server equipped with an NVIDIA RTX A6000 GPU; edge–cloud experiments use an NVIDIA A100 (80GB) GPU in the cloud server. Following existing practice in physical AI serving [13, 24], we set the robot to the edge server gateway via Wi-Fi 7 (2.50 ms base latency, 2 Gbps upload, 3 Gbps download) and the edge gateway to the cloud server via a WAN link (100 ms base latency, 1 Gbps symmetric). We evaluate additional edge–cloud network configurations in §5.5.

Baselines and metrics. We compare Kairos against two baselines: (1) FIFO, a first-come-first-served scheduler and (2) Autellix [37], a multi-round-aware scheduler that prioritizes requests with lower accumulated generation time. Both are assigned the largest static execution horizon H that maintains the highest accuracy, tuned offline, and held constant throughout a task. Both baselines represent the state of the practice: FIFO is the default policy in existing LLM serving frameworks [16, 27, 66], while Autellix is the state-of-the-art scheduler for multi-round agent workloads but is unaware of the physical execution phase. For evaluation, we report two metrics: *accuracy*, the fraction of tasks completed successfully, and *end-to-end latency*, the wall-clock time from the first inference request to the last action executed, reported at the average, P25, and P95 over all tasks.

Evaluation Methodology. We evaluate Kairos with real and simulated robots. However, our serving scalability evaluation requires varying the number of concurrent robots, but co-locating many simulator instances on a single machine distorts execution timing and task accuracy, and physical robots are expensive to provision at scale. We therefore adopt a trace-driven evaluation. Specifically, we first execute each task in isolation—including on a physical robot and via a single simulator instance—and record a trace: the complete observation–inference–execution sequence along with the action index at which each inference request is triggered. At evaluation time, we replay many such traces concurrently against the serving system; each replayed application issues its inference request only at the recorded action index, faithfully preserving the original execution horizon. As in prior work [27, 66], we sample from collected task traces and simulate task arrivals using a Poisson process with varying arrival rates to capture different load conditions.

Table 1. Summary of model workload pairs.

Robotic Benchmarks	VLA Model(s)
LIBERO / simulator	Pi0.5, SmolVLA, XVLA
MetaWorld / simulator	SmolVLA
Isaac Lab / simulator	GR00T N1.5
Real robot / Bimanual SO101 (real)	Pi0.5
RoboTwin / simulator	Fast-WAM
SIMPLER / simulator	mimic-video

5.2 Effectiveness of Diffusion Confidence

For each workload, we sweep the static horizon h and the confidence threshold t, running 10 trials per configuration to account for randomness in simulation and the diffusion process, and plot task accuracy against the average execution horizon—a proxy for system efficiency, since shorter horizons yield more frequent inference. Figure 11 shows the results across all six workloads. The fine-grained confidence knob (orange) consistently Pareto-dominates the static execution horizon (gray). At comparable accuracy, it achieves up to 2.67× longer average execution horizons; at comparable horizons, it improves accuracy by up to 30%. The gap demonstrates the benefit of a fine-grained, per-round execution horizon over a static, per-task one.

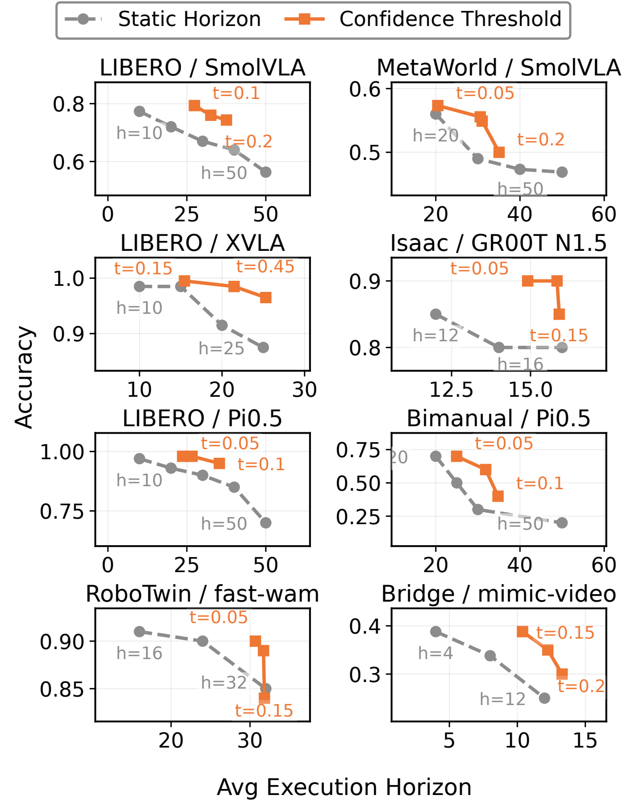


Figure 11. Accuracy–efficiency trade-off across six workloads (Table 1). Each gray point corresponds to a fixed horizon h; each orange point corresponds to a confidence threshold t. The confidence-based policy consistently Pareto-dominates the static horizon across all workloads.

