PHYAI: A UNIFIED INFERENCE RUNTIME FOR ROBOT POLICIES ACROSS EDGE AND CLOUD

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent breakthroughs in foundational models have driven significant progress in Robot Learning Models (RLMs), spanning Vision-Language-Action (VLA) and World Action Model (WAM) architectures. However, unlike the mature serving ecosystems for LLMs, RLM inference infrastructure remains highly fragmented across diverse deployment scales, from batch-1 real-time device control to massive batch-N edge serving, evaluationa and RL post-training rollouts. We present PhyAI, a unified, high-performance inference runtime for RLMs. By cleanly decoupling model semantics from execution, PhyAI dynamically adapts its parallelism (e.g., unbatched, DP, TP) and kernel dispatch strategies to the target hardware, seamlessly bridging edge and cloud environments. Through system-level optimizations including CUDA graph replay, kernel fusion, and state reuse, PhyAI achieves 1.40× to 4.65× latency speedups over official implementations across 11 benchmark configurations (anonymous real-robot demo[1]). To investigate how these inference speedups translate to physical control frequencies, we introduce the *Control Rate Roofline* model, demonstrating that our optimizations either directly boost control rates or unlock time margins for deploying larger models.

1 INTRODUCTION

Recent advances in large language models (LLMs) and vision-language models (VLMs), driven by continued scaling, have catalyzed significant progress in robot learning models (RLMs). By leveraging pretrained Vision-Language-Action (VLA) models or World Action Models (WAMs) backbones trained on large-scale robot manipulation datasets, researchers have begun scaling RLMs to demonstrate emergent generalization, enabling end-to-end learning of robot control policies.

Limitations of existing inference infrastructures. Despite rapid algorithmic progress, the development of dedicated inference infrastructure for RLMs has fallen far behind:

First, *standard LLM serving frameworks are fundamentally mismatched with RLM workloads.* Existing LLM engines (e.g., vLLM Kwon et al. (2023) and SGLang Zheng et al. (2024)) rely heavily on KV-cache management and autoregressive decoding. In contrast, RLM inference, which often employs Diffusion Transformers (DiT) for iterative action denoising—operates as a stateless, prefill, only process that requires no historical KV cache.

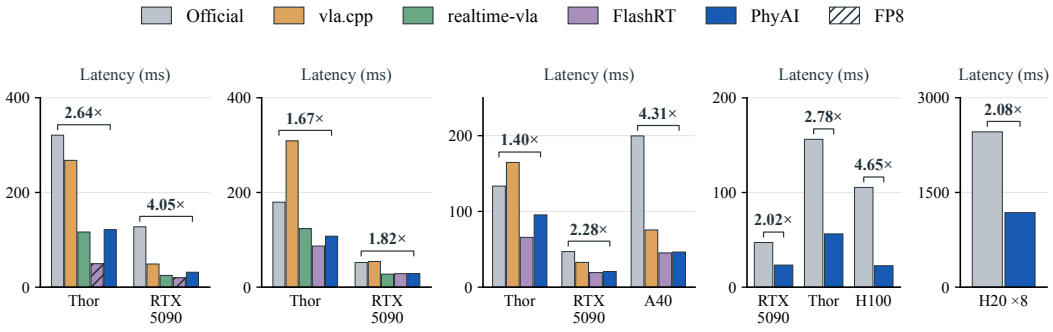
Second, *existing RLM infrastructures Cadene et al. (2026); StarVLA Community (2026); Yu et al. (2026); Choi et al. (2026); Su (2026); Nguyen et al. (2026); Ma et al. (2025); Xu et al. (2026) cannot support full-scenario deployments or diverse model architectures.* This manifests in two ways: (i) *Deployment diversity:* RLM workloads span four distinct regimes: real-time on-device control (batch-1), multi-request edge serving (dynamic batch-N), alongside cloud-based evaluation and high-throughput RL rollouts (massive batch-N). Bridging these scales requires a unified runtime that dynamically adapts its parallelism (e.g., unbatched, DP, TP) to the target environment, a capability missing in current rigid frameworks. (ii) *Architectural diversity:* The coexistence of distinct paradigms, such as VLA models and WAMs, introduces highly heterogeneous computational graphs. WAMs often employ cascaded or concurrent world-action branching, diverging from the direct observation-to-action mapping of standard VLAs. Rigid runtimes fail to accommodate these complex, dynamic state-update mechanisms and memory reuse patterns.

[1] https://anonymous.4open.science/r/phyai-iclr-2027-page-A8CC/

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



Figure 1: PhyAI is a shared inference foundation designed to support diverse RLM workloads consistently across a wide set of hardware targets.



(a) π_0 **(b)** $\pi_{0.5}$ **(c)** GR00T N1.7 **(d)** MiniCPM-Robot **(e)** Cosmos3

Figure 2: Latency comparison between PhyAI and existing inference infrastructures. "Official" refers to the opensource inference code provided by the model providers (e.g., OpenPI for π0.5).

We introduce **PhyAI**, an open-source inference framework for RLMs that provides a unified, high-performance runtime for evaluation, RL rollout, and robot deployment (Figure 1). It scales inference in cloud while supporting low-latency execution on edge devices, delivering two core capabilities:

- **Flexible kernel dispatch and scalable serving.** PhyAI cleanly decouples model semantics from execution. It features an affinity rule-based kernel dispatcher that dynamically adapts to hardware, numerical precision, and tensor shapes at runtime, automatically selecting the optimal kernel implementations and fusion strategies. Furthermore, through a unified module interface, the engine automatically orchestrates configurable parallelism (e.g., DP, TP, CFG), seamlessly scaling from batch-one edge inference to distributed, high-throughput cloud workloads.
- **Optimized inference speed.** By integrating advanced system optimizations, such as kernel fusion, CUDA graph replay, and cache precomputation, PhyAI drastically reduces launch overhead and memory traffic. Across 11 benchmark configurations, it achieves 1.40× to 4.65× latency speedups over official implementations, as shown in Figure 2.

Beyond achieving raw inference speedups, a critical systems question remains: *how does accelerating policy inference translate to real-world control frequencies?* To answer this, we introduce the *Control Rate Roofline* model based on Real-Time Chunking (RTC) Black et al. (2025c). Our analysis reveals two distinct operational regimes: in inference-bound scenarios (e.g., edge deployments), PhyAI's speedups directly elevate the robot's control rate; in environment-bound scenarios, where mechanical speed dictates the ceiling, our framework unlocks valuable timing margins that can be repurposed for deploying larger models or handling concurrent requests.

Our main contributions can be summarized as follows:

- We identify the extreme deployment and architectural diversity and propose PhyAI, a unified inference runtime seamlessly bridging edge and cloud environments.
- We achieve up to 4.65× latency speedups through system-level optimizations. Real-robot deployement shows PhyAI's effectiveness.
- We propose *Control Rate Roofline* model to characterize how system acceleration translates to physical control frequencies.

