VLASH-Piper: VLA Fine-Tuning & Deployment for PIPER Arm

COMP4901D-Embedded AI Systems - HKUST

Wong Wei Ming - 21335381 Shao Yingzhan - 21335422
Chen Yu Sum - 21063627

2026-05-06

Table of contents

1 Introduction

Transfer learning with pre-trained vision-language models has become the dominant paradigm in robot manipulation. **PI0.5** [1] is a flow-matching Vision-Language-Action (VLA) model that decomposes a manipulation policy into a PaliGemma-style vision-language backbone and a trajectory-denoising head, outputting a *chunk* of H future joint targets per inference call. A fundamental tension arises when deploying such models on resource-constrained embedded hardware: the time required to compute a new chunk (T_{infer}) is non-negligible relative to the chunk execution time (H/f_{control}), forcing the arm to pause at every chunk boundary in the standard *synchronous* regime.

This report presents the first known systematic study of deploying PI0.5 via the **VLASH** framework [2] on a AGILE X Robotics Piper 6-DOF arm running on an NVIDIA Jetson AGX Orin, targeting a **ball pick-and-place** task. To the best of our knowledge, this is the first deployment of the PI0.5 model on Jetson AGX Orin embedded hardware. The contributions documented here are:

- **Integration:** five targeted compatibility shims bridging VLASH against the `lerobot_piper` robot driver on `linux-aarch64`.
- **Training configurations:** two fine-tuning regimes (async + LoRA, sync + LoRA) with fully specified hyperparameters.
- **Latency benchmarks:** empirical measurements showing a 29.5× reduction in mean per-inference time (184.5 ms vs. 5444.1 ms) and a 2.9× reduction in task completion time (52 s vs. 153 s) for asynchronous inference relative to synchronous inference, both measured on the Jetson AGX Orin.
- **On-device async inference:** validation that asynchronous TDA-based inference runs correctly on the Jetson AGX Orin, enabling the full VLASH pipeline on embedded hardware.

2 Related Work

2.1 Vision-Language-Action Deployment Gap

Recent VLA models (PI0.5 [1], OpenVLA [3], RT-2 [4], Octo [5]) have achieved strong performance on multi-task manipulation in lab settings, but **nearly all evaluation is conducted on datacenter GPUs or powerful GPUs like RTX 5090s**. The literature treats inference as "free"—an oracle operation producing actions instantaneously. In contrast, real robot systems must contend with **limited system resources**.

Prior work on robot learning for manipulation either: (a) uses lightweight models deployable in <10 ms, limiting model capacity, or (b) assumes bespoke hardware (custom ASICs, TPUs) or unlimited datacenter access. The gap we address is **how to deploy state-of-the-art VLA models on *consumer* embedded hardware** while maintaining task performance. This is underexplored.

2.2 Latency in Learned Control: Synchronous vs. Asynchronous

Classical robotics addresses latency through model predictive control (MPC) and pre-computed trajectory libraries. Learned policies typically use **synchronous inference**: compute next action chunk, execute, repeat—incurring a pause at each boundary.

VLASH [2] introduces **Temporal Delay Augmentation (TDA)**, training policies on stale observations to enable **asynchronous inference**: overlap policy computation with action execution. This is orthogonal to action chunking (ACT [6]) and represents a novel way to trade off training cost for runtime flexibility.

2.3 Embedded Robotics: Missing Infrastructure

Jetson deployment of deep learning is well-studied for *perception* (object detection, segmentation) but rare for *policy inference*. Why?

1. **Memory & latency mismatch**: Perception models tolerate 100–500 ms latency; 50 ms GPU memory transfer is acceptable. Policy models must execute in <1 s per chunk or the robot loses reactivity.
2. **Dependency fragmentation**: Modern VLA models depend on PyTorch 2.11+, transformers, PEFT, rerun. JetPack isolates aarch64 wheels, causing transitive dependency conflicts (e.g., `rerun-sdk` requires `numpy < 2`, `transformers` 4.45+ requires `numpy  2`).