5.3 End-to-End System Performance

In this section, we evaluate Kairos's end-to-end serving performance under two deployment scenarios: (1) an online setting where tasks arrive according to a Poisson process at varying rates, evaluated on both an edge-only server and an edge–cloud configuration; and (2) an offline setting where a fixed set of tasks is executed back-to-back by robot fleets of varying sizes. To focus on evaluating serving efficiency, we configure the execution horizon policies of Kairos and baselines using the accuracy–efficiency trade-off results from §5.2: for the static-horizon baselines (FIFO and Autellix), we assign each workload the largest fixed execution horizon H that achieves the highest accuracy for each task; for Kairos, we select the highest confidence threshold t that matches or exceeds that accuracy. Across all workloads, Kairos consistently achieves lower latency than both baselines, with the gap widening at higher task rates.

Online serving with varying request rates. Figure 12 shows the end-to-end latency performance under varying system loads across physical AI workloads with a single edge server. For each workload, we sweep the task arrival rates from low to high and report the average latency, with error bars spanning the P25 and P95 range. Across all workloads, Kairos consistently achieves lower latency than both baselines, with the gap widening as task arrival rates scale. Notably, Autellix consistently performs worse than FIFO, as it assigns priority solely based on total generation time, which can starve the very short tasks it is designed to prioritize. Under the highest load, Kairos reduces average latency by 31.8–66.5%, P25 latency by 39.5–88.4%, and P95 latency by 22.2–52.0% over both baselines. Table 2 reports the full latency reduction results at the highest arrival rate across all workloads. We further ablate two key components of Kairos: the dynamic execution horizon and the execution-aware scheduler.

First, replacing the Kairos scheduler with either baseline scheduler while keeping the dynamic execution horizon alone reduces P25 latency by 3.9–58.0%, average latency by 15.1–54.9%, and P95 latency by 18.9–49.8% over its static-horizon counterpart. This confirms that the dynamic horizon directly reduces total serving load by reducing inference requests. However, a latency gap up to 43.9% P25, 22.3% average, and 12.2% P95 remains compared to the full Kairos system, showing that dynamic execution horizon alone can not compensate for execution-unaware scheduling.

Second, removing the dynamic execution horizon, Kairos still yields substantial reductions in P25 (21.1–63.5%), average (10.9–39.3%), and P95 (4.1–13.2%) latency over both static baselines, validating the effectiveness of execution-aware scheduling. Notably, Kairos (static) achieves higher P25 reductions than the baseline schedulers paired with the

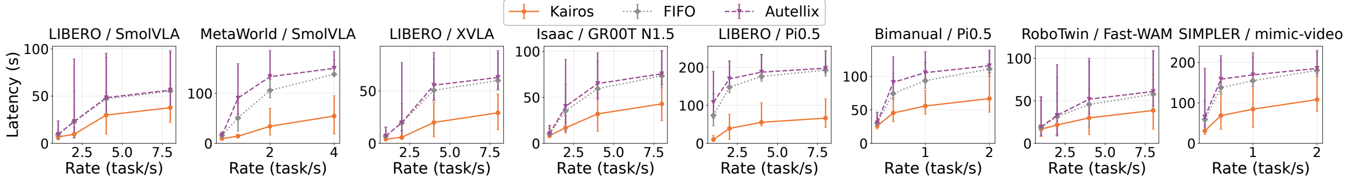


Figure 12. Average end-to-end latency under increasing task arrival rates across workloads. Error bars show P25 and P95.

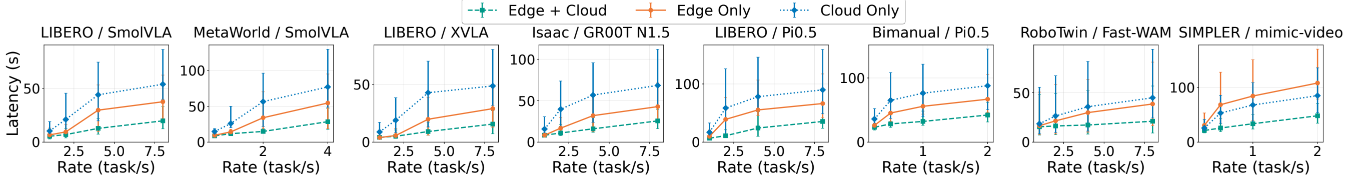


Figure 13. Effect of edge–cloud offloading on average end-to-end latency for increasing arrival rates. Error bars show P25-90.

dynamic horizon (up to 26.9%) in the majority of workloads. This validates that Kairos's execution-aware scheduler, with visibility into both generation and execution phases, correctly identifies and prioritizes the truly short requests—without coming at the expense of longer tasks. Finally, Kairos (which combines both components) delivers the largest latency reductions.