Table 1: Comparison of PhyAI with other representative inference infrastructures.

Framework	VLA	WAM	Quantization	Deployment	Parallelism
PhyAI (ours)	✓	✓	✓	Edge + Cloud	DP/TP/CFG
FlashRT	✓	✓	✓	Edge + Cloud	–
vla.cpp	✓	–	–	Edge	–
realtime-vla	✓	–	–	Edge	–

2 BACKGROUND

2.1 ROBOT LEARNING MODELS

Current RLMs primarily encompass two core architectures: VLA models and WAMs:

VLA Models map multimodal inputs directly to robot actions. While early systems (e.g., RT-1 (Brohan et al., 2023), OpenVLA (Kim et al., 2025)) established the basic pipeline, modern architectures (e.g., Octo (Ghosh et al., 2024), the π series (Black et al., 2025b;a; Physical Intelligence et al., 2026)) increasingly adopt Diffusion Transformer (DiT)-style inference and diverse action-head couplings (NVIDIA et al., 2025; Wu et al., 2026; Li et al., 2024; Yang et al., 2026; Dang et al., 2026; Li et al., 2026a). This diversity imposes distinct system requirements for state reuse and scheduling.

WAMs augment action prediction by explicitly modeling future states or latent dynamics (Wang et al., 2026). Like advanced VLAs, modern WAMs predominantly rely on DiT-based iterative denoising. Whether generating states and actions jointly (e.g., Cosmos3 (NVIDIA, 2026), DreamZero (Ye et al., 2026), and others (Zhang et al., 2026b; Cen et al., 2025)) or exploring specialized branches (Liu et al., 2026; Cai et al., 2026; Yuan et al., 2026), WAMs introduce larger evolving states and more intricate generation dependencies than standard VLAs.

The Need for RLM Inference Systems. Standard LLM serving frameworks (e.g., vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024)) are optimized for autoregressive decoding and historical KV-cache management. This prefill-decode abstraction fundamentally mismatches RLMs: mainstream VLAs and WAMs rely on DiT denoising (resembling continuous prefill without AR decoding) and typically execute stateless, single-step inference at each control cycle (e.g., $\pi_{0.5}$, Cosmos3). These architectural shifts, alongside strict real-time deadlines for repeated observation processing, demand a novel inference infrastructure tailored for physical AI.

2.2 PHYSICAL AI INFRASTRUCTURE

Training and evaluation frameworks. Existing RLM infrastructures primarily serve training and evaluation. Systems like LeRobot (Cadene et al., 2026), StarVLA (StarVLA Community, 2026), and RLinf (Yu et al., 2026) focus on training workflows, data management, and RL scheduling, while vla-eval (Choi et al., 2026) targets unified cross-benchmark evaluation. However, none provides a unified inference infrastructure for multi-scenario deployment.

Edge inference runtimes. Existing runtimes target single-device execution. Frameworks like FlashRT (Su, 2026), vla.cpp (Nguyen et al., 2026), realtime-vla (Ma et al., 2025), and Embodied.cpp (Xu et al., 2026) optimize real-time control, portability, or heterogeneous edge deployment. Yet, none offers a unified scaling mechanism for cloud-side multi-GPU execution. Table 1 compares PhyAI with these systems across key capabilities.

Efficiency optimizations. Recent work has developed effective techniques for improving deployment efficiency for VLAs and WAMs, including quantization, pruning, cache reuse, and asynchronous execution (Zhang et al., 2026a; Akbari et al., 2026; Yang et al., 2025; Liu et al., 2025; Li et al., 2026b; Xu et al., 2025; Oi et al., 2026; Zhao et al., 2026; Black et al., 2025c). However, these methods are typically built and evaluated within isolated codebases, each tied to specific model architectures, hardware, and precision settings. This makes it hard to combine them, reproduce results, or compare them fairly.

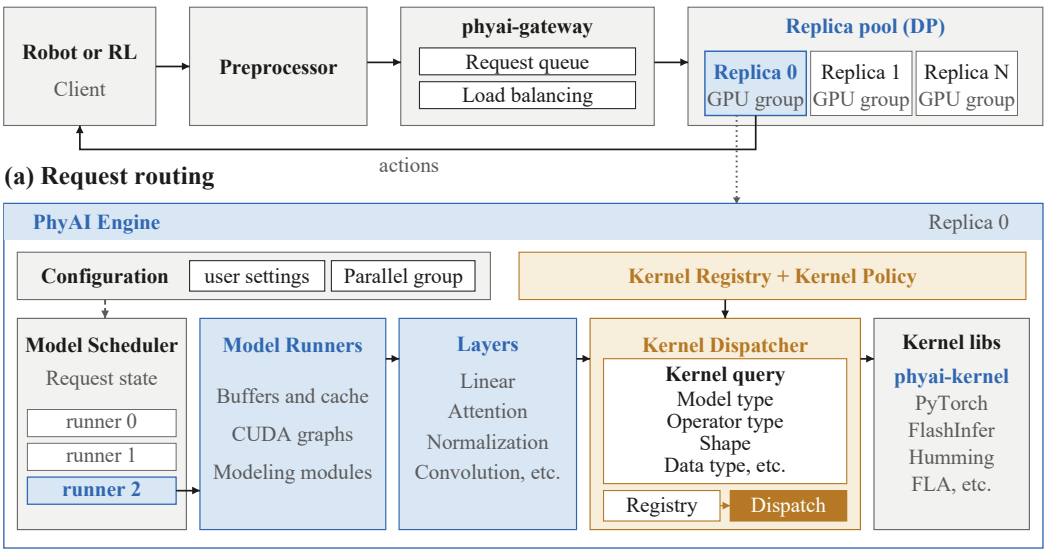


Figure 3: PhyAI architecture. (a) Requests are routed to one replica in a replica pool. (b) The replica schedules model runners, shared Layers, and kernel dispatch.

2.3 Challenges in RLM Inference Infrastructure

The primary challenge is supporting diverse RLMs across on-device, edge, and cloud environments without fragmenting the software stack into separate, ad-hoc implementations for each scenario.

- **Deployment diversity.** Different execution environments impose conflicting constraints. On-device inference demands ultra-low latency under strict memory and compute limits. Edge servers must concurrently handle requests from multiple robots, whereas cloud-based evaluation and reinforcement learning (RL) rollouts rely on heavy batching and distributed execution to maximize throughput. Consequently, optimal scheduling and parallelism configurations vary drastically across deployments, even for the identical model.
- **Model architecture diversity.** VLAs and WAMs exhibit highly heterogeneous computational graphs and state-update mechanisms. Structurally, these models organize computation in diverse ways, such as employing cascaded versus concurrent world-action branching in WAMs. Temporally, the lifespan and update frequency of intermediate tensors vary significantly during generation. For example, within a DiT-based cycle, visual observation embeddings typically remain static across denoising steps (enabling state reuse), whereas action latents are updated iteratively. Together, these structural and temporal dependencies dictate how a shared runtime can safely reuse memory, fuse operators, or capture execution graphs (e.g., via CUDA graphs), making a rigid execution strategy ineffective.