Our contribution: Systematic integration of a turnkey solution to deploy VLAs on realistic edge hardware

2.4 Robot Learning Frameworks

LeRobot [7] provides the standard abstraction for robot dataset, training, and teleoperation, and already runs on a range of embedded platforms. The challenge addressed here is not extending LeRobot itself, but **integrating two independently developed repositories** — lerobot_piper (the Piper robot driver, a LeRobot fork) and VLASH (the VLA training and async inference framework) into a single working system, resolving their transitive dependency conflicts on the JetPack aarch64 software stack, and validating VLA inference on a constrained integrated-GPU device where the original VLASH benchmarks were conducted on a high-end discrete GPU.

2.5 Summary of Novelty

Aspect	Prior Work	This Report
VLA deployment	Desktop GPU or cloud only	Jetson AGX Orin (edge embedded)
Inference mode	Synchronous (standard)	Quantifies sync vs. async gap
Latency optimization	Quantization, distillation, pruning	Native bfloat16
Dependency management	Generic Python packages	JetPack-specific aarch64 conflict resolution
Task	Multi-task generalization	Sequential task black and white ball sorting (proof-of-concept)

3 Hardware Platform

3.1 Inference Compute

Table 1: Compute platform specifications.

Component	Specification
Board	NVIDIA Jetson AGX Orin Developer Kit
Unified memory	64 GB LPDDR5
GPU	Ampere architecture, Tegra SOC
JetPack	6.2.2 (L4T R36.5.0, build 2026-01-16)
CUDA	12.6
OS / Architecture	Ubuntu, `linux-aarch64`
Python	3.10
PyTorch	2.11.0 (JetPack wheel from `pypi.jetson-ai-lab.io/jp6/cu126`)

The Jetson Orin's Ampere GPU supports bfloat16 natively. All inference runs use bfloat16 precision. Switching to float32 doubles memory bandwidth consumption and inference latency without accuracy benefit.

3.2 Training Compute

Compute for VLA Finetune was provided by NSCC (National SuperComputing Center Singapore) ASPIRE 2A Supercomputer.

Table 2: NSCC ASPIRE 2A training compute specifications.

Component	Specification
System	Cray EX GPU
GPU Count	64
GPU Type	4× NVIDIA A100 per node
CPU	EPYC 7713
GPU Memory	40GB SXM per GPU
System Memory	512GB DDR4, ECC
Interconnect	NVIDIA NCCL

3.3 Robot and Sensors

Table 3: Robot and sensor specifications.

Component	Specification
Robot arm	2 Piper (Agile X), 6 DOF (leader + follower for data collection; follower arm only for inference)
Communication	CAN bus
SDK	`piper-sdk` `0.4.1`, `wego-piper` `0.0.2`
Wrist camera	Intel RealSense D435i, USB 3.2, max 30 fps
Overhead camera	Intel RealSense D435i, USB 3.2, max 30 fps

