RL-VLA^3: A Flexible and Asynchronous Reinforcement Learning Framework for VLA Training

Haoran Sun[1]*, **Yongjian Guo**[2]*, **Zhong Guan**[3]*,
Shuai Di[4], **Xiaodong Bai**[4], **Jing Long**[1,4], **Tianyun Zhao**[2], **Mingxi Luo**[4],
Xiaotie Deng[1], **Xu Chu**[1†], **Hongke Zhao**[3†], **Likang Wu**[3], **Xi Xiao**[2], **Sheng Wen**[2],
Yicheng Gong[4], **Junwu Xiong**[4†]
[1]Peking University, [2]Tsinghua University, [3]Tianjin University, [4]JDT AI Infra
[5]Swinburne University of Technology

Abstract

Reinforcement learning (RL) has emerged as a critical paradigm for post-training Vision-Language-Action (VLA) models, enabling embodied agents to adapt and improve through environmental interaction. However, existing RL frameworks for VLAs inherit synchronous design principles from traditional LLM training, treating entire rollouts as indivisible units and alternating strictly between data collection and policy optimization. This fundamentally mismatches the unique characteristics of VLA training, as physical simulators introduce highly variable, resource-intensive latencies. To address this, we introduce RL-VLA^3, a fully asynchronous distributed RL framework that enables fine-grained asynchronous interaction between simulation, inference, and training components through dynamic batching schedulers and flexible environment sharding strategies. Extensive experiments across diverse simulation backends, VLA architectures, and RL algorithms demonstrate that RL-VLA^3 achieves throughput improvements of up to 85.2% over synchronous baselines while maintaining identical sample efficiency, with scalability validated from 8 to 256 GPUs. To our knowledge, RL-VLA^3 is the first fully asynchronous RL training framework tailored specifically for the system-level challenges of VLA training.

1 Introduction

Vision-Language Action models (VLAs) (Ma et al., 2024; Zhong et al., 2025; Zhang et al., 2025) have emerged as a powerful paradigm for embodied AI, enabling agents to perceive and interact with their environments through a combination of visual and linguistic understanding. While foundation VLAs (Zitkovich et al., 2023; Black et al., 2024; Bjorck et al., 2025; Gemini Robotics Team, 2025; Kim et al., 2024; Shukor et al., 2025) trained via large-scale supervised fine-tuning have demonstrated basic capabilities, reinforcement learning (RL) has shown promise in further enhancing their performance and adaptability (Wagenmaker et al., 2025; Guo et al., 2025; Liu et al., 2025; Lu et al., 2025). Currently, the predominant approach for training VLAs with RL relies on traditional RL frameworks that treat entire rollouts as indivisible units for policy optimization (Li et al., 2025a; Zang et al., 2025; Li et al., 2025b). Consequently, the underlying training infrastructure largely follows the same design principles as traditional RL frameworks for LLM agents.

However, *VLA training exhibits distinct characteristics compared to classic LLM RL training*. Typical RL training can be divided into two phases: data generation (rollout) and policy optimization (training) (Sheng et al., 2025b; Hu et al., 2024). The rollout phase in LLM training is performed by an inference engine to generate large amounts of tokens, while VLA training involves extremely frequent interaction between the agent and the environment.

*Equal contributions, any order is acceptable by all authors.

†Corresponding authors. (chu_xu@pku.edu.cn, hongke@tju.edu.cn, xiongjunwu.1@jd.com)

Unlike environments such as web search or API calling, the environment in VLA training is a complex physics simulator, which is considerably more resource-intensive and time-consuming. For instance, in ManiSkill (Mu et al., 2021), a $\pi_{0.5}$ model (Black et al., 2024) takes 9 seconds to produce a batch of 640 actions, while the simulator requires 22 seconds to compute the next states and observations according to these actions. Moreover, different environment simulators exhibit varying patterns of resource consumption. Some simulators can efficiently leverage GPU resources for parallel processing (Mu et al., 2021; Nasiriany et al., 2024), while others rely more heavily on CPU computations, resulting in varying levels of GPU utilization (Liu et al., 2023; Chen et al., 2025a). Still others involve a mix of CPU and GPU computations for rendering and physics simulation, leading to unstable GPU utilization (Zhou et al., 2026).

Motivated by these observations, we propose RL-VLA^3 a flexible Reinforcement Learning framework specifically designed for training VLA with Asynchronous rollout interaction and Asynchronous training. Following the RLinf (Zang et al., 2025) framework, we explicitly define three resource groups: the Simulator, the Generator, and the Trainer. We introduce fine-grained decoupling, which provides a more flexible interface for users to configure. The interaction logic between these three groups is illustrated in Figure 1 and will be detailed in Section 3.1. Notably, *our entire training process executes asynchronously, allowing all three resource groups to progress independently and simultaneously*. The main contributions are as follows:

- A fully flexible rollout interface that allows users to configure the grouping and number of parallel environments within the Simulator, and specify the interaction order between Simulators and Generators.
- A fully asynchronous training framework in which the three components, Generators, Simulators, and Trainers, interact entirely asynchronously, with user-configurable batching strategies for Generators.
- Extensive experimental evaluations demonstrating the training efficiency and performance improvements of RL-VLA^3 across various tasks.

To our knowledge, RL-VLA^3 is the first fully asynchronous distributed RL training framework specifically designed for VLAs.

The remainder of the paper is organized as follows. In Section 2, we review related work on foundation VLA models and reinforcement learning frameworks for LLMs. In Section 3, we present the design principles of RL-VLA^3, including the overall framework, asynchronous environment interaction, and asynchronous policy optimization. In Section 4, we report experimental results on training throughput and performance, along with ablation studies and resource-utilization analysis. Finally, in Section 5, we conclude the paper and discuss future work.

2 Related Work

2.1 Reinforcement Learning for Foundation VLAs

Foundation VLAs models, such as RT-2 (Zitkovich et al., 2023), OpenVLA (Kim et al., 2024), π_0 (Black et al., 2024), and GR00T (Bjorck et al., 2025), have demonstrated strong basic robotic manipulation capabilities through large-scale supervised fine-tuning (SFT) on human-teleoperated data. However, because SFT policies often struggle to recover from out-of-distribution errors or generalize to novel states, reinforcement learning has emerged as a crucial post-training paradigm to enable continuous improvement and environmental adaptation (Wagenmaker et al., 2025; Guo et al., 2025; Tan et al., 2025; Kim et al., 2025; Xiao et al., 2025).

Recent works have demonstrated the efficacy of RL in enhancing VLA success rates (Liu et al., 2025; Lu et al., 2025; Li et al., 2025b; Chen et al., 2025b). To facilitate this, initial distributed RL frameworks like SimpleVLA (Li et al., 2025a) and RLinf (Zang et al., 2025) have been proposed to adapt RL algorithms for embodied AI. Current frameworks typically

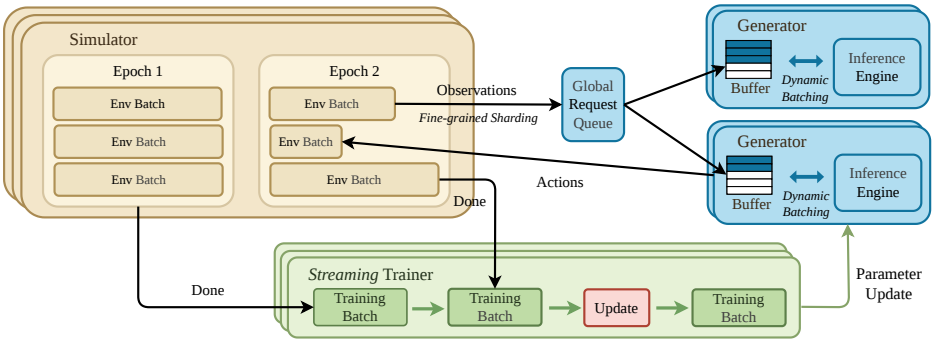

Figure 1: Overall architecture of RL-VLA^3. Generators, Simulators, and Trainers interact fully asynchronously; black arrows (→) indicate data flow. We use *fine-grained sharding* of environment batches and *dynamic batching* for Generator inference to accommodate asynchronous interaction and rollouts, improving throughput.

require all GPUs to load all resource groups concurrently (a colocated strategy), which introduces substantial efficiency losses due to frequent context switching between resources, such as alternating back and forth between the physical simulator and the model inference engine. Although RLinf (Zang et al., 2025) has proposed separated and hybrid resource allocation strategies to mitigate this overhead, they still rely on synchronous execution during the rollout phase and maintain a synchronous training-inference pipeline. These synchronization barriers severely bottleneck overall throughput. Motivated by these limitations, RL-VLA^3 introduces what is, to the best of our knowledge, the first fully asynchronous RL training framework specifically designed for VLAs.