3 PhyAI Inference Infrastructure

3.1 Architecture Overview

PhyAI separates model semantics from execution mechanisms to improve component reuse (Figure 3). Each model has a dedicated Scheduler that coordinates its *Model Runners*. Each Model Runner encapsulates the corresponding model submodule, maintains execution state such as buffers and metadata, and invokes operators through the shared Layers interface.

PhyAI addresses deployment diversity and model architecture diversity through two capabilities: (i) flexible parallel group configuration for different models and (ii) kernel selection and dispatch for different deployment settings. The engine creates parallel groups from the execution configuration,

```python
Impl(
    kernel_id="flashinfer.rmsnorm_add",
    op="rmsnorm_add",
    priority=Priority.OPTIMIZED + 2,
    when=all_of(device.sm >= 90,
                dtype.residual.in_({"bf16",
      "fp16"})),
    prepare=_flashinfer_rmsnorm_add,
),
# PyTorch fallback omitted for brevity...
```

(a) Kernel Registration (Python)

```yaml
rules:
  - id: sm90-fused-rmsnorm
    priority: 90
    match:
      op: rmsnorm_add
      device: {vendor: nvidia, arch: ">=
       sm90"}
    prefer:
      - flashinfer.rmsnorm_add
      - phyai_kernel.rmsnorm_add
```

(b) Deployment Policy (YAML)

Figure 4: An example of rule-based kernel dispatch for `rmsnorm_add`. The model logic remains entirely decoupled from these hardware-specific configurations. (Full example in Appendix B)

which selects among the parallelism modes and degrees supported by the model. The Scheduler and Distributed Layers organize computation and communication within these groups. The kernel dispatcher selects compatible kernels based on the device, data type, tensor shape, and execution mode, then routes operator requests from Layers to the selected implementations.

3.2 Kernel dispatcher

As the number of models and devices grows, hard-coded kernel selection becomes harder to maintain. Compatibility checks, precision settings, and backend priorities add branches to model code. These branches complicate debugging and tie each model implementation to specific kernels. To address this, PhyAI delegates this choice to a *rule-based kernel dispatcher*. As illustrated in Figure 4, developers simply register kernel capabilities in Python (e.g., hardware constraints for `rmsnorm_add`), while a separate YAML policy dynamically routes execution based on runtime contexts (e.g., pinning a specific kernel during CUDA graph capture). Therefore, model modules only need to state *what* to compute, while the dispatcher dynamically decides *how* to compute it.

We treat dispatch as constrained selection. Let q be one operator request and $\mathcal{K}(o)$ the implementations registered for operator o. Each implementation k declares a capability condition $C_k(q)$. A policy P supplies an admissibility constraint $A_P(k, q)$ and a priority $\rho_P(k, q)$. The dispatcher selects

$$k^\star(q; P) = \arg\max_{k \in \mathcal{K}(o)} \rho_P(k, q) \quad \text{s.t.} \quad C_k(q) = 1,\ A_P(k, q) = 1. \tag{1}$$

The query q specifies the model, operator, device, tensor shape, data type, and execution mode. The dispatcher uses the capability conditions $C_k(q)$ recorded in the Registry to identify compatible kernels. The Policy then filters these candidates through $A_P(k, q)$ and ranks them by $\rho_P(k, q)$. During graph capture, the dispatcher also excludes implementations that violate capture constraints. It caches the selected implementation for calls with the same query signature.

Models and kernels then evolve independently. A new model describes computation through shared Layers, and a new kernel registers capability and priority. Neither side needs to hard-code kernel choices in model code.

3.3 Parallel execution and scheduling

Different model architectures necessitate distinct parallel execution strategies. For compact models like $\pi_{0.5}$, which can be executed entirely on a single device, PhyAI employs data parallelism (DP) to scale throughput across independent requests. Conversely, for larger architectures like Cosmos3, the system combines tensor parallelism (TP) and classifier-free guidance (CFG) parallelism, which executes conditional and unconditional branches simultaneously, to minimize inference latency. Ultimately, the model's intrinsic structure dictates the feasible parallelism strategies, whereas available deployment resources and specific performance targets determine the optimal degree of parallelism.

PhyAI implements these strategies across and within replicas. The gateway implements DP by routing independent requests to complete model replicas. Within a replica, Distributed Layers provide the sharded operators and collectives required for TP. The model Scheduler coordinates CFG branch execution and combines the outputs. The runtime creates parallel groups from the execution configuration, and Model Runners execute in the corresponding parallel contexts. Compared with

runtimes focused on single-device inference, PhyAI supports replica scaling and parallelism within replicas through a common execution path. This allows the same model implementation to support edge inference, cloud evaluation, and RL rollout.

3.4 Optimization methods

PhyAI includes the following optimizations:

Kernel fusion. For small batches, kernel launches and intermediate memory traffic can dominate the cost of short GPU operators. We fuse operations with compatible tensor contracts to reduce these costs. Our implementations include fused AdaRMSNorm and residual addition with RMSNorm. We also combine the QKV projections and the MLP gate and up projections, and fuse elementwise activation with multiplication where these patterns occur. On H20, enabling fusion on top of graph replay raises MiniCPM-RobotManip throughput from 33.28 to 36.77 Hz, a further 10.5%.

Graph replay. Denoising and solver loops often repeat the same GPU operations with unchanged tensor shapes and control flow. Dispatching each operation from the host adds launch overhead at every step. We use CUDA Graph replay to reduce this repeated dispatch. The plugin declares how long cached state remains valid, and the runner retains conditioning features, attention metadata, and index arrays within that scope. After warmup stabilizes buffer addresses, the runner captures the matching graph bucket and replays it with updated buffer contents. On H20, graph replay raises MiniCPM-RobotManip throughput from 10.12 to 33.28 Hz, a $3.29\times$ speedup.

Cache precomputation. Computing timestep embeddings and layer modulation inside the denoising loop adds per-step overhead. When these values follow a known schedule, the runner precomputes the sinusoidal timestep and AdaRMS modulation tables before the loop. Subsequent steps read these entries from the request cache, including during graph replay.

4 Control Rate Roofline

PhyAI provides a unified, optimized runtime for executing RLMs across deployment settings. This allows us to investigate a further systems question: *how far can inference acceleration improve the control rate, and what sets the remaining limit?* We study this question for the complete control loop, with the deployment and action execution window held fixed.