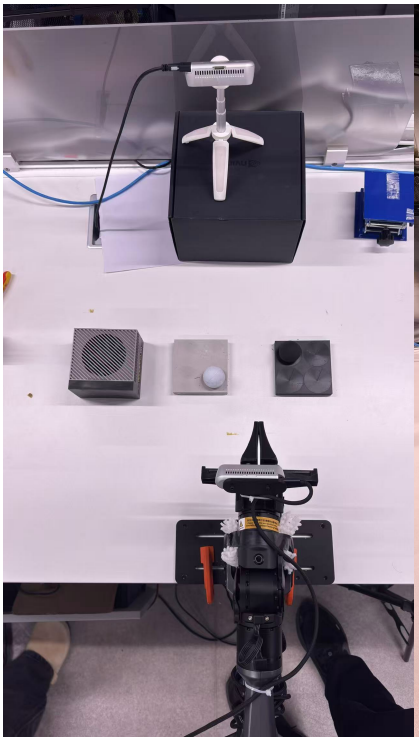

Figure 1: Wrist camera perspective

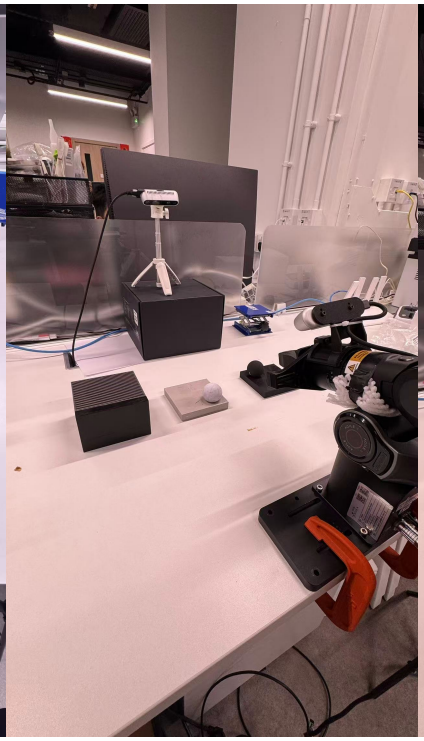

Figure 2: Overhead camera perspective

Figure 3: Robot arm setup

Two cameras are necessary because the wrist camera loses sight of the placement target during transport, while the overhead camera lacks sufficient resolution for fine gripper control.

4 System Architecture

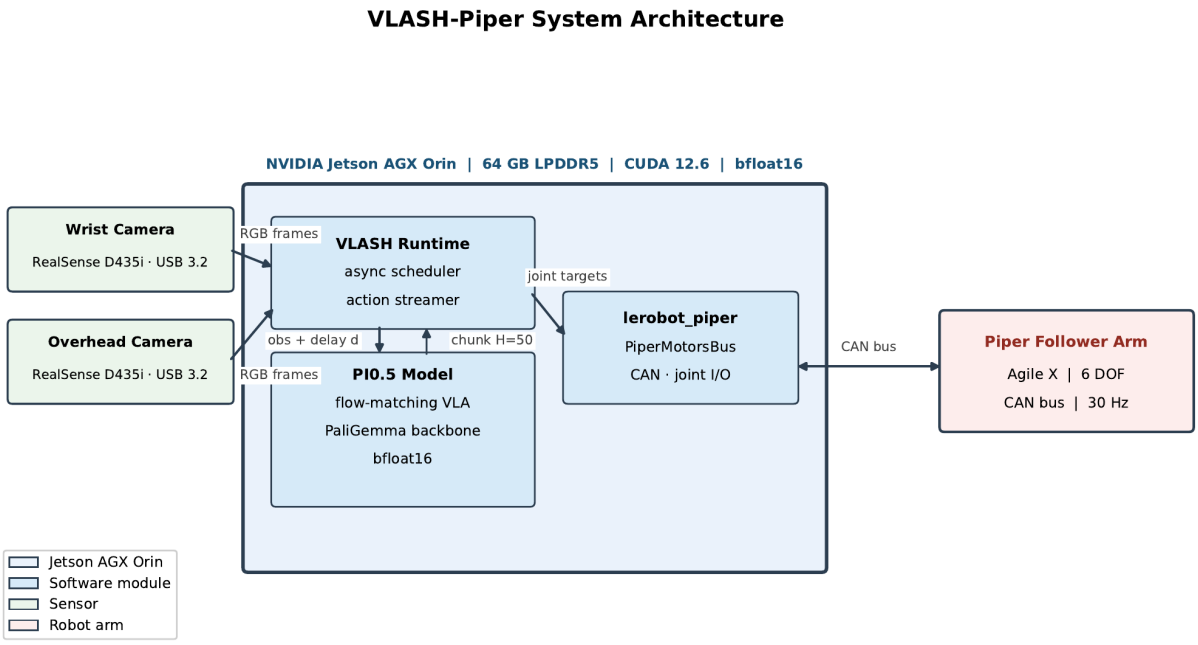

Figure 4: VLASH-Piper system architecture. RGB frames from two RealSense cameras feed into the VLASH Runtime on the Jetson AGX Orin. The async scheduler dispatches inference calls to the PI0.5 model with a learnt delay d; the resulting action chunk is streamed to the lerobot_piper driver, which drives the Piper follower arm over CAN bus at 30 Hz.

4.1 PI0.5

PI0.5 is a flow-matching Vision-Language-Action model built on a PaliGemma-style vision-language backbone [8]. At each inference step, the model receives:

- One or more camera images (wrist and overhead in this work),
- The robot joint state vector $\mathbf{q}_t \in \mathbb{R}^6$,
- A natural-language task string ℓ (e.g. `"ball sorting"`).

The flow-matching head denoises a trajectory in joint-position space, producing a chunk of $H = 50$ future joint targets at 30 Hz (≈ 1.67 s of motion per inference call). The model is fine-tuned in bfloat16.