2.2 Distributed RL Frameworks for LLMs vs. Embodied AI

The rapid evolution of RL from Human Feedback (RLHF) has driven the development of highly efficient distributed RL frameworks for LLMs. Systems such as VeRL (Sheng et al., 2025b) and OpenRLHF (Hu et al., 2024) introduce hierarchical control, zero-redundancy model resharding, and highly optimized vLLM inference to balance flexibility and efficiency. To overcome global synchronization bottlenecks in large-scale deployment, recent systems like AReaL (Fu et al., 2025), Laminar (Sheng et al., 2025a), and ROLLART (Gao et al., 2025) advocate for fully decoupled, fine-grained asynchronous execution at the trajectory level.

While RL-VLA^3 is inspired by these decoupled LLM architectures, VLA training presents fundamentally different system-level challenges. In LLM RL pipelines, the "environment" is typically a neural reward model that runs entirely on GPUs with stable, predictable latencies. Conversely, the "environment" in embodied AI is a complex physics simulator (e.g., ManiSkill (Mu et al., 2021), LIBERO (Liu et al., 2023)). These simulators rely heavily on varied CPU and GPU computations, require substantial memory footprints, and exhibit highly unpredictable latencies due to dynamic collision calculations and rendering tasks. Simply migrating LLM RL frameworks to VLA tasks fails to address these simulator-specific bottlenecks. To address this, RL-VLA^3 introduces a fully flexible, asynchronous simulator interface alongside dynamic batching strategies, specifically tailored to absorb the latency fluctuations of physics engines and maximize throughput for embodied AI tasks.

3 Design Principles of RL-VLA^3

In this section, we elaborate on the system design of RL-VLA^3. We first provide an overview of the framework, defining the distinct resource groups, their interactions, data flow, and user-configurable features. Next, we delve into the specific mechanisms driving the two critical interactions within our pipeline: the asynchronous rollout between parallel Simulators

and Generators, and the asynchronous training between the rollout workers (Simulators and Generators) and the Trainer.

3.1 Overall Framework

The standard reinforcement learning pipeline consists of two phases: data generation (rollout) and policy optimization (training). In synchronous RL, these phases execute sequentially, an iterative process expressed as $(\text{Rollout} \rightarrow \text{Training})^N$, where N denotes the number of training steps. Because VLAs rely on physical simulation, the rollout phase can be further subdivided into Simulator-side environment stepping (which yields observations) and Generator-side action inference, expressed as $\text{Rollout} = (\text{Simulator} \rightarrow \text{Generator})^{N_e}$, where N_e is the number of environment interactions. Motivated by these naturally decoupled phases, we build upon Zang et al. (2025) by explicitly separating the architecture into three distinct resource groups: the Simulator, the Generator, and the Trainer. As illustrated in Figure 1, each Generator loads the VLA model and operates an independent inference engine; each Trainer is dedicated to policy gradient computations and parameter updates; and each Simulator hosts several environment batches (Env). Within each batch, vectorized environments execute concurrently to produce parallel observations.

The Execution Flow. To manage the high-throughput flow of these observations, we introduce a highly configurable, fully asynchronous communication mechanism. Once generated, observations are processed through an optional sharding step and posted to a global request queue. Users can implement arbitrary *sharding strategies* to shard the environment batch into multiple slots and explicitly route them to specific Generators. The default request queue operates as a priority queue sorted by slot arrival time and also supports custom sorting functions. A *dynamic batching scheduler* manages the trigger conditions for Generator inference, allowing users to bound execution by maximum batch size and maximum wait latency. When an environment batch completes a full episode, the entire trajectory is asynchronously transmitted to the Trainer for continuous policy optimization. Crucially, *our entire training process executes asynchronously, allowing all three resource groups to progress independently and simultaneously.*

For further illustration, we summarize the execution flow in pseudo code, which is presented in the following Figure 2. The main pipeline (Left) initializes rollouts (Line 4) and then continuously collects completed trajectories for optimization (Line 6). It also manages model version synchronization (Line 11) when the number of update epochs meets the global condition. The rollout phase is mainly executed by the collect_rollout function (Right). It first starts the Simulator processing (Line 3-9), in which the simulators are computing the observation data. And it also controls the triggering of Generator inference (Line 11-12) via the dynamic batching scheduler. These asynchronous interactions reduce the blocking time and significantly improve the overall training throughput.

```
1  async def pipeline():
2      rollout_version = 0
3      local_updates = 0
4      spawn(collect_rollout(rollout_version))
5      while not finished():
6          batch = await queue.get()
7          train(batch)
8          local_updates += 1
9          if local_updates == sync_interval:
10             rollout_version += 1
11             emit(sync_to_rollout(
12                 rollout_version))
13             local_updates = 0
14
```

```
1  async def collect_rollout(version):
2      for env in environments:
3          spawn(
4              while not done(env):
5                  submit(env.state, version)
6                  action = await next_action(env)
7                  transition, env.state = env.step(action)
8                  await queue.put((transition, version))
9                  )
10     while True:
11         reqs = await collect_requests(
12             max_batch_size, timeout_ms)
13         return_actions(reqs, model.forward(reqs))
14
```

Figure 2: Pseudocode of two asynchronous components: `pipeline` (left) manages versioned training and synchronization, and `collect_rollout` (right) handles environment stepping, request submission, and batched inference.

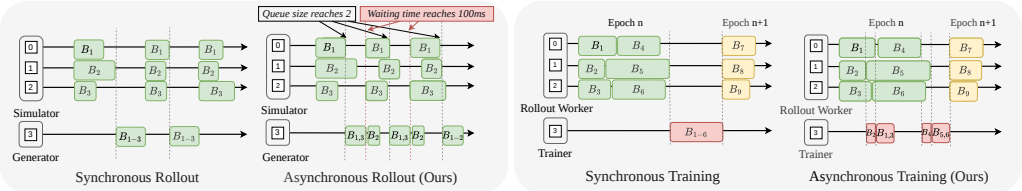

Figure 3: A demonstration of our proposed asynchronous Rollout (Left) and Training (Right). We introduce a dynamic batching scheduler to aggregate requests into a batch and trigger Generator inference.

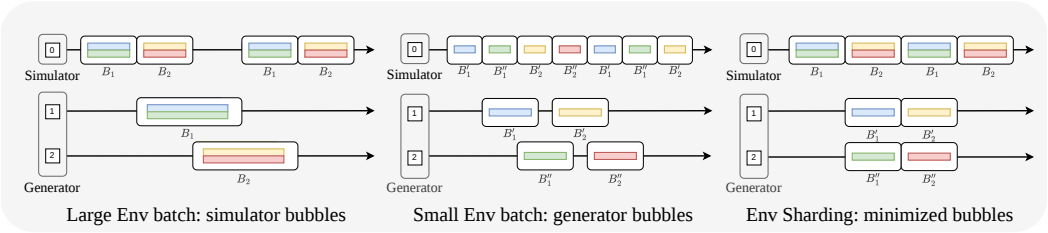

Figure 4: Fine-grained environment sharding for parallel interaction. Our framework shards parallel environment batches into multiple slots and distributes requests across different Generators. This maintains high throughput from large environment batches while reducing the Simulators' waiting time.

In the following two subsections, we detail the specific mechanisms enabling these core interactions: asynchronous environment stepping between the Simulator and Generator, and asynchronous policy optimization between the rollout workers and the Trainer.

3.2 Asynchronous Rollout between Parallel Simulators and Generators

During the RL training of VLA models, rollout throughput is heavily bottlenecked by rigid synchronization dependencies. As illustrated in Figure 3 (Left), classic synchronous rollout frameworks wait for all Simulators to finish environment stepping before collecting observations to form a unified inference batch for the Generator. This creates severe synchronization overhead and exacerbates the long-tail latency caused by slower, computationally heavy environment batches.

To resolve this, we propose a fine-grained asynchronous interaction mechanism that strictly decouples Simulator stepping from Generator inference. Instead of relying on global synchronization barriers, observation requests are posted to a request queue immediately after an individual environment completes its step. Generators are then triggered to perform inference independently, without waiting for the entire cohort of Simulators to finish. To maximize throughput and adapt to the varying computational characteristics of different environments, we introduce two flexible strategies: a *dynamic batching scheduler* and *fine-grained environment sharding*.