To quantify how inference speedup affects control rate, we decompose the total request latency into inference and non-inference components:

$$L_{\mathrm{req}} = L_{\mathrm{obs}} + 2L_{\mathrm{comm}} + L_{\mathrm{queue}} + L_{\mathrm{inf}}. \tag{2}$$

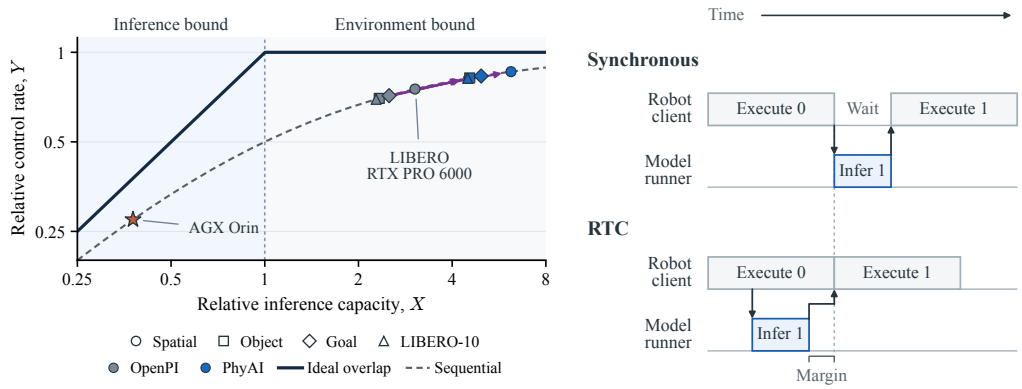
Here L_{obs} covers observation capture and preprocessing, L_{queue} is the batching and scheduling wait, and L_{inf} is the model inference time. L_{comm} are the latencies of request delivery and action return, including controller handoff.

Reducing L_{inf} shortens the critical path of a request, but the overall control rate also heavily depends on the overlap between request processing and environment execution. Traditionally, these two phases execute sequentially, leading to accumulated delays. Recently, however, Real-Time Chunking (RTC) (Black et al., 2025c) has been widely adopted to minimize execution wait times by predicting the next action chunk while the current one is actively executing (Figure 5b). Consequently, given a request latency L_{req} and an environment execution time T_{env}, the end-to-end physical execution time T under ideal RTC overlap is formulated as:

$$T = \max\left(L_{\mathrm{req}}, T_{\mathrm{env}}\right). \tag{3}$$

The control rate is $T_{\mathrm{ctrl}} = 1/T$, where T_{ctrl} is the period under the selected schedule. With non-inference costs fixed, reducing L_{inf} decreases T_{seq}. Under ideal overlap, inference acceleration raises the rate while $L_{\mathrm{req}} > T_{\mathrm{env}}$; once $L_{\mathrm{req}} \leq T_{\mathrm{env}}$, the environment sets the ceiling and further acceleration adds timing margin. That margin can support a larger model or more concurrent requests.

Based on our evaluation across diverse hardware platforms (e.g., AGX Orin and RTX PRO 6000) and frameworks, the experimental timing results are summarized in Figure 5a. To isolate the core computational bottlenecks, this comparison strictly accounts for model inference and environment



(a) Control Rate Roofline **(b)** RTC scheduling

Figure 5: Control rate and scheduling. **(a)** Normalized inference capacity and control rate for four LIBERO suites and the AGX Orin aggregate. Dashed and solid curves show sequential execution and ideal overlap, respectively. **(b)** Comparison of synchronous execution and RTC.

execution, intentionally omitting observation, communication, and queueing overheads. We normalize these timings to map the measured workloads onto a unified roofline model, defining the relative inference capacity as $X = T_{\text{env}}/L_{\text{inf}}$ and the relative control rate as $Y = T_{\text{env}} * T_{\text{ctrl}}$.

This normalized view clearly delineates two distinct performance regimes. When $X < 1$ (i.e., model inference takes longer than physical execution), the system is **inference-bound**. The AGX Orin aggregate, for instance, operates in this regime at $X \approx 0.38$, indicating that accelerating inference will directly raise its control rate ceiling. Conversely, when $X \geq 1$, the system becomes **environment-bound**. As shown in Figure 5a, both OpenPI and PhyAI demonstrate this across all four LIBERO suites ($X > 1$), meaning their ideal overlap ceiling is strictly dictated by the robot's physical execution speed rather than computation. Under this mathematical formulation, traditional sequential execution yields a baseline curve of $Y = X/(1 + X)$, whereas ideal RTC overlap establishes a theoretical roofline of $Y = \min(X, 1)$. As illustrated, our plotted empirical data points currently align with the sequential execution baseline.

Control Rate Roofline. Inference is not always the bottleneck in robot control. Under ideal overlap, once the complete request fits within the environment execution window, further inference acceleration adds timing margin without increasing the control rate.

5 Evaluation

5.1 Experimental setup

We evaluate PhyAI's inference performance across VLA and world action models on edge and server GPUs. We test π_0, $\pi_{0.5}$, GR00T N1.7, MiniCPM-Robot, and Cosmos3-Nano-Policy. The experiments measure inference latency, characterize batch-dependent bottlenecks, evaluate DP scaling, and assess performance in an RL workflow. Configurations are reported with each experiment.

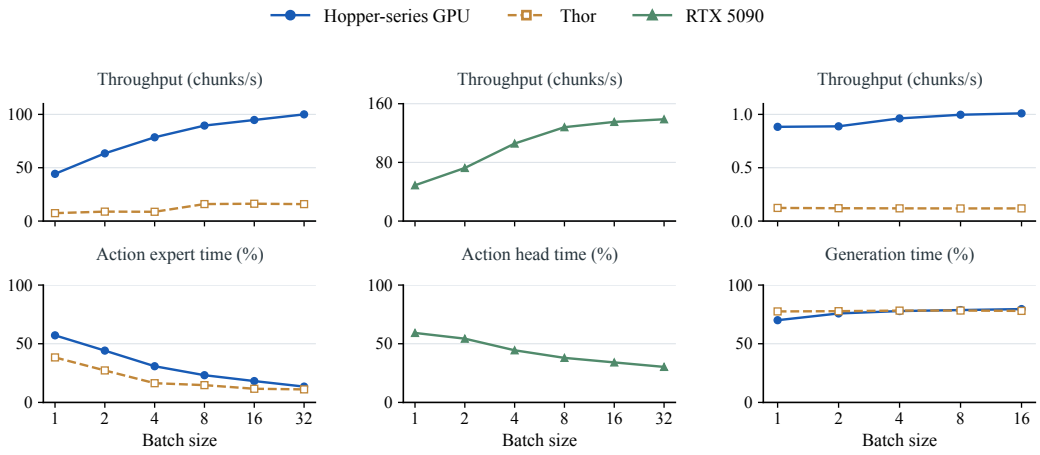
Unless stated otherwise, latency is the model-runner time for one complete action chunk after warm-up. It excludes observation capture, transport, queueing, and action handoff. We assess downstream behavior by comparing $\pi_{0.5}$ with OpenPI on four LIBERO suites, using 500 episodes per suite. Appendix A provides the simulation and record-validation protocols and the behavioral results.