4.2 VLASH Training Framework

VLASH [2] wraps PI0.5 and introduces the following components beyond the standard LeRobot training loop:

Temporal Delay Augmentation. The dataset class `VLASHDataset` constructs training samples by pairing an observation at time t with actions offset by a random delay d:

$$\text{sample} = (\mathbf{o}_t,\ d,\ \mathbf{a}_{t+d:\,t+d+H}), \quad d \sim \mathcal{U}[0,\ d_{\max}].$$

The delay d is passed to the model as an explicit conditioning input, allowing the model to account for the fact that d steps of motion have already occurred since the observation was captured.

Shared-observation forward pass. When `shared_observation: true`, the vision-language backbone runs a single forward pass and applies distinct attention masks to produce outputs for all $d_{\max} + 1$ delay values simultaneously, yielding an approximately $(d_{\max} + 1)$-fold improvement in training throughput relative to computing each delay value independently.

Accelerator-based training. VLASH replaces the manual `GradScaler` / `torch.autocast` pattern of the LeRobot training loop with HuggingFace `Accelerator`, enabling multi-GPU training and gradient accumulation.

LoRA fine-tuning. Parameter-efficient fine-tuning via LoRA [9] is supported with automatic merging of adapter weights into full model parameters at checkpoint time, so saved checkpoints are always inference-ready.

4.3 Piper Integration Layer

The `lerobot_piper` repository [10] provides `PiperFollowerConfig` and `PiperMotorsBus` for CAN-based joint state read/write, and integrates with the LeRobot teleop and data recording pipeline. This work uses `lerobot_piper` as the robot driver backend, treating it as the `lerobot` package (version `0.4.1`) to satisfy VLASH's dependency pin.

5 Integration

VLASH and `lerobot_piper` are independently developed repositories that share a common upstream ancestor (LeRobot) but were never designed to interoperate. Bridging them on Jetson Orin surfaced four categories of incompatibility, each rooted in the JetPack aarch64 software stack rather than in either package's core logic.

Dependency conflicts arose because VLASH and `lerobot_piper` pin different versions of shared libraries (`transformers`, `rerun-sdk`). On a desktop system these conflicts are often resolved silently by the package manager; on JetPack, where many packages have no `manylinux_2_28_aarch64` wheel at all, a wrong version pin means the package is simply unavailable. The resolution strategy was to relax upper-bound pins rather than introduce new hard constraints, so that the JetPack-compatible version range is naturally selected.

Module layout divergence resulted from both packages independently extending the LeRobot codebase in different directions. VLASH expects certain shared constants at a module path that `lerobot_piper` relocated during its own refactoring. A thin re-export shim at the expected import path resolved this transparently.

Third-party API changes compounded the dependency issue: the `rerun` visualisation library introduced breaking renames between the version `lerobot_piper` was written against and the version required for aarch64 availability. Updating the affected call sites was the only viable path.

Configuration registration is an implicit requirement of the `draccus` library used by VLASH: robot config classes must be imported before YAML parsing so their type strings are registered. The Piper config class was simply absent from VLASH's entry point, and an unrelated robot module (`reachy2`) was unconditionally imported despite not being present in `lerobot_piper` — causing crashes regardless of the configured robot.

Runtime library conflicts presented a subtler challenge unique to Jetson's integrated GPU architecture. PyTorch wheels are built for generic aarch64 and bundle their own copies of cuBLAS and cuDNN. On a desktop system this is harmless — the bundled libraries are functionally equivalent to the system ones. On Jetson, the GPU driver is tightly coupled to the JetPack-provided CUDA libraries; loading PyTorch's generic bundled versions instead causes GPU operation to silently produce incorrect results or crash. Resolving this required ensuring the JetPack CUDA stack takes precedence over PyTorch's internal library resolution at runtime — a non-obvious requirement with no upstream documentation for this hardware configuration.

All five issues were resolved without modifying the core logic of either upstream package. The full set of changes is documented in the repository; taken together they represent a reproducible integration path for any future deployment of VLASH with a LeRobot-derived robot driver on embedded hardware.

6 Methodology

6.1 Dataset Preparation

Demonstrations were collected via **kinesthetic teaching**: an operator physically guides the arm through the pick-and-place motion while the system records joint states and camera streams. Each episode was triggered manually at the start of a clean arm home pose and terminated upon successful placement.