Online serving across hybrid edge-cloud deployments. Figure 13 further evaluates Kairos under hybrid edge–cloud deployments, where Kairos dynamically offloads requests to a cloud GPU (A100 80GB) over a 100 ms WAN link (§5.1). At the highest arrival rate, Kairos (hybrid) reduces average latency by 36.9–47.7% over Kairos (edge-only) and 51.9–67.9% over Kairos (cloud-only) across all workloads, similarly for P25 and P95. Cloud-only serving consistently performs the worst due to the WAN round-trip overhead for every request, with the exception of mimic-video, where the cloud GPU's faster compute outweighs the WAN cost and makes cloud-only more efficient than edge-only.

These results show that Kairos's scheduler generalizes to heterogeneous serving settings, maintaining each task's generation and execution bookkeeping regardless of each inference request's placement and inference latency, effectively using extra cloud capacity to reduce latency.

Offline serving with varying fleet sizes. In addition to evaluating Kairos as a shared online serving platform, we evaluate it in dedicated offline deployments, where a fleet of n robots collectively executes a fixed set of tasks back-to-back. We vary n from 10 to 100 robots to measure Kairos's performance across different deployment scales. Figure 14 shows the results. Kairos consistently outperforms both baselines across all fleet sizes and workloads, and the gap widens as the fleet scales up: at 10 robots, Kairos reduces average latency by 20.4–21.4% over two baselines; at 50 robots, the reduction grows to 33.9–37.8%; and at 100 robots, it reaches 41.0–42.8%. This widening gap demonstrates that Kairos scales effectively with fleet size, as the dynamic execution horizon reduces the total number of inference requests while the execution-aware scheduler more effectively utilizes the serving budget under increasing contention.

Table 2. Latency reduction over static baselines at the highest arrival rate. Results are for Kairos and ablations removing the scheduler or dynamic execution horizon.

(a) vs. FIFO

	Kairos			Kairos w/o dynamic horizon			Kairos w/o scheduler		
Workload	P25	Avg	P95	P25	Avg	P95	P25	Avg	P95
LIBERO/SmolVLA	39.5%	31.8%	34.6%	21.1%	10.9%	9.9%	3.9%	16.8%	29.0%
LIBERO/XVLA	74.5%	51.4%	45.2%	39.4%	17.8%	4.5%	44.6%	41.9%	37.5%
LIBERO/Pi0.5	76.4%	65.6%	51.9%	57.0%	33.0%	5.7%	58.0%	54.9%	49.6%
MetaWorld/SmolVLA	86.9%	60.5%	43.0%	59.0%	34.1%	4.3%	49.9%	43.7%	39.3%
Isaac/GR00T N1.5	59.2%	41.6%	29.0%	41.7%	24.5%	7.6%	24.0%	25.4%	23.4%
Bimanual/Pi0.5	53.3%	39.9%	22.2%	47.9%	28.1%	4.1%	24.9%	24.3%	18.9%
RoboTwin/Fast-WAM	57.9%	33.0%	25.7%	29.0%	8.1%	8.0%	46.5%	32.5%	21.7%
Bridge/minic-video	52.2%	40.1%	28.5%	51.0%	27.2%	2.8%	36.6%	31.4%	24.5%

(b) vs. Autellix

	Kairos			Kairos w/o dynamic horizon			Kairos w/o scheduler		
Workload	P25	Avg	P95	P25	Avg	P95	P25	Avg	P95
LIBERO/SmolVLA	41.9%	33.1%	36.1%	24.2%	12.5%	11.9%	8.0%	15.1%	25.5%
LIBERO/XVLA	74.9%	53.7%	46.7%	40.3%	21.6%	7.1%	43.2%	39.4%	36.1%
LIBERO/Pi0.5	77.4%	66.5%	52.0%	59.0%	34.6%	5.9%	55.8%	53.9%	49.8%
MetaWorld/SmolVLA	88.4%	63.6%	48.2%	63.5%	39.3%	13.2%	44.5%	41.3%	36.0%
Isaac/GR00T N1.5	61.6%	43.3%	29.2%	45.2%	26.7%	7.9%	27.5%	24.2%	22.9%
Bimanual/Pi0.5	55.9%	42.3%	24.6%	50.8%	31.0%	7.0%	23.9%	23.1%	19.8%
RoboTwin/Fast-WAM	62.4%	36.6%	25.2%	36.6%	12.9%	7.3%	44.2%	34.2%	22.5%
Bridge/minic-video	54.6%	41.6%	28.7%	53.4%	29.1%	4.9%	34.9%	30.4%	25.9%