5.2 Latency

On Thor, PhyAI accelerates π_0, $\pi_{0.5}$, GR00T, and MiniCPM-Robot by $2.64\times$, $1.67\times$, $1.40\times$, and $2.78\times$, respectively, over the official implementations. On RTX 5090, the corresponding speedups are $4.05\times$, $1.82\times$, $2.28\times$, and $2.02\times$ (Table 2).

Table 2: Model-runner latency relative to the official path. Lower values are better; speedup is Official divided by PhyAI.

Model	Device	Official (ms)	PhyAI (ms)	Speedup
π_0	Thor	321.0	121.5	**2.64**×
π_0	RTX 5090	127.5	31.5	**4.05**×
$\pi_{0.5}$	Thor	179.5	107.7	**1.67**×
$\pi_{0.5}$	RTX 5090	52.1	28.6	**1.82**×
GR00T N1.7	Thor	133.5	95.4	**1.40**×
GR00T N1.7	RTX 5090	47.0	20.6	**2.28**×
GR00T N1.7	A40	199.7	46.3	**4.31**×
MiniCPM-Robot	RTX 5090	47.2	23.3	**2.02**×
MiniCPM-Robot	Thor	156.2	56.3	**2.78**×
MiniCPM-Robot	H100	105.4	22.6	**4.65**×
Cosmos3-Nano-Policy	H20 × 8	2460.0	1180.0	**2.08**×



(a) $\pi_{0.5}$ **(b)** GR00T N1.7 **(c)** Cosmos3

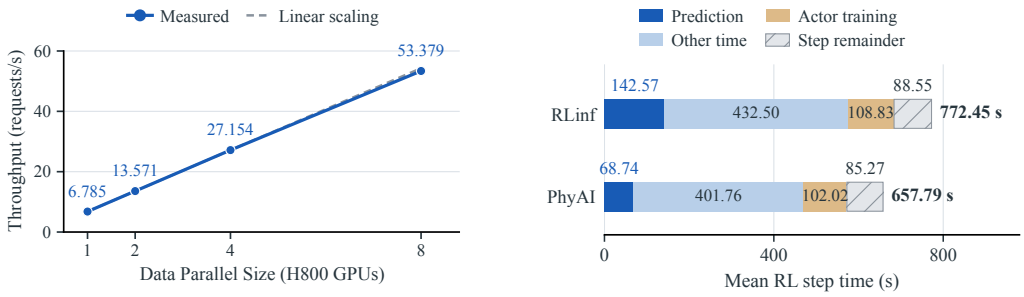
Figure 6: Batch scaling and phase profiles. Top: action-chunk throughput. Bottom: the indicated phase's share of summed GPU phase time, measured separately from full-batch latency. Throughput means use 50, 30, and 5 timed iterations after 10, 8, and 2 warm-ups for $\pi_{0.5}$, GR00T, and Cosmos3.

These speedups reflect the optimizations in Section 3.4: kernel fusion reduces launches and intermediate memory traffic, graph replay reduces host dispatch, and cache precomputation avoids repeated setup. The kernel dispatcher supports these deployments by selecting compatible implementations through the shared Layers interface (Section 3.2), so each model can use kernels suited to the target hardware. The same runtime also accelerates inference on server GPUs, with a $4.31\times$ speedup for GR00T on A40 and a $4.65\times$ speedup for MiniCPM-Robot on H100.

For distributed inference, the parallel runtime in Section 3.3 supports Cosmos3 with TP=4 and CFG=2 on eight H20 GPUs. In this configuration, PhyAI reduces latency from 2.46 to 1.18 s, a $2.08\times$ speedup over the official implementation using the same device count.

5.3 Batch Scaling and Model Bottlenecks

We profile static BF16 batches from 1 to 32 for $\pi_{0.5}$ and GR00T, and from 1 to 16 for Cosmos3 (Figure 6). The VLA runners use CUDA Graphs, while Cosmos3 runs eagerly. Measurements exclude batch formation and queueing.



(a) DP scaling **(b)** RL step time

Figure 7: DP scaling and RL step times for $\pi_{0.5}$. (a) Aggregate throughput on one to eight H800 GPUs, with one replica per GPU. The dashed line shows linear scaling from one replica. (b) Mean RL step times for RLinf and PhyAI on four H800 GPUs.

On Hopper, a GPU Roofline estimate based on modeled tensor traffic places $\pi_{0.5}$'s VLM on the compute-bound side and its action expert on the memory-bound side at batch size 1. The expert processes short action sequences through small matrix operations with low compute utilization. Batching improves its efficiency, shifting the dominant cost to the VLM.

For GR00T on RTX 5090, batching improves Action Head utilization, while Backbone becomes the dominant cost by batch size 4. Backbone utilization levels off around batch size 8, where overall throughput already reaches 92.2% of the throughput at batch size 32. Further batching therefore mainly increases full-batch latency.

Cosmos3's long video and action sequences already yield high generation compute utilization at batch size 1. Its generation phase lies on the compute-bound side of the same Roofline estimate, leaving limited gains from batching. On Hopper, increasing batch size from 1 to 16 improves throughput by only 14.3%, while full-batch latency reaches 15.8 s.

We separately test the DP execution described in Section 3.3, with one complete $\pi_{0.5}$ replica per H800 GPU. Increasing the replica count from one to eight raises throughput from 6.785 to 53.379 requests/s (Figure 7a). The resulting 7.87× gain reaches 98.3% of the linear reference.

5.4 RL ROLLOUT

We profile $\pi_{0.5}$ using PPO with GAE in RLinf (Yu et al., 2026) on four H800 GPUs across 64 LIBERO-Spatial environments (configured with 3 denoising steps, 5 actions per query, and 1 actor-update epoch). Replacing RLinf's default rollout backend with PhyAI, we average measurements over four RL steps, excluding warmup, validation, and checkpointing.

PhyAI reduces the mean prediction time (maximum across concurrent ranks) from 142.57 s to 68.74 s, achieving a 2.07× speedup. In the baseline, prediction accounts for 24.8% of the rollout and 18.5% of the full RL step. Consequently, PhyAI reduces the total RL step time from 772.45 s to 657.79 s, yielding a 1.17× end-to-end speedup (Figure 7b). The overall gain is primarily bottlenecked by non-inference overheads, such as environment interaction, orchestration, and actor training—which see only marginal reductions.

6 CONCLUSION

PhyAI is a unified inference runtime for RLMs that decouples model semantics from execution policies, seamlessly supporting edge-to-cloud deployments, evaluation, and RL rollouts. It significantly outperforms official implementations across diverse benchmarks and accelerates policy inference during RL training. Furthermore, we introduce the *Control Rate Roofline* model, revealing that inference speedups increase control rates when inference dominates, but primarily expand timing margins when environment interaction dominates. PhyAI have been open-sourced to advance RLM infrastructure research.