Dynamic Batching Scheduler. Because most environments do not natively support generating massive observation batches instantaneously, triggering Generator inference the moment a queue becomes non-empty leads to severe hardware underutilization. To counter this, we introduce a dynamic batching scheduler to intelligently aggregate requests into an optimal batch before inference. As shown in Figure 3 (Right), the scheduler operates on two orthogonal constraints: a maximum batch size and a maximum wait latency. Generator inference is triggered only when the queue size reaches the batch size threshold, or when the oldest pending request exceeds the latency limit. By tuning these hyperparameters, we can effectively eliminate Generator pipeline bubbles and improve overall throughput via optimized inference batch sizes.

Fine-grained Environment Sharding. For environments capable of a high degree of parallelism, we shard massive environment batches into multiple smaller slots, distributing the resulting requests across different Generators. This strategy preserves the high throughput inherent to large environment batches while drastically reducing the waiting time for any single Simulator. As demonstrated in Figure 4, mapping a massive environment batch to a single Generator forces the Simulator to stall while waiting for inference, creating Simulator-side bubbles. Conversely, using undersized batches fails to saturate the Simulators' throughput, creating Generator-side bubbles. By sharding batches and routing slots across multiple Generators, our framework strikes an optimal balance, effectively minimizing idle time across both resources.

3.3 Asynchronous Training

Beyond the asynchronous rollout, we also decouple the global interaction between the rollout workers and the Trainer to eliminate hardware idle time during policy updates. In standard synchronous RL pipelines (Figure 3, Right), data collection and policy optimization alternate strictly. The Trainer must wait for all rollout workers to complete their assigned trajectories and aggregate a full dataset before initiating the epoch's policy update. This rigid barrier inevitably leaves the Trainer idle during the rollout phase and forces the rollout workers to stall during the training phase.

To eliminate these pipeline bubbles, we implement a continuous, fully asynchronous training mechanism. In RL-VLA^3, once a Simulator batch completes an episode, the resulting trajectory is immediately pushed to the Trainer. The Trainer continuously processes this incoming data, ensuring that both the rollout workers and the optimization engine maintain high utilization. This overlapping execution significantly reduces the overall wall-clock training time.

4 Experimental Results

In this section, we evaluate RL-VLA^3 against baseline training strategies across a diverse set of configurations, encompassing various backbone models, simulation environments, training algorithms, and GPU resource scales.[1] Our experiments demonstrate that our proposed flexible, asynchronous execution framework achieves substantially higher throughput while maintaining strict training stability.

4.1 Experiment Setup

Models, Simulators, and Algorithms. We evaluate RL-VLA^3 using several state-of-the-art pretrained VLA models. This includes diffusion-based architectures such as GR00T N1.5 (Bjorck et al., 2025) and the π series (π_0 and $\pi_{0.5}$) (Black et al., 2024), as well as OpenVLA-OFT (Kim et al., 2025), an autoregressive model. Following Zang et al. (2025), we adapt OpenVLA-OFT to predict action chunks rather than single-step actions to improve interaction efficiency.

For simulation environments, we select LIBERO (Liu et al., 2023), ManiSkill (Mu et al., 2021), Meta-World (Yu et al., 2020), and RoboCasa (Nasiriany et al., 2024). This selection provides a comprehensive testbed of simulation backends: LIBERO and Meta-World rely on the CPU-bound MuJoCo physics engine; ManiSkill utilizes highly parallelized, GPU-accelerated ray-traced rendering; and RoboCasa features computationally heavy, photorealistic environments based on Robosuite. For policy optimization, we test our framework using two representative reinforcement learning algorithms: Proximal Policy Optimization (PPO) (Schulman et al., 2017) and Group Relative Policy Optimization (GRPO) (Shao et al., 2024).

[1]The code is available at `https://github.com/Haoran0301/RL-VLA3`

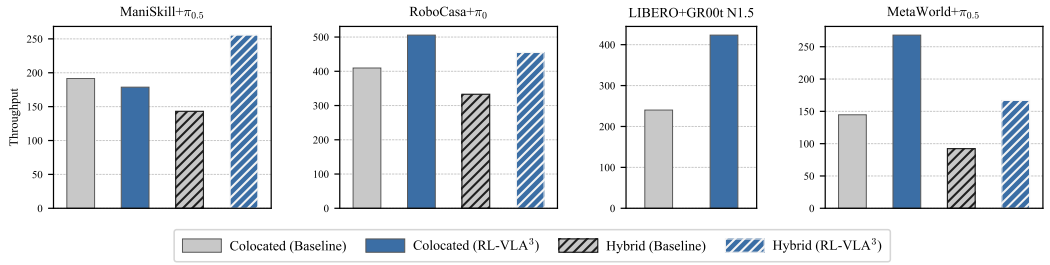

Figure 5: Single-node (8-GPU) throughput of RL-VLA^3 versus the synchronous baseline across backbone models, environments, and the Colocated/Hybrid placements reported in our setup. RL-VLA^3 achieves the highest throughput in every reported setting.

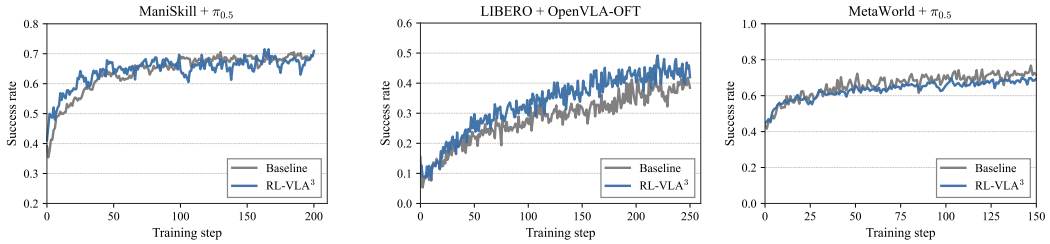

Figure 6: Success-rate curves for ManiSkill+$\pi_{0.5}$, LIBERO+OpenVLA-OFT, and Meta-World+$\pi_{0.5}$ under the synchronous baseline versus RL-VLA^3. The trajectories align in environment steps, confirming comparable sample efficiency while RL-VLA^{3}ś higher throughput substantially reduces wall-clock time to comparable success levels.

Baselines and Evaluation Metrics. We build our codebase upon the foundation of RLinf (Zang et al., 2025) and adopt their synchronous training pipeline as our primary baseline. We compare performance across two GPU placement strategies: **Colocated**, where a single GPU timeshares the Simulator, Generator, and Trainer; and **Hybrid**, where each GPU hosts a Trainer alongside either a Simulator or a Generator.

To assess system efficiency, our primary metric is **throughput**, defined as the total number of environment state transitions processed per unit time. Assuming a constant action chunk size, this is mathematically equivalent to the total number of action-inference steps executed by the Generator per unit time. To ensure a rigorous and fair architectural comparison, we carefully calibrate hyperparameters to maintain comparable overall GPU utilization between the baseline and RL-VLA^3, thereby isolating the performance gains attributed strictly to our asynchronous design. Detailed configurations are provided in Section A. To verify algorithmic correctness, we also evaluate **training performance** by plotting success rates over a finite step count.

4.2 Benchmark Results

In this subsection, we present the main benchmark results demonstrating the efficiency and scalability of RL-VLA^3.

Throughput Comparison. Figure 5 illustrates the throughput of RL-VLA^3 compared to the synchronous baseline on a single 8-GPU node. Because LIBERO environments do not require GPU acceleration for physics computation, we omit Simulator-dedicated GPUs for LIBERO and report only Colocated placement.

As shown, RL-VLA^3 achieves the highest throughput across all environments. Under the Hybrid placement strategy, RL-VLA^3 outperforms the baseline by approximately 78.6% on

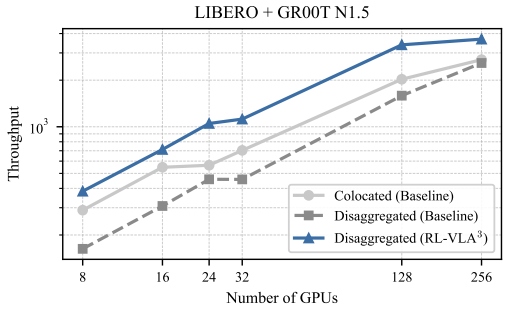

Figure 7: The scaling behavior regarding the throughput versus GPU count for LIBERO+GR00T N1.5 under multiple placement modes.