5.4 Trace Fidelity Evaluation with Real Robots

When replaying many traces concurrently, scheduling decisions may delay action delivery, causing a robot to stall when freshly generated actions are not yet available; however, once generation completes, the generated actions themselves

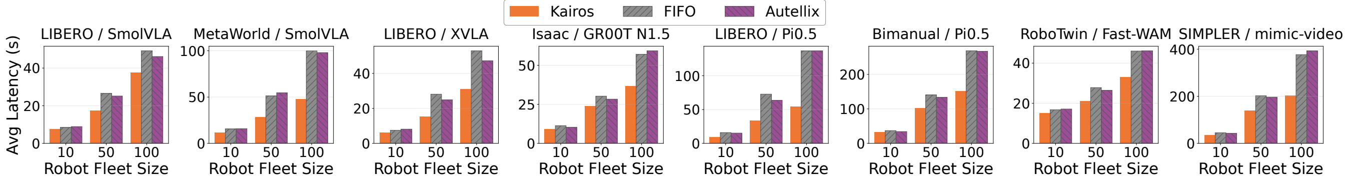


Figure 14. Average end-to-end latency as the dedicated robot fleet scales from 10 to 100 concurrent robots.

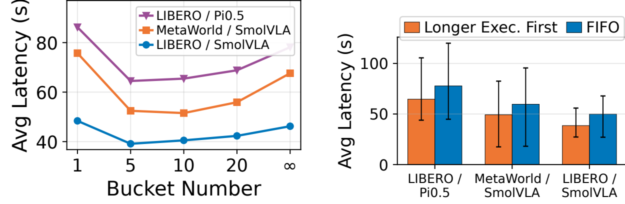


Figure 15. Impact of wait ratio bucket count on average latency.

Figure 16. Impact of intra-bin sorting policy. Error bars show P25–P95 range.

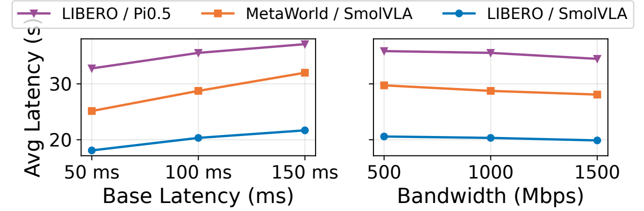


Figure 17. Impact of edge to cloud network conditions on average latency. (Left) Varying base latency at 1000 Mbps bandwidth. (Right) Varying bandwidth at 100 ms base latency.

remain identical to those produced in isolation. In simulation, stalls are benign: the simulator pauses deterministically and resumes from an identical state. We therefore validate through real-robot experiments on the bimanual SO-101 setup that replay-induced stalls do not impact task accuracy. Specifically, we randomly sample 20 tasks from the bimanual online serving experiments (Figure 12 in §5.3) at each arrival rate and inject the corresponding contention-induced delays on the physical robot. Across all arrival rates (0.5, 1.0, and 2.0 tasks/s), accuracy remains stable at 13–14/20, matching the original contention-free trace (13/20). The slight variation at rate = 0.50 is within expected noise from diffusion stochasticity and imperfect initial object placements, confirming the fidelity of our trace-driven evaluation.

5.5 Sensitivity and Ablation Studies

In this section, we microbenchmark parameters in Kairos's scheduler. Results are based on three representative workloads; the observed trends hold across all workloads. The experiment setup follows §5.1.

Impact of bucket size. Figure 15 shows the effect of varying the number of wait ratio buckets (default 10) on average latency. Too few buckets (e.g., 2) coarsen the priority signal, grouping requests with substantially different priorities into the same bucket, and lead to worse latency performance (up to 33.9% worse). Conversely, too many buckets (e.g., ∞, i.e., prioritize by absolute wait ratio) eliminate the stabilizing effect of bucketing, causing frequent switching between scheduling requests. Across all three workloads, 5–10 buckets strike the best balance between priority granularity and scheduling stability.

Impact of intra-bucket policy Figure 16 validates the choice of sorting by estimated execution latency (descending) within each bucket. Compared to FIFO, it reduces tail latency (P95) by up to 17.9%, and these per-round gains accumulate into lower average end-to-end latency as well.