AI USE STATEMENT

Generative AI tools assisted in implementing PhyAI's kernel components and running precision alignment workflows. They also aided in translation, figure preparation, and manuscript refinement to improve readability.

These tools did not contribute to the analysis or the experiments. The Control Rate Roofline, the system design, and the interpretation of results are our own work. We manually confirmed that PhyAI's accuracy on the downstream tasks is correct.

We have reviewed all AI-assisted work. The authors inspected the kernel and runtime changes, compared the precision alignment outputs with the reference implementations, checked the translations against the source text, and verified the figures against the measurement data. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Appendix A specifies simulator interfaces, episode counts, action handling, and matched evaluation settings for the models that report downstream results. Appendix C gives additional RL configuration and profiling procedures. Appendix B provides a kernel registry and policy example. The runtime includes benchmark and test suites. The figure-source tables list the values used in the figures. Experiments require matching checkpoints, normalization assets, and simulator or trajectory data. We will open-source the PhyAI runtime, the benchmark harness, and the figure-source tables so that these procedures can be rerun.

REFERENCES

Arash Akbari, Arman Akbari, Masih Eskandar, Qitao Tan, Yixiao Chen, Jingwu Luo, Bertha Pangaribuan, Liyun Zhang, Jennifer Dy, Geng Yuan, Xue Lin, Gaowen Liu, Stratis Ioannidis, and Yanzhi Wang. ActQuant: Sub-4-bit Action-Guided Quantization for Vision-Language-Action Models. *arXiv preprint arXiv:2605.24011*, 2026.

Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: a Vision-Language-Action Model with Open-World Generalization. In Joseph Lim, Shuran Song, and Hae-Won Park (eds.), *Proceedings of The 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pp. 17–40, 27–30 Sep 2025a.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, Laura Smith, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A Vision-Language-Action Flow Model for General Robot Control. In *Proceedings of Robotics: Science and Systems*, June 2025b.

Kevin Black, Manuel Galliker, and Sergey Levine. Real-Time Execution of Action Chunking Flow Policies. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 33383–33407, 2025c.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alexander Herzog, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Tomas Jackson, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Kuang-Huei Lee, Sergey Levine, Yao Lu, Utsav Malla, Deeksha Manjunath, Igor Mordatch, Ofir Nachum, Carolina Parada, Jodilyn Peralta, Emily Perez, Karl Pertsch, Jornell Quiambao, Kanishka Rao, Michael S Ryoo, Grecia Salazar, Pannag R Sanketi,

Kevin Sayed, Jaspiar Singh, Sumedh Sontakke, Austin Stone, Clayton Tan, Huong Tran, Vincent Vanhoucke, Steve Vega, Quan H Vuong, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. RT-1: Robotics Transformer for Real-World Control at Scale. In *Proceedings of Robotics: Science and Systems*, July 2023.

Remi Cadene, Simon Alibert, Francesco Capuano, Michel Aractingi, Adil Zouitine, Pepijn Kooijmans, Jade Choghari, Martino Russi, Caroline Pascal, Steven Palma, Dana Aubakirova, Mustafa Shukor, Jess Moss, Alexander Soare, Quentin Lhoest, Quentin Gallouédec, and Thomas Wolf. LeRobot: An Open-Source Library for End-to-End Robot Learning. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 122398–122417, 2026.

Jisong Cai, Long Ling, Shiwei Chu, Zhongshan Liu, Jiayue Kang, Zhixuan Liang, Wenjie Xu, Yinan Mao, Weinan Zhang, Xiaokang Yang, Ru Ying, Ran Zheng, and Yao Mu. AHA-WAM: Asynchronous Horizon-Adaptive World-Action Modeling with Observation-Guided Context Routing. *arXiv preprint arXiv:2606.09811*, 2026.

Jun Cen, Siteng Huang, Yuqian Yuan, Kehan Li, Hangjie Yuan, Chaohui Yu, Bohan Hou, Yuming Jiang, Jiayan Guo, Xin Li, et al. RynnVLA-002: A Unified Vision-Language-Action and World Model. *arXiv preprint arXiv:2511.17502*, 2025.

Suhwan Choi, Yunsung Lee, Yubeen Park, Chris Dongjoo Kim, Ranjay Krishna, Dieter Fox, and Youngjae Yu. vla-eval: A Unified Evaluation Harness for Vision-Language-Action Models. *arXiv preprint arXiv:2603.13966*, 2026.

Ronghao Dang, Jiayan Guo, Bohan Hou, Sicong Leng, Kehan Li, Xin Li, Jiangpin Liu, Yunxuan Mao, Zhikai Wang, Yuqian Yuan, et al. RynnBrain: Open Embodied Foundation Models. *arXiv preprint arXiv:2602.14979*, 2026.

Dibya Ghosh, Homer Rich Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, Jianlan Luo, You Liang Tan, Lawrence Yunliang Chen, Quan Vuong, Ted Xiao, Pannag R Sanketi, Dorsa Sadigh, Chelsea Finn, and Sergey Levine. Octo: An Open-Source Generalist Robot Policy. In *Proceedings of Robotics: Science and Systems*, July 2024.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P. Foster, Pannag R. Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. OpenVLA: An Open-Source Vision-Language-Action Model. In Pulkit Agrawal, Oliver Kroemer, and Wolfram Burgard (eds.), *Proceedings of The 8th Conference on Robot Learning*, volume 270 of *Proceedings of Machine Learning Research*, pp. 2679–2713, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. 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*, pp. 611–626, 2023.

Kehan Li, Bohan Hou, Minghao Zhu, Tianyi Zhang, Zesen Cheng, Zhikai Wang, Sicong Leng, Xin Li, Xiao Lin, Biying Yao, Minghua Zeng, Jiangpin Liu, Ronghao Dang, Jiayan Guo, Siteng Huang, Haoyu Zhao, Heng Ping, Yaxi Zhao, Kexiang Wang, Tong Lu, Shengke Xue, Jiahao Tang, Yulei Wang, Zejing Wang, Jianwei Gao, Shijian Lu, Chengju Liu, Jianfei Yang, Mingxiu Chen, and Deli Zhao. RynnBrain 1.1: Towards More Capable and Generalizable Embodied Foundation Model. *arXiv preprint arXiv:2607.17977*, 2026a.

Qixiu Li, Yaobo Liang, Zeyu Wang, Lin Luo, Xi Chen, Mozheng Liao, Fangyun Wei, Yu Deng, Sicheng Xu, Yizhong Zhang, et al. CogACT: A Foundational Vision-Language-Action Model for Synergizing Cognition and Action in Robotic Manipulation. *arXiv preprint arXiv:2411.19650*, 2024.

Xiangyu Li, Huaizhi Tang, Xin Ding, Weijun Wang, Ting Cao, and Yunxin Liu. OxyGen: Unified KV Cache Management for VLA Inference under Multi-Task Parallelism. *arXiv preprint arXiv:2603.14371*, 2026b.