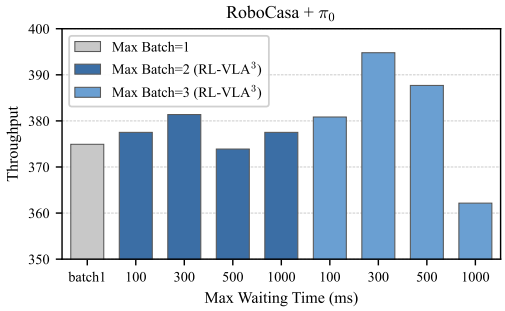

Figure 8: RoboCasa+π_0 throughput under Hybrid placement when varying the dynamic batching scheduler's maximum batch number and latency threshold.

ManiSkill, 36.7% on RoboCasa, and 80.9% on Meta-World. These substantial gains confirm that our fully asynchronous design successfully masks the blocking time inherent to the synchronous rollout and training phases.

Under the Colocated placement strategy, RL-VLA^3 still yields significant improvements: 23.5% on RoboCasa, 76.3% on LIBERO, and 85.2% on Meta-World. The synchronous baseline in Colocated mode is heavily bottlenecked by fluctuating resource demands, resulting in inefficient GPU/CPU utilization. RL-VLA^3 resolves this by enabling the Simulator, Generator, and Trainer to execute asynchronously, yielding a continuously smoothed, high-utilization workload.

ManiSkill presents an interesting edge case. Because its simulation is natively highly parallelizable, the baseline strategy can already achieve massive batch sizes (up to 256) for both environment stepping and inference. In a strictly Colocated setup, forcing asynchronous execution can paradoxically introduce overhead due to hyper-frequent context switching between the GPU-heavy simulation and the model inference engine. However, RL-VLA^3 easily overcomes this in Hybrid mode; by physically isolating the Simulator and Generator on separate GPUs, we preserve the massive environment batch sizes while eliminating the context-switching penalty, resulting in superior end-to-end throughput.

Scaling Behavior. We further investigate the distributed scaling properties of RL-VLA^3 compared to the baseline. Figure 7 plots the throughput for the LIBERO+GR00T N1.5 configuration scaling from 8 up to 256 GPUs on various placement modes. Here, we include a disaggregated mode which places the rollout workers (Generator and Simulator) and Trainer on separate GPUs. RL-VLA^3 exhibits near-linear scaling efficiency from 8 to 24 GPUs. While the scaling efficiency moderates up to 128 GPUs and experiences sublinear degradation toward 256 GPUs, RL-VLA^3 maintains a consistent throughput advantage over the baseline at every scale. The efficiency drop at extreme scales is a known artifact of distributed RL, primarily driven by the heavy communication overhead required for weight broadcasting and gradient synchronization. Mitigating this communication bottleneck at the 256+ GPU scale remains a promising direction for future work.

Training Performance. To confirm that asynchronous execution does not degrade the mathematical integrity of the policy updates, we validate the training stability of RL-VLA^3. As shown in Figure 6, RL-VLA^3 achieves success rate curves directly comparable to the synchronous baseline with respect to training steps. Consequently, because our framework generates these steps at a significantly higher throughput, RL-VLA^3 drastically accelerates the policy's wall-clock convergence time.

Table 1: Ablation study on different levels of asynchrony. We present the throughput for 4 different configurations: (a) LIBERO+$\pi_{0.5}$ on Hybrid mode; (b) OpenVLA-OFT+$\pi_{0.5}$ on Hybrid mode; (c) RoboCasa+π_0 on Colocated mode; (d) MetaWorld+$\pi_{0.5}$ on Hybrid mode.

Configuration	(a)			(b)			(c)	(d)
	8 GPUs	16 GPUs	32 GPUs	8 GPUs	16 GPUs	32 GPUs	8 GPUs	8 GPUs
Baseline	162.75	307.84	457.23	144.35	249.66	472.33	379.25	92.29
+ Rollout Async	229.68	441.49	737.46	242.29	397.19	748.98	437.60	113.62
+ Rollout & Train Async	383.40	713.38	1120.91	376.31	569.88	989.22	505.68	166.94

4.3 Ablation Studies

Ablation on Dynamic Batching Strategy. To validate the dynamic batching scheduler introduced in Section 3.2, we analyze its impact on the throughput using the RoboCasa + π_0 configuration in Hybrid placement. We allocate 6 GPUs for the Simulators and 2 GPUs for the Generators, assigning two parallel environment batches to each Simulator GPU. Because the sizes of these 12 environment batches are identical, we define the "maximum batch number" as the proxy for maximum batch size.

The results, presented in Figure 8, illustrate a classic systems trade-off between batching efficiency and queueing latency. Taking a maximum batch number of 3 as an example, throughput initially climbs as the maximum tolerable latency increases, before eventually declining. When the latency threshold is too restrictive, the system is forced into suboptimal, small-batch inference. Conversely, an excessively high latency threshold introduces severe queueing delays that stall the pipeline and outweigh the computational benefits of larger batches. This ablation confirms that properly tuning the dynamic batching hyperparameters is critical for locating the optimal hardware operating point.

Ablation on Different Levels of Asynchrony. Finally, we isolate the independent contributions of rollout-side and training-side asynchrony, which are denoted by Rollout Async and Train Async, respectively. Table 1 reports throughput for four settings: (a) LIBERO+$\pi_{0.5}$ in hybrid placement; (b) OpenVLA-OFT+$\pi_{0.5}$ in hybrid placement; (c) RoboCasa+π_0 in colocated placement; and (d) MetaWorld+$\pi_{0.5}$ in hybrid placement, and various GPU scales from 8 up to 32.

In every configuration, isolating either asynchronous component yields measurable throughput improvements over the baseline. Notably, Train Async consistently provides the more pronounced performance lift. This suggests that successfully overlapping the computationally expensive policy gradient updates with continuous data generation is the dominant factor in maximizing end-to-end throughput. Rollout Async also shows clear improvements; we attribute these primarily to our dynamic batching strategy for environment interaction (Section 3.2), and the consistent lift over the baseline further validates the effectiveness of our asynchronous design.

4.4 Resource Utilization Analysis

Beyond aggregate throughput, we examine fine-grained CPU/GPU utilization and memory usage during the first training step to make the pipeline-level gains more concrete. Figure 9 compares the synchronous baseline and RL-VLA^3 on LIBERO+GR00T N1.5 under colocated placement, the setting that achieves the highest throughput for each method in Table 4.

The utilization curves reveal two characteristic differences. First, RL-VLA^3 starts policy optimization immediately after the first rollout finishes, so GPU memory and utilization rise early and remain elevated; the baseline keeps the GPU comparatively idle until all rollouts are collected and then incurs a late training burst. Second, asynchronous training streams trajectories in batches, which avoids the one-shot host-to-device transfer of a large VLA dataset (including image observations) that drives prolonged GPU waiting and host-memory pressure in the synchronous pipeline. Together, these effects explain why the same

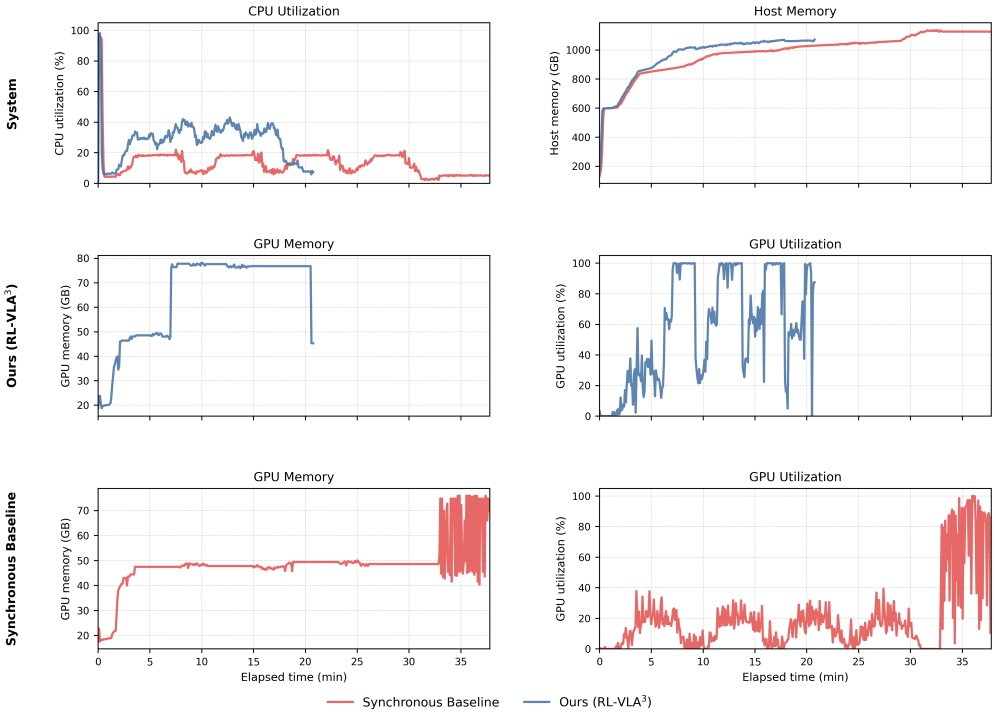

Figure 9: Comparison of resource utilization during the first training step between Colocated (Baseline) and Colocated (Ours) on LIBERO+GR00T N1.5. RL-VLA^3 finishes the step substantially earlier while sustaining higher GPU utilization, as training begins once the first rollout completes rather than after the full rollout cohort.

colocated hardware finishes the first step much faster under RL-VLA^3. Full utilization traces for ManiSkill and Meta-World, along with additional discussion, are provided in Appendix A.4.