Impact of network latency in hybrid setup. Figure 17 examines how edge-to-cloud network conditions affect Kairos's hybrid serving performance. Faster network connections (lower base latency, higher bandwidth) benefit Kairos by creating more opportunities to offload requests to the cloud (up to 20.5% more requests offloaded to cloud). Since observation payloads in our physical AI workloads are small, base latency dominates network latency, making Kairos more sensitive to base latency than bandwidth.

6 Additional Related Work

LLM and Agent serving. LLM-based applications have moved beyond text-based chatting toward tool-augmented workflows that interleave token generation with external function calls [8, 43, 57], forming a synchronous loop of generation and execution. Existing works optimize this interleaving through KV-cache management during tool-call pauses to support more concurrent requests [8, 29], but none schedule requests factoring in execution-phase duration, which constitutes a significant fraction of end-to-end latency in physical AI. Other work proposes asynchronous tool calling [19, 53], which fully decouples generation from execution so that the model need not wait for tool-call results. Physical AI is fundamentally different: action chunking produces multiple actions per inference call, enabling asynchronous inference without fully decoupling generation from execution; the right time to trigger the next generation depends on how far the current action chunk has been executed. Kairos is built with such execution-awareness for physical AI that digital AI systems lack.

Efficient physical AI. Prior work reduces the diffusion cost through skipping diffusion steps [45, 52, 60]. Orthogonally,

model compression techniques—quantization [17, 54] and pruning [23, 36]—reduce per-step compute at the model level. These techniques optimize inference efficiency at the model level and are thus composable with Kairos.

Tiered physical AI systems. Several physical AI systems decompose reasoning and control across tiers: a cloud-hosted LLM performs high-level planning across multiple tasks while an on-device physical AI model handles them one by one [50]. Kairos focuses on serving the physical AI model tier, treating the upstream planner as an input source. Other systems leverage a dual-system design, pairing a slow perception-reasoning module (System 2) with a fast on-device action generator (System 1) [2, 50]. Kairos's techniques seamlessly apply to serving the shared System 2 across robots.

7 Conclusion

We presented Kairos, the first serving system designed for physical AI serving. Kairos introduces two key mechanisms: a diffusion confidence measure that dynamically adjusts per-round execution horizons, and an execution-aware scheduler that leverages visibility into both generation and execution phases to prioritize requests under contention. Across four model families, three simulation benchmarks, and real-robot experiments, Kairos reduces average end-to-end task latency by 31.8–66.5% over state-of-the-art serving practices at peak load, scales from 10 to 100 concurrent robots with growing gains, and generalizes to hybrid edge–cloud deployments.