Yushan Liu, Peibo Sun, Shoujie Li, Yifan Xie, Lingfeng Zhang, Xintao Chao, Shiyuan Dong, Fang Chen, Xiao-Ping Zhang, and Wenbo Ding. OA-WAM: Object-Addressable World Action Model for Robust Robot Manipulation. *arXiv preprint arXiv:2605.06481*, 2026.

Ziyan Liu, Yeqiu Chen, Hongyi Cai, Tao Lin, Shuo Yang, Zheng Liu, and Bo Zhao. Bridging the Semantic-Action Gap in Visual Token Pruning for Efficient VLA Inference. *arXiv preprint arXiv:2511.16449*, 2025.

Yunchao Ma, Yizhuang Zhou, Yunhuan Yang, Tiancai Wang, and Haoqiang Fan. Running VLAs at Real-Time Speed. *arXiv preprint arXiv:2510.26742*, 2025.

Khanh D. Nguyen, Hung T. Ho, Chinh T. Nguyen, Thanh Q. Duong, Linh D. Le, Duy M. H. Nguyen, Vien A. Ngo, and An T. Le. vla.cpp: A Unified Inference Runtime for Vision-Language-Action Models. *arXiv preprint arXiv:2606.08094*, 2026.

NVIDIA. Cosmos 3: Omnimodal World Models for Physical AI. *arXiv preprint arXiv:2606.02800*, 2026.

NVIDIA, Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi "Jim" Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, Joel Jang, Zhenyu Jiang, Jan Kautz, Kaushil Kundalia, Lawrence Lao, Zhiqi Li, Zongyu Lin, Kevin Lin, Guilin Liu, Edith Llontop, Loic Magne, Ajay Mandlekar, Avnish Narayan, Soroush Nasiriany, Scott Reed, You Liang Tan, Guanzhi Wang, Zu Wang, Jing Wang, Qi Wang, Jiannan Xiang, Yuqi Xie, Yinzhen Xu, Zhenjia Xu, Seonghyeon Ye, Zhiding Yu, Ao Zhang, Hao Zhang, Yizhou Zhao, Ruijie Zheng, and Yuke Zhu. GR00T N1: An Open Foundation Model for Generalist Humanoid Robots. *arXiv preprint arXiv:2503.14734*, 2025.

Ryuji Oi, Hikari Otsuka, Kosuke Matsushima, Yuki Ichikawa, Masato Motomura, Tatsuya Kaneko, and Daichi Fujiki. Training-Free Acceleration for Vision-Language-Action Models with Action Caching and Refinement. *arXiv preprint arXiv:2607.06370*, 2026.

Physical Intelligence, Bo Ai, Ali Amin, Raichelle Aniceto, Ashwin Balakrishna, Greg Balke, Kevin Black, George Bokinsky, Shihao Cao, Thomas Charbonnier, et al. Pi-0.7: A Steerable Generalist Robotic Foundation Model with Emergent Capabilities. *arXiv preprint arXiv:2604.15483*, 2026.

StarVLA Community. StarVLA: A Lego-like Codebase for Vision-Language-Action Model Developing. *arXiv preprint arXiv:2604.05014*, 2026.

Liang Su. Execution-State Capsules: Graph-Bound Execution-State Checkpoint and Restore for Low-Latency, Small-Batch, On-Device Physical-AI Serving. *arXiv preprint arXiv:2606.20537*, 2026.

Siyin Wang, Junhao Shi, Zhaoyang Fu, Xinzhe He, Feihong Liu, Chenchen Yang, Yikang Zhou, Zhaoye Fei, Jingjing Gong, Jinlan Fu, Mike Zheng Shou, Xuanjing Huang, Xipeng Qiu, and Yu-Gang Jiang. World Action Models: The Next Frontier in Embodied AI. *arXiv preprint arXiv:2605.12090*, 2026.

Wei Wu, Fangjing Wang, Fan Lu, He Sun, Shi Liu, Yunnan Wang, Yibin Yan, Yong Wang, Shuailei Ma, Xinyang Wang, et al. From Foundation to Application: Improving VLA Models in Practice. *arXiv preprint arXiv:2607.06403*, 2026.

Ling Xu, Borui Li, Hao Wu, Chuyu Han, Xiangyu Li, Mohan Hua, Shiqi Jiang, Ting Cao, Chuanyou Li, Sheng Zhong, et al. Embodied.cpp: A Portable Inference Runtime of Embodied AI Models on Heterogeneous Robots. *arXiv preprint arXiv:2607.02501*, 2026.

Siyu Xu, Yunke Wang, Chenghao Xia, Dihao Zhu, Tao Huang, and Chang Xu. VLA-Cache: Efficient Vision-Language-Action Manipulation via Adaptive Token Caching. *Advances in Neural Information Processing Systems*, 38:164448–164473, 2025.

Yandan Yang, Shuang Zeng, Tong Lin, Xinyuan Chang, Dekang Qi, Junjin Xiao, Haoyun Liu, Ronghan Chen, Yuzhi Chen, Dongjie Huo, et al. ABot-M0: VLA Foundation Model for Robotic Manipulation with Action Manifold Learning. *arXiv preprint arXiv:2602.11236*, 2026.

Yantai Yang, Yuhao Wang, Zichen Wen, Luo Zhongwei, Chang Zou, Zhipeng Zhang, Chuan Wen, and Linfeng Zhang. EfficientVLA: Training-Free Acceleration and Compression for Vision-Language-Action Models. *Advances in Neural Information Processing Systems*, 38:40891–40914, 2025.

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.

Chao Yu, Yuanqing Wang, Zhen Guo, Hao Lin, Si Xu, Hongzhi Zang, Quanlu Zhang, Yongji Wu, Chunyang Zhu, Junhao Hu, et al. RLinf: Flexible and Efficient Large-Scale Reinforcement Learning via Macro-to-Micro Flow Transformation. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 829–846, 2026.

Tianyuan Yuan, Zibin Dong, Yicheng Liu, and Hang Zhao. Fast-WAM: Do World Action Models Need Test-Time Future Imagination? *arXiv preprint arXiv:2603.16666*, 2026.

Jingxuan Zhang, Yunta Hsieh, Zhongwei Wan, Haokun Lin, Xin Wang, Ziqi Wang, Yingtie Lei, and Mi Zhang. QuantVLA: Scale-Calibrated Post-Training Quantization for Vision-Language-Action Models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 39539–39549, June 2026a.

Qihang Zhang, Lin Li, Luyao Zhang, Shuai Yang, Yiming Luo, Shuaiting Li, Ruilin Wang, Junke Wang, Jiahao Shao, Gangwei Xu, et al. Native Video-Action Pretraining for Generalizable Robot Control. *arXiv preprint arXiv:2607.08639*, 2026b.