5 Conclusion and Future Work

In this paper, we introduced RL-VLA^3, a fully asynchronous distributed reinforcement learning framework designed specifically for training Vision-Language-Action models. By enabling fine-grained asynchronous interaction between simulation, inference, and training components through dynamic batching schedulers and flexible environment sharding strategies, we eliminate the rigid synchronization barriers that plague traditional synchronous RL frameworks. Extensive experiments demonstrate that RL-VLA^3 achieves substantial throughput improvements of up to 85.2% over synchronous baselines while maintaining identical sample efficiency, with scalability validated from 8 to 256 GPUs.

Several promising directions remain for future exploration. First, optimizing communication overhead between resource groups could further improve scaling efficiency, particularly at extreme scales where current bottlenecks limit sharding strategies. Second, developing autonomous agents to dynamically adjust placement strategies, batching scheduler hyperparameters, and environment sharding policies represents an exciting avenue for adaptive system optimization. Third, designing more efficient RL algorithms specifically tailored to the unique characteristics of VLA training could yield both computational and sample efficiency gains.

Acknowledgements

This work was supported by New Generation Artificial Intelligence-National Science and Technology Major Project (2025ZD0122904), Natural Science Foundation of China (Grant No. 62572010), the Beijing-Tianjin-Hebei Natural Science Foundation Cooperation Project (25JJJJC0045), Natural Science Foundation of Tianjin (No. 24JCQNJC01560), Natural Science Foundation of Guangdong Province (2025A1515011946).

Ethics Statement

This paper presents a novel framework for reinforcement learning in virtual learning environments. We have taken care to ensure that our research adheres to ethical guidelines, including considerations for data privacy, fairness, and potential societal impacts.

References

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

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Zixuan Li, Qiwei Liang, Xianliang Lin, Yiheng Ge, Zhenyu Gu, et al. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation. *arXiv preprint arXiv:2506.18088*, 2025a.

Yuhui Chen, Shuai Tian, Shugao Liu, Yingting Zhou, Haoran Li, and Dongbin Zhao. Conrft: A reinforced fine-tuning method for vla models via consistency policy. *arXiv preprint arXiv:2502.05450*, 2025b.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

Wei Gao, Yuheng Zhao, Tianyuan Wu, Shaopan Xiong, Weixun Wang, Dakai An, Lunxi Cao, Dilxat Muhtar, Zichen Liu, Haizhou Zhao, et al. Rollart: Scaling agentic rl training via disaggregated infrastructure. *arXiv preprint arXiv:2512.22560*, 2025.

Gemini Robotics Team. Gemini Robotics 1.5: Pushing the Frontier of Generalist Robots with Advanced Embodied Reasoning, Thinking, and Motion Transfer. *arXiv e-prints*, art. arXiv:2510.03342, October 2025. doi: 10.48550/arXiv.2510.03342.

Yanjiang Guo, Jianke Zhang, Xiaoyu Chen, Xiang Ji, Yen-Jen Wang, Yucheng Hu, and Jianyu Chen. Improving vision-language-action model with online reinforcement learning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 15665–15672. IEEE, 2025.

Jian Hu, Xibin Wu, Zilin Zhu, Weixun Wang, Dehao Zhang, Yu Cao, et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 6, 2024.

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

Moo Jin Kim, Chelsea Finn, and Percy Liang. Fine-tuning vision-language-action models: Optimizing speed and success. *arXiv preprint arXiv:2502.19645*, 2025.

Haozhan Li, Yuxin Zuo, Jiale Yu, Yuhao Zhang, Zhaohui Yang, Kaiyan Zhang, Xuekai Zhu, Yuchen Zhang, Tianxing Chen, Ganqu Cui, et al. Simplevla-rl: Scaling vla training via reinforcement learning. *arXiv preprint arXiv:2509.09674*, 2025a.

Hengtao Li, Pengxiang Ding, Runze Suo, Yihao Wang, Zirui Ge, Dongyuan Zang, Kexian Yu, Mingyang Sun, Hongyin Zhang, Donglin Wang, et al. Vla-rft: Vision-language-action reinforcement fine-tuning with verified rewards in world simulators. *arXiv preprint arXiv:2510.00406*, 2025b.

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

Jijia Liu, Feng Gao, Bingwen Wei, Xinlei Chen, Qingmin Liao, Yi Wu, Chao Yu, and Yu Wang. What can rl bring to vla generalization? an empirical study. *arXiv preprint arXiv:2505.19789*, 2025.

Guanxing Lu, Wenkai Guo, Chubin Zhang, Yuheng Zhou, Haonan Jiang, Zifeng Gao, Yansong Tang, and Ziwei Wang. Vla-rl: Towards masterful and general robotic manipulation with scalable reinforcement learning. *arXiv preprint arXiv:2505.18719*, 2025.

Yueen Ma, Zixing Song, Yuzheng Zhuang, Jianye Hao, and Irwin King. A survey on vision-language-action models for embodied ai. *arXiv preprint arXiv:2405.14093*, 2024.

T Mu, Z Ling, F Xiang, D Yang, X Li, S Tao, Z Huang, Z Jia, and H Su. Maniskill: Generalizable manipulation skill benchmark with large-scale demonstrations. In *35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

Soroush Nasiriany, Abhiram Maddukuri, Lance Zhang, Adeet Parikh, Aaron Lo, Abhishek Joshi, Ajay Mandlekar, and Yuke Zhu. Robocasa: Large-scale simulation of everyday tasks for generalist robots. *arXiv preprint arXiv:2406.02523*, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, et al. Laminar: A scalable asynchronous rl post-training framework. *arXiv preprint arXiv:2510.12633*, 2025a.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025b.

Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, Simon Alibert, Matthieu Cord, Thomas Wolf, and Remi Cadene. Smolvla: A vision-language-action model for affordable and efficient robotics, 2025. URL https://arxiv.org/abs/2506.01844.

Shuhan Tan, Kairan Dou, Yue Zhao, and Philipp Krähenbühl. Interactive post-training for vision-language-action models. *arXiv preprint arXiv:2505.17016*, 2025.

Andrew Wagenmaker, Yunchu Zhang, Mitsuhiko Nakamoto, Seohong Park, Waleed Yagoub, Anusha Nagabandi, Abhishek Gupta, and Sergey Levine. Steering your diffusion policy with latent space reinforcement learning. In *Conference on Robot Learning*, pp. 258–282. PMLR, 2025.

Wenli Xiao, Haotian Lin, Andy Peng, Haoru Xue, Tairan He, Yuqi Xie, Fengyuan Hu, Jimmy Wu, Zhengyi Luo, Linxi "Jim" Fan, Guanya Shi, and Yuke Zhu. Self-improving vision-language-action models with data generation via residual rl, 2025. URL https://arxiv.org/abs/2511.00091.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on robot learning*, pp. 1094–1100. PMLR, 2020.

Hongzhi Zang, Mingjie Wei, Si Xu, Yongji Wu, Zhen Guo, Yuanqing Wang, Hao Lin, Liangzhi Shi, Yuqing Xie, Zhexuan Xu, et al. Rlinf-vla: A unified and efficient framework for vla+ rl training. *arXiv preprint arXiv:2510.06710*, 2025.

Dapeng Zhang, Jing Sun, Chenghui Hu, Xiaoyan Wu, Zhenlong Yuan, Rui Zhou, Fei Shen, and Qingguo Zhou. Pure vision language action (vla) models: A comprehensive survey. *arXiv preprint arXiv:2509.19012*, 2025.