References

[1] 1X Technologies. https://www.1x.tech, 2024.
[2] Figure AI. https://www.figure.ai, 2024.
[3] Unitree Robotics. https://www.unitree.com, 2024.
[4] Fourier gr-1. https://www.fftai.com/products-gr1, 2026.
[5] Nvidia isaac lab. https://developer.nvidia.com/isaac/lab, 2026.
[6] Nvidia isaac sim. https://developer.nvidia.com/isaac/sim, 2026.
[7] So-101. https://huggingface.co/docs/lerobot/en/so101, 2026.
[8] R. Abhyankar, Z. He, V. Srivatsa, H. Zhang, and Y. Zhang. Infercept: Efficient intercept support for augmented large language model inference, 2024.
[9] Amazon. Amazon robotics. https://www.aboutamazon.com/news/operations/amazon-robotics-robots-fulfillment-center, 2024.
[10] Anthropic. The Claude 3 model family: Opus, Sonnet, Haiku. Technical report, 2024. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf.
[11] K. Black, M. Y. Galliker, and S. Levine. Real-time execution of action chunking flow policies, 2025.
[12] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, X. Chen, K. Choromanski, T. Ding, D. Driess, A. Dubey, C. Finn, P. Florence, C. Fu, M. G. Arenas, K. Gopalakrishnan, K. Han, K. Hausman, A. Herzog, J. Hsu, B. Ichter, A. Irpan, N. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, I. Leal, L. Lee, T.-W. E. Lee, S. Levine, Y. Lu, H. Michalewski, I. Mordatch, K. Pertsch, K. Rao, K. Reymann, M. Ryoo, G. Salazar, P. Sanketi, P. Sermanet, J. Singh, A. Singh, R. Soricut, H. Tran, V. Vanhoucke, Q. Vuong, A. Wahid, S. Welker, P. Wohlhart, J. Wu, F. Xia, T. Xiao, P. Xu, S. Xu, T. Yu, and B. Zitkovich. Rt-2: Vision-language-action models transfer web knowledge to robotic control, 2023.
[13] R. Cadene, S. Aliberts, F. Capuano, M. Aractingi, A. Zouitine, P. Kooijmans, J. Choghari, M. Russi, C. Pascal, S. Palma, M. Shukor, J. Moss, A. Soare, D. Aubakirova, Q. Lhoest, Q. Gallouédec, and T. Wolf. Lerobot: An open-source library for end-to-end robot learning, 2026.
[14] T. Chen, Z. Chen, B. Chen, Z. Cai, Y. Liu, Z. Li, Q. Liang, X. Lin, Y. Ge, Z. Gu, W. Deng, Y. Guo, T. Nian, X. Xie, Q. Chen, K. Su, T. Xu, G. Liu, M. Hu, H. ang Gao, K. Wang, Z. Liang, Y. Qin, X. Yang, P. Luo, and Y. Mu. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation, 2025.
[15] 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, 2024.
[16] N. Corporation. Tensorrt-llm: An open-source library to accelerate inference of large language models on nvidia gpus. https://github.com/NVIDIA/TensorRT-LLM, 2023. Library: TensorRT-LLM.
[17] H. Fang, Y. Liu, Y. Du, L. Du, and H. Yang. Sqap-vla: A synergistic quantization-aware pruning framework for high-performance vision-language-action models, 2025.
[18] Gemini Team, Google. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.
[19] I. Gim, S. seob Lee, and L. Zhong. Asynchronous llm function calling, 2024.
[20] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: a gpu cluster manager for distributed deep learning. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 485–500, USA, 2019. USENIX Association.
[21] P. B. Hansen. *Operating system principles.* Prentice-Hall, Inc., USA, 1973.
[22] P. Intelligence, K. Black, N. Brown, J. Darpinian, K. Dhabalia, D. Driess, A. Esmail, M. Equi, C. Finn, N. Fusai, M. Y. Galliker, D. Ghosh, L. Groom, K. Hausman, B. Ichter, S. Jakubczak, T. Jones, L. Ke, D. LeBlanc, S. Levine, A. Li-Bell, M. Mothukuri, S. Nair, K. Pertsch, A. Z. Ren, L. X. Shi, L. Smith, J. T. Springenberg, K. Stachowicz, J. Tanner, Q. Vuong, H. Walke, A. Walling, H. Wang, L. Yu, and U. Zhilinsky. $\pi_{0.5}$: a vision-language-action model with open-world generalization, 2025.
[23] T. Jiang, X. Jiang, Y. Ma, X. Wen, B. Li, K. Zhan, P. Jia, Y. Liu, S. Sun, and X. Lang. The better you learn, the smarter you prune: Towards efficient vision-language-action models via differentiable token pruning, 2025.
[24] W. Jiang, J. Clemons, K. Sankaralingam, and C. Kozyrakis. How fast can i run my vla? demystifying vla inference performance with vla-perf, 2026.
[25] M. J. Kim, Y. Gao, T.-Y. Lin, Y.-C. Lin, Y. Ge, G. Lam, P. Liang, S. Song, M.-Y. Liu, C. Finn, and J. Gu. Cosmos policy: Fine-tuning video models for visuomotor control and planning, 2026.
[26] M. J. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, G. Lam, P. Sanketi, Q. Vuong, T. Kollar, B. Burchfiel, R. Tedrake, D. Sadigh, S. Levine, P. Liang, and C. Finn. Openvla: An open-source vision-language-action model, 2024.
[27] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
[28] S.-W. Lee, X. Kang, and Y.-L. Kuo. Diff-dagger: Uncertainty estimation with diffusion policy for robotic manipulation, 2025.
[29] H. Li, Q. Mang, R. He, Q. Zhang, H. Mao, X. Chen, H. Zhou, A. Cheung, J. Gonzalez, and I. Stoica. Continuum: Efficient and robust multi-turn llm agent scheduling with kv cache time-to-live, 2026.
[30] P. Li, Y. Zhou, D. Muhtar, L. Yin, S. Yan, L. Shen, S. Vosoughi, and S. Liu. Diffusion language models know the answer before decoding, 2026.
[31] S. Li, Y. Gao, D. Sadigh, and S. Song. Unified video action model, 2025.
[32] X. Li, K. Hsu, J. Gu, K. Pertsch, O. Mees, H. R. Walke, C. Fu, I. Lunawat, I. Sieh, S. Kirmani, S. Levine, J. Wu, C. Finn, H. Su, Q. Vuong, and T. Xiao. Evaluating real-world robot manipulation policies in simulation, 2024.
[33] Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel, and M. Le. Flow matching for generative modeling, 2023.