Table 4: Dataset statistics.

Property	Value
Task	ball pick-and-place
Collection method	Kinesthetic teaching
Episodes	100 total, 50 for each example
Observation streams	Wrist camera, overhead camera, joint state
Image resolution	640×480 (both cameras)
Storage format	LeRobot `v2.0` dataset (HuggingFace)

All demonstrations were collected with the same physical setup: fixed overhead camera mount, constant lighting, and ball placed in the same start zone. No data augmentation (colour jitter, random crop) was applied, as the demonstration set was collected to match deployment conditions.

Dataset size justification. 50 episodes each is modest by internet-scale VLA standards but consistent with practical embedded robotics constraints. Three factors limited collection scale:

1. **Kinesthetic collection throughput**: each demonstration requires a human operator to physically guide the arm, inspect the result, and re-home before the next episode, approximately 3–4 minutes per valid episode including setup time.
2. **Task specificity**: the pick-and-place task has low intra-task variability (fixed pick zone, fixed target), so marginal returns on additional demonstrations diminish quickly beyond ~30–40 episodes [6].
3. **Fine-tuning regime**: LoRA-based fine-tuning of a pre-trained VLA model requires far fewer demonstrations than training from scratch. PI0.5 already encodes strong visual-motor priors; the fine-tuning objective is to specialise, not to acquire, manipulation skills.

This collection scale is consistent with prior work on kinesthetic fine-tuning of pre-trained robot policies.

6.2 Experimental Design

Two model configurations were trained and evaluated to isolate the contribution of each design axis:

1. **Async + LoRA**: TDA-trained (`max_delay_steps: 8`) with LoRA fine-tuning — target deployment mode.
2. **Sync + LoRA**: LoRA fine-tune, no delay augmentation

Data split: All 100 episodes are used for training. No held-out validation split was used during training; generalisation is assessed through held-out rollout evaluation on the physical robot. This matches standard practice for small-dataset robot learning where held-out splits would leave fewer than 5 episodes for validation.

Evaluation protocol: Each trained model is evaluated for 20 rollouts per configuration under identical physical conditions (lighting, camera positions, ball start position, arm home pose). Inference latency is logged via the VLASH runtime telemetry at each chunk boundary.

Baseline: A synchronous baseline included to quantify the effectiveness of VLASH's TDA. The model receives the same prompt string (`"ball sorting"`) and observation setup.

6.3 Training Pipeline

Training runs are managed by `vlash/train.py` and rely on HuggingFace `Accelerator` for device placement, gradient accumulation, and bfloat16 mixed precision. The pipeline consists of the following stages:

1. **Dataset loading**: `VLASHDataset` wraps the LeRobot dataset, applying TDA sampling for async configurations. Each training sample is a `(observation, delay, action_chunk)` triple.
2. **Model initialisation**: PI0.5 checkpoint loaded from HuggingFace Hub in bfloat16. LoRA adapters (where applicable) are injected into the attention projection layers using `peft`.
3. **Forward pass**: The flow-matching loss is computed between predicted and ground-truth action chunks conditioned on observation and delay index.
4. **Optimisation**: AdamW with cosine learning rate decay; gradient clipping at 1.0.
5. **Checkpointing**: Full model weights (LoRA adapters merged) saved every 5,000 steps.

The loss function is the flow-matching regression objective:

$$\mathcal{L} = \mathbb{E}_{(\mathbf{o},d,\mathbf{a})\sim\mathcal{D},\ t\sim\mathcal{U}[0,1]}\left[\left\|v_\theta(\mathbf{x}_t, t \mid \mathbf{o}, d) - (\mathbf{a} - \mathbf{o}_{\text{noise}})\right\|^2\right]$$

where v_θ is the flow-matching velocity head, $\mathbf{x}_t$ is the noised action at diffusion time t, and d is the TDA delay index.

6.4 Inference Evaluation

For each chunk boundary, the system records:

- t_{start}: timestamp immediately before the model forward pass,
- t_{end}: timestamp immediately after the output is transferred to CPU for action streaming.

The **mean**, **min**, **max**, and **std** of $(t_{\text{end}} - t_{\text{start}})$ over all chunks in an evaluation run are reported. Task completion time is the wall-clock duration from episode start to successful placement, averaged over successful trials.