Yifan Zhong, Fengshuo Bai, Shaofei Cai, Xuchuan Huang, Zhang Chen, Xiaowei Zhang, Yuanfei Wang, Shaoyang Guo, Tianrui Guan, Ka Nam Lui, et al. A survey on vision-language-action models: An action tokenization perspective. *arXiv preprint arXiv:2507.01925*, 2025.

Chen Zhou, Haoran Sun, Hedan Yang, Jing Long, Junwu Xiong, Luqiao Wang, Mingxi Luo, Qiming Yang, Shuai Di, Song Wang, et al. Thousand-gpu large-scale training and optimization recipe for ai-native cloud embodied intelligence infrastructure. *arXiv preprint arXiv:2603.11101*, 2026.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, Quan Vuong, Vincent Vanhoucke, Huong Tran, Radu Soricut, Anikait Singh, Jaspiar Singh, Pierre Sermanet, Pannag R. Sanketi, Grecia Salazar, Michael S. Ryoo, Krista Reymann, Kanishka Rao, Karl Pertsch, Igor Mordatch, Henryk Michalewski, Yao Lu, Sergey Levine, Lisa Lee, Tsang-Wei Edward Lee, Isabel Leal, Yuheng Kuang, Dmitry Kalashnikov, Ryan Julian, Nikhil J. Joshi, Alex Irpan, Brian Ichter, Jasmine Hsu, Alexander Herzog, Karol Hausman, Keerthana Gopalakrishnan, Chuyuan Fu, Pete Florence, Chelsea Finn, Kumar Avinava Dubey, Danny Driess, Tianli Ding, Krzysztof Marcin Choromanski, Xi Chen, Yevgen Chebotar, Justice Carbajal, Noah Brown, Anthony Brohan, Montserrat Gonzalez Arenas, and Kehang Han. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In Jie Tan, Marc Toussaint, and Kourosh Darvish (eds.), *Proceedings of The 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pp. 2165–2183. PMLR, 06–09 Nov 2023. URL https://proceedings.mlr.press/v229/zitkovich23a.html.

A Further Experimental Results

In this section, we provide additional experimental results on more placement strategies and hyperparameters.

A.1 Implementation Details and Additional Throughput Results

Tables 2–5 report full per-run settings and measured throughput for the four simulators used in our study, complementing the summary in Figure 5. The bold values in the tables are the ones we reported in Figure 5. All experiments in this section were conducted on a single node with 8 GPUs. We sweep Colocated versus Hybrid placement, the number of environment batches hosted per Simulator GPU, and which asynchronous features are enabled (rollout-side decoupling with dynamic batching, training-side streaming optimization, or both). Each value is an average over 5 training steps. We present a detailed analysis below.

Table 2: Detailed hyperparameters and throughput for different experimental setups on ManiSkill+$\pi_{0.5}$ with GRPO.

Method	Baseline	Baseline	RL-VLA^3	Baseline	RL-VLA^3	Baseline	RL-VLA^3	RL-VLA^3
Placement Settings								
Simulator	0-7	0-7	0-7	0-7	0-7	0-5	0-5	0-5
Generator	0-7	0-7	0-7	0-7	0-7	6-7	6-7	6-7
Trainer	0-7	0-7	0-7	0-7	0-7	0-7	0-7	0-7
Environment Settings								
Total Env	2048	2048	2048	2048	2048	3264	3264	3264
Batch Number Per GPU	1	2	2	4	4	2	2	2
Max Episode Steps	80	80	80	80	80	80	80	80
Training Hyperparameters								
Rollout Epoch	4	4	4	4	4	4	4	4
Update Epoch	2	2	2	2	2	2	2	2
Micro Batch Size	32	32	32	32	32	32	32	32
Global Batch Size	2048	2048	2048	2048	2048	2048	2048	2048
Model Specifications								
Model Num Step	4	4	4	4	4	4	4	4
Chunk Size	5	5	5	5	5	5	5	5
Dynamic Batching Settings								
Max Batch Number	-	-	1	-	1	-	1	1
Latency Threshold	-	-	-	-	-	-	-	-
Asynchronous Training Settings								
Rollout Async	Off	Off	On	Off	On	Off	On	On
Train Async	Off	Off	Off	Off	Off	Off	Off	On
Metric								
Throughput	191.51	187.76	170.61	178.76	155.82	143.10	154.40	255.58

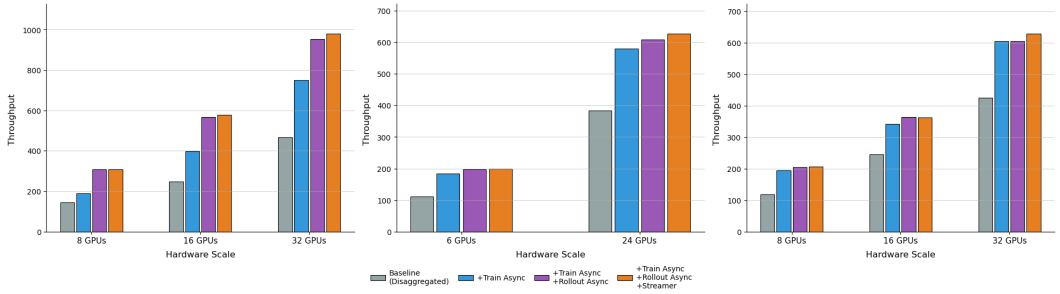

Figure 10: LIBERO + OpenVLA-OFT. The left, center, and right panels represent rollouts with GPU allocation ratios of 1:1, 2:1, and 3:1 between rollout workers (simulator and generator) and trainer, respectively.

ManiSkill ($\pi_{0.5}$, GRPO). ManiSkill (Mu et al., 2021) combines very large vectorized environment counts with GPU-accelerated simulation. As shown in Table 2, we can set a very large environment batch size: up to 256 (2048/8) for colocated placement and 272 ($3276/(6*2)$) for hybrid placement. Under colocated placement, it is interesting to see that the throughput of the synchronous baseline is already the best. The reason is that the generator's throughput already reaches its peak with respect to batch size. Therefore, there is no benefit from increasing the batch number and aggregating different environment batches for one generator inference. In this case, the increased context-switching overhead from adding asynchrony to the rollout phase dominates any benefit from the asynchronous rollout design. On the other hand, in hybrid placement, both rollout asynchrony and train asynchrony from RL-VLA^3 improve throughput. Notably, the highest throughput is

Table 3: Detailed hyperparameters and throughput for different experimental setups on RoboCasa+π_0 with PPO.

Method	Baseline	Baseline	RL-VLA^3	RL-VLA^3	Baseline	RL-VLA^3	RL-VLA^3	RL-VLA^3	RL-VLA^3
Placement Settings									
Simulator	0-7	0-7	0-7	0-7	0-5	0-5	0-5	0-5	0-5
Generator	0-7	0-7	0-7	0-7	6-7	6-7	6-7	6-7	6-7
Trainer	0-7	0-7	0-7	0-7	0-7	0-7	0-7	0-7	0-7
Environment Settings									
Total Env	160	160	160	160	168	168	168	168	168
Batch Number Per GPU	1	2	2	2	2	2	2	2	2
Max Episode Steps	320	320	320	320	320	320	320	320	320
Training Hyperparameters									
Rollout Epoch	4	4	4	4	4	4	4	4	4
Update Epoch	2	2	2	2	2	2	2	2	2
Micro Batch Size	32	32	32	32	32	32	32	32	32
Global Batch Size	1024	1024	1024	1024	1024	1024	1024	1024	1024
Model Specifications									
Model Num Step	5	5	5	5	5	5	5	5	5
Chunk Size	10	10	10	10	10	10	10	10	10
Dynamic Batching Settings									
Max Batch Number	-	-	1	1	-	1	2	1	2
Latency Threshold (ms)	-	-	-	-	-	-	200	-	200
Asynchronous Training Settings									
Rollout Async	Off	Off	On	On	Off	On	On	On	On
Train Async	Off	Off	Off	On	Off	Off	On	Off	On
Metric									
Throughput	409.6	379.25	437.60	505.68	332.92	355.64	349.32	454.63	455.01

achieved when both are enabled. The improvement is about 78.6% over the synchronous hybrid baseline and 33.5% over the synchronous colocated baseline.