Weisen Zhao, Lam Nguyen, Zhicong Lu, and Yuzhang Shang. C^3ache: Accelerating World Action Models with Cross Inference Chunk Cache. *arXiv preprint arXiv:2606.08962*, 2026.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

A SIMULATION PROTOCOLS

This appendix describes how we keep the simulator separate from policy inference and compare each model's official implementation with PhyAI under matched settings. The simulator produces observations and sends them to a policy server. The server returns an action chunk that the simulator applies before the next observation. Unless stated otherwise, the two run in separate processes. Timing excludes model loading and reports the warmed-up model-runner interval. The simulator computes task success.

A.1 MATCHED LIBERO EVALUATION

For $\pi_{0.5}$ we use the released `vla-evaluation-harness`. LIBERO calls a PhyAI policy server over WebSocket. The server uses the converted checkpoint with the PaLI-Gemma 3B tokenizer, BF16, FlashInfer, the PhyAI normalization kernel, and CUDA Graphs. Each request sends the agent view, wrist view, and robot state, and returns ten actions. The four suites have ten tasks each and 50 episodes per task, for 500 episodes per suite. We accept a run after the expected records are present and each request returns at most ten actions. Official and PhyAI runs share the simulator image, checkpoint, settings, and suite manifests. This evaluation checks task success rather than latency. Table 3 reports the matched success rates.

Table 3: Matched $\pi_{0.5}$ LIBERO results. Each suite uses 500 episodes, batch size one, and ten-action chunks. Delta is PhyAI minus OpenPI in percentage points.

Suite	OpenPI	PhyAI	Δ (pp)
`libero_spatial`	98.2%	97.8%	-0.4
`libero_object`	99.6%	99.8%	$+0.2$
`libero_goal`	97.6%	98.0%	$+0.4$
`libero_10`	95.6%	94.2%	-1.4

GR00T N1.7 uses LIBERO 0.1.0, MuJoCo 3.2.3, and robosuite 1.4.0 with EGL. It sends two 256×256 views, robot state, and the instruction over ZeroMQ. LIBERO-10 uses ten tasks, 50 fixed initial states per task, and seeds 42 through 91. The simulator executes the first eight decoded actions and scores success for at most 720 steps. Official Isaac-GR00T and PhyAI share the checkpoint, initial states, task order, and simulator parameters. Isaac-GR00T completes 456/500 (91.2%) and PhyAI 459/500 (91.8%), so downstream task results align with the official implementation.

A.2 ADDITIONAL INTEGRATION CHECKS

RoboTwin is an input and action-contract check for $\pi_{0.5}$ on ALOHA-AgileX assets with a fixed 100-episode manifest. The policy expects camera views in head, left, right order, while the simulator supplies head, right, left, so we swap the wrist views. Normalization uses the released joint-training statistics, including q_{01} and q_{99} quantiles. Before the full run, one request must produce a finite 32×14 action tensor. OpenPI and PhyAI share the scene, instruction, initial noise, and manifest. The policy uses BF16, ten denoising steps, and noise seed 42. The controller executes the first ten actions of each 32-action chunk at 30 Hz. PhyAI precision and downstream task behavior align with official OpenPI.

The π_0 LIBERO check uses the spatial and object suites, ten episodes per task, and batch size one, for 200 episodes. The policy produces a 50-step chunk, the simulator executes one action per step, and actions are clipped to $[-1, 1]$. PhyAI reports 143/200 successes (71.5%). This is a functional check rather than a matched comparison with an official implementation.

B KERNEL POLICY EXAMPLES

We illustrate the dispatcher in Section 3.2 with `rmsnorm_add`.

B.1 KERNEL REGISTRATION

The Python fragment registers three backends for this operator. Each `when` predicate defines $C_k(q)$. PhyAI/Triton accepts `TRITON_FLOATS`. `priority` sets the default order, `prepare` constructs the executable, and `reference=True` marks the PyTorch fallback.

```
Impl(
    kernel_id="flashinfer.rmsnorm_add",
    op="rmsnorm_add",
    priority=Priority.OPTIMIZED + 2,
    when=all_of(device.sm >= 90,
                dtype.residual.in_({"bf16", "fp16"})),
    prepare=_flashinfer_rmsnorm_add,
),
Impl(
    kernel_id="phyai_kernel.rmsnorm_add",
    op="rmsnorm_add",
    priority=Priority.OPTIMIZED,
    when=dtype.residual.in_(TRITON_FLOATS),
    prepare=_triton_rmsnorm_add,
),
Impl(
    kernel_id="torch.rmsnorm_add",
    op="rmsnorm_add",
    priority=Priority.REFERENCE,
    reference=True,
    when=dtype.input.is_set(),
    prepare=_torch_rmsnorm_add,
),
```

B.2 DEPLOYMENT POLICY

The YAML fragment orders eligible kernels using the highest-priority matching rule. `prefer` tries FlashInfer then PhyAI/Triton for BF16 on SM90 or newer, and PhyAI/Triton below SM90. `fallback: reference` permits PyTorch fallback. The illustrative graph-capture override takes precedence: `use` pins the PhyAI kernel, and selection fails if it is unusable.

```
schema: phyai.kernel/v1
profile: static
defaults:
  fallback: reference
rules:
  - id: sm90-fused-rmsnorm
    priority: 90
    match:
      op: rmsnorm_add
      device: {vendor: nvidia, arch: ">=sm90"}
      dtype.input: bf16
    prefer:
      - flashinfer.rmsnorm_add
      - phyai_kernel.rmsnorm_add
  - id: edge-rmsnorm
    priority: 80
    match:
      op: rmsnorm_add
      device.arch: "<sm90"
    prefer:
      - phyai_kernel.rmsnorm_add
overrides:
  - id: graph-capture-safe
    priority: 1000
    match:
      op: rmsnorm_add
      attrs.mode: graph_capture
    use: phyai_kernel.rmsnorm_add
```

C RL ROLLOUT PROFILE AND REPRODUCIBILITY DETAILS

We profile $\pi_{0.5}$ PPO with GAE in RLinf on one node with four NVIDIA H800 GPUs. RLinf and PhyAI are measured in independent fresh profiling jobs with the same initial SFT model, matched workload settings, and `runner.resume_dir=null`.

Each job records five profiling steps. Step 1 is warm-up, and all reported timings are averages over steps 2–5. Validation is disabled, and no checkpoint is written during these steps. Table 4 lists the shared training and profiling settings.

Table 4: RL training and profiling settings shared by both backends.

Setting	Value
Training environments	64 LIBERO-Spatial
Steps per rollout epoch	240
Environment group size	1
Rollout epochs	8
Actions per policy query	5
Flow-matching denoising steps	3
Actor micro batch size	128
Training global batch size	2,048
Actor update epochs	1
Rollout pipeline stages	1
Reported profiling steps	2–5
Number of averaged steps	4