[34] B. Liu, Y. Zhu, C. Gao, Y. Feng, Q. Liu, Y. Zhu, and P. Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning, 2023.

[35] Y. Liu et al. A survey of embodied AI in healthcare: Techniques, applications, and opportunities. *arXiv preprint arXiv:2501.07468*, 2025.

[36] Z. Liu, Y. Chen, H. Cai, T. Lin, S. Yang, Z. Liu, and B. Zhao. Vla-pruner: Temporal-aware dual-level visual token pruning for efficient vision-language-action inference, 2026.

[37] M. Luo, X. Shi, C. Cai, T. Zhang, J. Wong, Y. Wang, C. Wang, Y. Huang, Z. Chen, J. E. Gonzalez, and I. Stoica. Autellix: An efficient serving engine for llm agents as general programs, 2025.

[38] T. Ma, J. Zheng, Z. Wang, C. Jiang, A. Cui, J. Liang, and S. Yang. Dit4dit: Jointly modeling video dynamics and actions for generalizable robot control, 2026.

[39] M. Nuyens and A. Wierman. The foreground–background queue: A survey. *Performance Evaluation*, 65(3):286–307, 2008.

[40] NVIDIA, :, J. Bjorck, F. Castañeda, N. Cherniadev, X. Da, R. Ding, L. J. Fan, Y. Fang, D. Fox, F. Hu, S. Huang, J. Jang, Z. Jiang, J. Kautz, K. Kundalia, L. Lao, Z. Li, Z. Lin, K. Lin, G. Liu, E. Llontop, L. Magne, A. Mandlekar, A. Narayan, S. Nasiriany, S. Reed, Y. L. Tan, G. Wang, Z. Wang, J. Wang, Q. Wang, J. Xiang, Y. Xie, Y. Xu, Z. Xu, S. Ye, Z. Yu, A. Zhang, H. Zhang, Y. Zhao, R. Zheng, and Y. Zhu. Gr00t n1: An open foundation model for generalist humanoid robots, 2025.

[41] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[42] J. Pai, L. Achenbach, V. Montesinos, B. Forrai, O. Mees, and E. Nava. mimic-video: Video-action models for generalizable robot control beyond vlas, 2025.

[43] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023.

[44] S. Pohland, X. Foukas, G. Ananthanarayanan, A. Kolobov, S. Mehrotra, B. Radunovic, and A. Verma. Offload or overload: A platform measurement study of mobile robotic manipulation workloads, 2026.

[45] A. Prasad, K. Lin, J. Wu, L. Zhou, and J. Bohg. Consistency policy: Accelerated visuomotor policies via consistency distillation, 2024.

[46] Robots Guide. Sawyer robot. https://robotsguide.com/robots/sawyer, 2026.

[47] H. Shi, B. Xie, Y. Liu, L. Sun, F. Liu, T. Wang, E. Zhou, H. Fan, X. Zhang, and G. Huang. Memoryvla: Perceptual-cognitive memory in vision-language-action models for robotic manipulation, 2026.

[48] M. Shukor, D. Aubakirova, F. Capuano, P. Kooijmans, S. Palma, A. Zouitine, M. Aractingi, C. Pascal, M. Russi, A. Marafioti, S. Alibert, M. Cord, T. Wolf, and R. Cadene. Smolvla: A vision-language-action model for affordable and efficient robotics, 2025.

[49] 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, 2025.

[50] G. R. Team, S. Abeyruwan, J. Ainslie, J.-B. Alayrac, M. G. Arenas, T. Armstrong, A. Balakrishna, R. Baruch, M. Bauza, M. Blokzijl, S. Bohez, K. Bousmalis, A. Brohan, T. Buschmann, A. Byravan, S. Cabi, K. Caluwaerts, F. Casarini, O. Chang, J. E. Chen, X. Chen, H.-T. L. Chiang, K. Choromanski, D. D'Ambrosio, S. Dasari, T. Davchev, C. Devin, N. D. Palo, T. Ding, A. Dostmohamed, D. Driess, Y. Du, D. Dwibedi, M. Elabd, C. Fantacci, C. Fong, E. Frey, C. Fu, M. Giustina, K. Gopalakrishnan, L. Graesser, L. Hasenclever, N. Heess, B. Hernaez, A. Herzog, R. A. Hofer, J. Humplik, A. Iscen, M. G. Jacob, D. Jain, R. Julian, D. Kalashnikov, M. E. Karagozler, S. Karp, C. Kew, J. Kirkland, S. Kirmani, Y. Kuang, T. Lampe, A. Laurens, I. Leal, A. X. Lee, T.-W. E. Lee, J. Liang, Y. Lin, S. Maddineni, A. Majumdar, A. H. Michaely, R. Moreno, M. Neunert, F. Nori, C. Parada, E. Parisotto, P. Pastor, A. Pooley, K. Rao, K. Reymann, D. Sadigh, S. Saliceti, P. Sanketi, P. Sermanet, D. Shah, M. Sharma, K. Shea, C. Shu, V. Sindhwani, S. Singh, R. Soricut, J. T. Springenberg, R. Sterneck, R. Surdulescu, J. Tan, J. Tompson, V. Vanhoucke, J. Varley, G. Vesom, G. Vezzani, O. Vinyals, A. Wahid, S. Welker, P. Wohlhart, F. Xia, T. Xiao, A. Xie, J. Xie, P. Xu, S. Xu, Y. Xu, Z. Xu, Y. Yang, R. Yao, S. Yaroshenko, W. Yu, W. Yuan, J. Zhang, T. Zhang, A. Zhou, and Y. Zhou. Gemini robotics: Bringing ai into the physical world, 2025.