RoboCasa (π_0, PPO). RoboCasa (Nasiriany et al., 2024) is a simulation environment focused on manipulation tasks in diverse visual scenes, producing high-resolution observations. Although the GPU memory occupation of a single RoboCasa environment is low, the computational cost of physics simulation is resource-intensive, making it difficult to scale up the environment batch size per GPU. As reported in Table 3, we configured a total of 160 environments for colocated placement and 168 for hybrid placement. In the colocated setup, the best synchronous baseline achieves 409.6 with a single environment batch per GPU. Simply increasing the number of batches decreases throughput to 379.25. However, RL-VLA^3 improves throughput by 15.4% with rollout asynchrony and 33.3% with full asynchrony. For hybrid placement, the benefit of enabling training asynchrony is more significant. However, the throughput of RL-VLA^3 with hybrid placement is lower than with colocated placement. This is again due to the characteristics of RoboCasa: GPU memory occupation is relatively low, but the number of environments is limited by CPU resources. Thus, having each GPU host all three resource groups makes better use of GPU memory and reduces communication overhead between GPUs.

LIBERO (GR00T N1.5, PPO). LIBERO (Liu et al., 2023) is representative of CPU-heavy MuJoCo-style stepping. Therefore, all runs in Table 4 use colocated placement on GPUs, isolating how batching and asynchrony interact when the simulator, generator, and trainer time-share the same devices. We mainly vary the number of environment batches per GPU while maintaining the same total environment count of 256. As shown in the table, since the baseline training adopts a synchronous pipeline, increasing the batch number per GPU does not yield a clear improvement in throughput (217.27 for 1, 242.30 for 2, and 240.13 for 4). However, by introducing the asynchronous rollout design, RL-VLA^3 significantly increases

Table 4: Detailed hyperparameters and throughput for different experimental setups on LIBERO+GR00T N1.5 with PPO.

Method	Baseline	Baseline	RL-VLA^3	RL-VLA^3	Baseline	RL-VLA^3	RL-VLA^3
Placement Settings							
Simulator	0-7	0-7	0-7	0-7	0-7	0-7	0-7
Generator	0-7	0-7	0-7	0-7	0-7	0-7	0-7
Trainer	0-7	0-7	0-7	0-7	0-7	0-7	0-7
Environment Settings							
Total Env	256	256	256	256	256	256	256
Batch Number Per GPU	1	2	2	2	4	4	4
Max Episode Steps	480	480	480	480	480	480	480
Training Hyperparameters							
Rollout Epoch	4	4	4	4	4	4	4
Update Epoch	2	2	2	2	2	2	2
Micro Batch Size	32	32	32	32	32	32	32
Global Batch Size	1024	1024	1024	1024	1024	1024	1024
Model Specifications							
Model Num Step	4	4	4	4	4	4	4
Chunk Size	5	5	5	5	5	5	5
Dynamic Batching Settings							
Max Batch Number	-	-	1	1	-	1	1
Latency Threshold	-	-	-	-	-	-	-
Asynchronous Training Settings							
Rollout Async	Off	Off	On	On	Off	On	On
Train Async	Off	Off	Off	On	Off	Off	On
Metric							
Throughput	217.27	242.30	297.38	349.01	240.13	344.47	423.39

throughput by 22.7% and 43.5% with 2 and 4 batches per GPU, respectively. This means that inference time can be effectively hidden by simulator stepping time, consistent with the fact that stepping time is much longer than inference time in MuJoCo-style environments. When we further activate training asynchrony, throughput can be improved to 423.39, which is about a 74.7% improvement over the best synchronous performance.

Meta-World (π_0, PPO). Meta-World (Yu et al., 2020) is more parallelizable than LIBERO and RoboCasa. Therefore, we can set a larger total environment count (512 for colocated and 768 for hybrid). As shown in Table 5, Meta-World exhibits similar trends to LIBERO in colocated placement: the synchronous baseline does not benefit from splitting environments into multiple batches, but RL-VLA^3 significantly improves throughput by activating rollout and training asynchrony. The best configuration uses 4 batches per GPU with full asynchrony, achieving 268 in throughput, about an 85.2% improvement over the best synchronous performance. Moreover, for hybrid placement, the throughput of RL-VLA^3 achieves about an 80.9% improvement over the synchronous baseline, showing that asynchronous design also helps mitigate communication overhead and rebalancing costs in hybrid placement.

A.2 Ablation on Different Ratios of GPU Allocation

As shown in the detailed results in Section A.1, the GPU resource ratio for hybrid placement is 3:1 between the simulator and the generator, i.e., for a total of 8 GPUs, we let 6 GPUs load environments and 2 GPUs load a model. Apart from this 3:1 ratio, we also evaluate the performance of RL-VLA^3 under 1:1 and 2:1 ratios. We use a different hybrid placement here for the LIBERO environment: we separate all GPUs into rollout workers and the trainer. For example, with a 1:1 ratio on an 8-GPU node, we let 4 GPUs simultaneously load an environment and a model, while the other 4 GPUs load only a model for training. Figure 10 presents the results for this placement on LIBERO + OpenVLA-OFT with different ratios;

Table 5: Detailed hyperparameters and throughput for different experimental setups on MetaWorld+π_0 with PPO.

Method	Baseline	Baseline	RL-VLA^3	RL-VLA^3	Baseline	RL-VLA^3	RL-VLA^3	Baseline	RL-VLA^3	RL-VLA^3
Placement Settings										
Simulator	0-7	0-7	0-7	0-7	0-7	0-7	0-7	0-5	0-5	0-5
Generator	0-7	0-7	0-7	0-7	0-7	0-7	0-7	6-7	6-7	6-7
Trainer	0-7	0-7	0-7	0-7	0-7	0-7	0-7	0-7	0-7	0-7
Environment Settings										
Total Env	512	512	512	512	512	512	512	768	768	768
Batch Number Per GPU	1	2	2	2	4	4	4	2	2	2
Max Episode Steps	100	100	100	100	100	100	100	100	100	100
Training Hyperparameters										
Rollout Epoch	4	4	4	4	4	4	4	4	4	4
Update Epoch	2	2	2	2	2	2	2	2	2	2
Micro Batch Size	128	128	128	128	128	128	128	128	128	128
Global Batch Size	2048	2048	2048	2048	2048	2048	2048	2048	2048	2048
Model Specifications										
Model Num Step	4	4	4	4	4	4	4	4	4	4
Chunk Size	5	5	5	5	5	5	5	5	5	5
Dynamic Batching Settings										
Max Batch Number	-	-	1	1	-	1	1	-	1	1
Latency Threshold	-	-	-	-	-	-	-	-	-	-
Asynchronous Training Settings										
Rollout Async	Off	Off	On	On	Off	On	On	Off	On	On
Train Async	Off	Off	Off	On	Off	Off	On	Off	Off	On
Metric										
Throughput	130.01	143.97	165.54	209.81	144.70	197.27	268.00	92.29	113.62	166.94

the left, center, and right panels correspond to rollout-side allocation ratios of 1:1, 2:1, and 3:1 between rollout workers and the trainer. Furthermore, we decouple full asynchrony into train async and rollout async, and evaluate how different levels of asynchrony influence throughput. Train async is further decoupled into basic training asynchrony and streamer training: basic training still waits for a full epoch of rollout data before starting training, while streamer training can start immediately after receiving the first batch of rollout data.

As shown in the figure, different placement ratios show similar trends, which is also consistent with our analysis in Section 4.3: train async tends to yield a larger step-change in throughput than rollout async, and the combination of both yields the best performance. The improvement from rollout async is more significant at the 1:1 ratio than at others. This is because the 1:1 ratio allocates fewer GPU resources to rollout workers, whereas asynchrony helps better utilize GPU resources by overlapping inference time with stepping time. Streamer training alone does not show significant improvement because the current rollout trajectory has a fixed length, so all training data from the same rollout epoch arrives at roughly the same time.

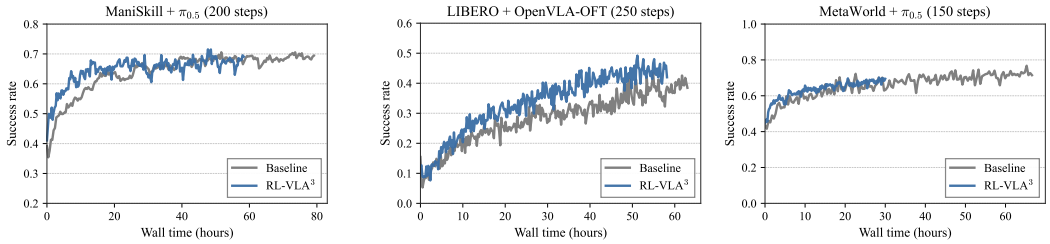

Figure 11: Success-rate curves for ManiSkill+$\pi_{0.5}$, LIBERO+OpenVLA-OFT, and Meta-World+$\pi_{0.5}$ under the synchronous baseline versus RL-VLA^3. The trajectories are aligned by environment steps, confirming comparable sample efficiency, while RL-VLA^3's higher throughput substantially reduces the wall-clock time needed to reach comparable success levels.

A.3 Additional Success Rate Curve

We present an alternative version of Figure 6, where we change the x-axis from training steps to wall-clock time. As shown in Figure 11, under the same configuration, RL-VLA^3 significantly improves training speed without noticeably compromising training stability or convergence compared to the baseline.

A.4 More Metrics: CPU/GPU Utilization

In this section, we present additional evaluation metrics, including CPU utilization, RAM usage, GPU utilization, and GPU memory usage. We plot the entire *first* training step, so the elapsed times differ. From the usage curves, it becomes clearer how RL-VLA^3's implementation improves throughput over the synchronous baseline. A representative LIBERO case is discussed in Section 4.4; here we provide the remaining environments and extended discussion.

ManiSkill ($\pi_{0.5}$, GRPO). For ManiSkill, we select two representative cases: colocated placement for the baseline (first column of Table 2) and hybrid placement for RL-VLA^3 (last column of Table 2), which achieve the highest throughput under each method. For the Hybrid placement, the total resources are divided into two resource groups: each GPU in resource group 0 hosts one Simulator and one Trainer, while each GPU in resource group 1 hosts one Generator and one Trainer. For the colocated placement, each GPU hosts one Simulator, one Generator, and one Trainer.

For RL-VLA^3, the noticeable increase in GPU memory usage arises because the data from the first rollout enters the training engine. Due to asynchronous training and inference, the GPU has already started training, thereby increasing memory consumption. In contrast, for the baseline, training starts only after all rollout data have been collected, so the peak GPU memory usage occurs much later. Moreover, we observe that because the baseline transfers all data in one batch after collection is complete, there is a pronounced spike in host memory usage and GPU waiting at around 23 minutes. This happens because the training data of the VLA includes large inputs such as images, which consume considerable memory and time when the dataset is large. Asynchronous training–inference means we transfer data in batches; after each training step the data can be discarded (following pre-defined rules), which reduces the burden on the system.

LIBERO (GR00T N1.5, PPO) and Meta-World (π_0, PPO). We discuss LIBERO and Meta-World together because both selected cases use colocated placement, allowing us to compare resource utilization between the synchronous baseline and RL-VLA^3. The specific configurations are those that achieve the highest throughput for the Baseline and RL-VLA^3 in Table 4 and Table 5, respectively.

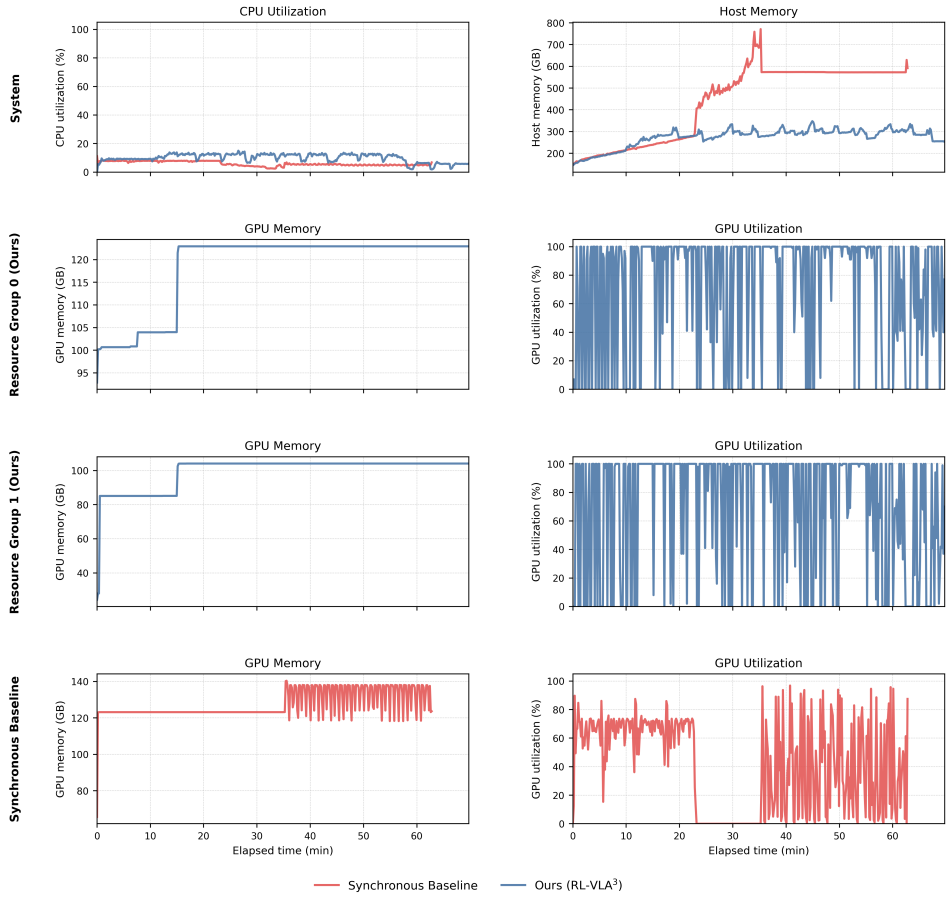

Figure 12: Comparison of resource utilization during the first step between the Colocated (Baseline) and Hybrid (Ours) on ManiSkill+$\pi_{0.5}$. Note that in this step, Hybrid (Ours) processes approximately 1.5$\times$ the data volume of Colocated (Baseline), resulting in a longer elapsed time.

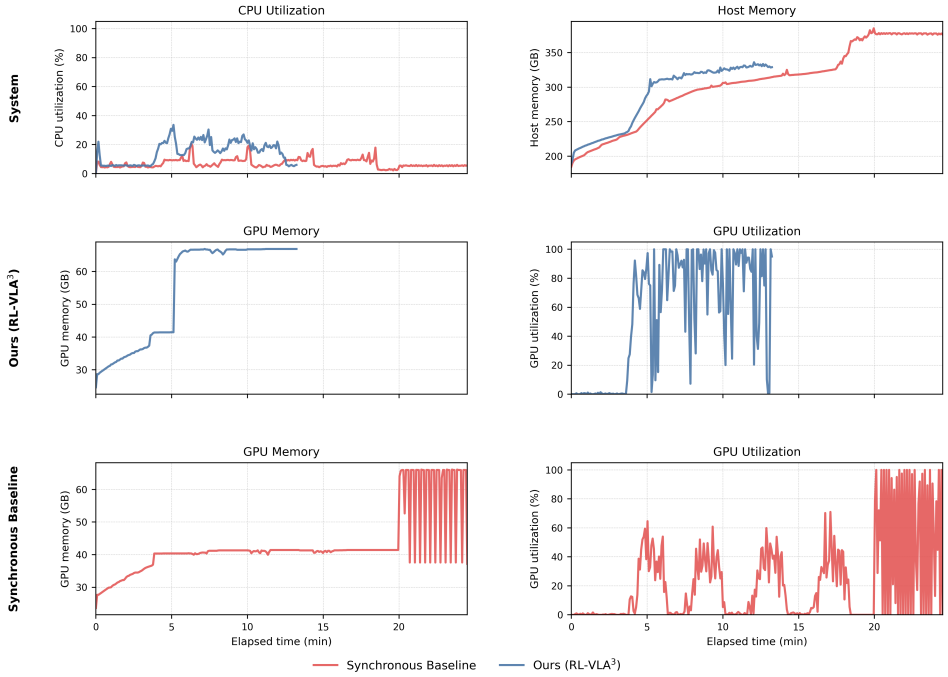

Figure 13: Comparison of resource utilization during the first step between Colocated (Baseline) and Colocated (Ours) on MetaWorld+$\pi_{0.5}$.

As can be seen, Figures 9 and 13 exhibit very similar comparative effects. We can clearly observe that RL-VLA^3 starts training immediately after the first rollout finishes, rather than waiting for all rollouts to complete. At the same time, similar to the ManiSkill case, asynchronous training–inference with batched data transfer reduces the high GPU waiting time and high CPU load caused by one-shot transfers. Another interesting observation is that, for the Meta-World baseline strategy, we see noticeable GPU idle periods between the four rollout epochs. This is likely because we increased the number of parallel environments per rollout to boost throughput, making the initialization phase very time-consuming. This suggests that we could leverage the environment initialization phase to use GPU resources more efficiently for training, thereby improving throughput.