[51] Tesla. Tesla optimus. https://www.tesla.com/optimus, 2024.

[52] Z. Wang, Z. Li, A. Mandlekar, Z. Xu, J. Fan, Y. Narang, L. Fan, Y. Zhu, Y. Balaji, M. Zhou, M.-Y. Liu, and Y. Zeng. One-step diffusion policy: Fast visuomotor policies via diffusion distillation, 2024.

[53] Y. Xu, X. Kong, T. Chen, and D. Zhuo. Conveyor: Efficient tool-aware llm serving with tool partial execution, 2024.

[54] Y. Xu, Y. Yang, Z. Fan, Y. Liu, Y. Li, B. Li, and Z. Zhang. Qvla: Not all channels are equal in vision-language-action model's quantization, 2026.

[55] H. Yan, Z. Zhong, J. Zhu, J. He, W. Yuan, W. Song, X. Gong, Y. Cai, G. Zhao, X. Yan, B. Liu, Y.-C. Chen, and H. Li. S-vam: Shortcut video-action model by self-distilling geometric and semantic foresight, 2026.

[56] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.

[57] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models, 2023.

[58] A. Ye, B. Wang, C. Ni, G. Huang, G. Zhao, H. Li, H. Li, J. Li, J. Lv, J. Liu, M. Cao, P. Li, Q. Deng, W. Mei, X. Wang, X. Chen, X. Zhou, Y. Wang, Y. Chang, Y. Li, Y. Zhou, Y. Ye, Z. Liu, and Z. Zhu. Gigaworld-policy: An efficient action-centered world–action model, 2026.

[59] H. Ye, J. Yuan, R. Xia, X. Yan, T. Chen, J. Yan, B. Shi, and B. Zhang. Training-free adaptive diffusion with bounded difference approximation strategy, 2024.

[60] S. Ye, Y. Ge, K. Zheng, S. Gao, S. Yu, G. Kurian, S. Indupuru, Y. L. Tan, C. Zhu, J. Xiang, A. Malik, K. Lee, W. Liang, N. Ranawaka, J. Gu, Y. Xu, G. Wang, F. Hu, A. Narayan, J. Bjorck, J. Wang, G. Kim, D. Niu, R. Zheng, Y. Xie, J. Wu, Q. Wang, R. Julian, D. Xu, Y. Du, Y. Chebotar, S. Reed, J. Kautz, Y. Zhu, L. J. Fan, and J. Jang. World action models are zero-shot policies, 2026.

[61] T. Yu, D. Quillen, Z. He, R. Julian, A. Narayan, H. Shively, A. Bellathur, K. Hausman, C. Finn, and S. Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning, 2021.

[62] T. Yuan, Z. Dong, Y. Liu, and H. Zhao. Fast-wam: Do world action models need test-time future imagination?, 2026.

[63] K. Zhang, M. Sharma, J. Liang, and O. Kroemer. A modular robotic arm control stack for research: Franka-interface and frankapy, 2020.

[64] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn. Learning fine-grained bimanual manipulation with low-cost hardware, 2023.

[65] J. Zheng, J. Li, Z. Wang, D. Liu, X. Kang, Y. Feng, Y. Zheng, J. Zou, Y. Chen, J. Zeng, Y.-Q. Zhang, J. Pang, J. Liu, T. Wang, and X. Zhan. X-vla: Soft-prompted transformer as scalable cross-embodiment vision-language-action model, 2025.

[66] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng. Sglang: Efficient execution of structured language model programs, 2024.

[67] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig. WebArena: A realistic web environment for building autonomous agents. In *ICLR*, 2024.