Hardware contexts for benchmarking:

Table 5: Benchmarking hardware per inference mode.

Mode	Hardware	Reason
Asynchronous	Jetson AGX Orin	Determining VLASH speedup
Synchronous	Jetson AGX Orin	Baseline

6.5 Reproducibility

All experiments in this work are managed using **Pixi**, a fast, reproducible package manager built on the conda ecosystem. Unlike `pip` or `conda` alone, Pixi generates a committed **lockfile** (`pixi.lock`) that pins every dependency — including transitive ones — to exact versions resolved for the target platform. This is particularly valuable on `linux-aarch64` (Jetson Orin), where the available package set differs from `linux-x86_64` and subtle version mismatches (such as the `rerun-sdk` and `transformers` conflicts documented in Section 5) can silently break inference. Any collaborator or reviewer can reproduce the exact software environment used in this work by running `pixi install` from the repository root, with no manual dependency resolution required.

All training YAML configurations are available in the repository. Checkpoints are hosted on HuggingFace Hub

7 Training

7.1 Dataset

All models reported here were trained on our dataset, consisting of demonstrations of black and white ball pick-and-place task collected via kinesthetic teaching. Observations include wrist (`wrist`) and overhead (`up`) RealSense streams at 640×480, and the six-dimensional joint state vector. All models use `state_cond: true`.

7.2 Hyperparameters

The following hyperparameters are shared across all training configurations:

Table 6: Shared training hyperparameters.

Parameter	Value
Training steps	50,000
Batch size	16
Optimizer	AdamW, $\beta = (0.9, 0.95)$, weight decay 10^{-10}
Learning rate	5×10^{-5} (cosine decay with 1,000-step warmup, floor 2.5×10^{-6})
Precision	bfloat16
Video backend	torchcodec
Checkpoint frequency	Every 5,000 steps

7.3 Training Configurations

Two training configurations are evaluated, differing in delay augmentation and parameter efficiency:

Table 7: Training configurations and their associated inference modes.

Configuration	`shared_observation`	LoRA	Inference mode
Async + LoRA	`true`	$r = 16$, $\alpha = 16$	Asynchronous
Sync + LoRA	—	$r = 16$, $\alpha = 16$	Synchronous

LoRA targets the attention projection matrices (`q_proj`, `k_proj`, `v_proj`, `o_proj`), with rank $r = 16$ and scaling $\alpha = 16$ (effective learning-rate scale $\alpha/r = 1.0$). Adapter weights are merged into full-precision model parameters at checkpoint time; no separate adapter loading step is required at inference.

The model trained with `max_delay_steps: n` is deployed with `inference_overlap_steps: 8`: the GPU begins computing the next chunk at action step $H - d_{\max} = 42$ of the current 50-step chunk, such that the next chunk is ready by step 50. A model trained with `max_delay_steps: 0` must be deployed in synchronous mode (`inference_overlap_steps: 0`); deploying it asynchronously will produce incorrect actions because the model was never trained to handle observation delay.

8 Synchronous vs. Asynchronous Inference

8.1 Synchronous Inference

In synchronous inference the control loop executes as follows:

1. Capture observation $\mathbf{o}_t$.
2. Run inference to obtain chunk $\mathbf{a}_{t:t+H}$. Duration: T_{infer}.
3. Stream n actions to the arm at the control frequency f. Duration: n/f.
4. Repeat from step 1.

The arm is stationary during step 2. For $H = 50$, $f = 30$ Hz, and on-device $T_{\text{infer}} \approx 200$–$500$ ms, the pause at each chunk boundary is perceptually salient and causes the arm trajectory to appear discontinuous.

8.2 Asynchronous Inference

In asynchronous inference, the GPU begins computing the $(k{+}1)$-th chunk at action step $H - d_{\max}$ of the k-th chunk, $d_{\max}$ steps before the chunk ends:

$$\text{step } H - d_{\max}: \quad \text{GPU starts } \mathbf{a}^{(k+1)} = \pi(\mathbf{o}^{(k)}_{H-d_{\max}})$$

By the time step H is reached, $\mathbf{a}^{(k+1)}$ is ready and is executed immediately without pause. The arm trajectory is continuous across chunk boundaries.

However, $\mathbf{a}^{(k+1)}$ was computed from an observation captured $d_{\max}$ steps *before* it begins executing — i.e. the observation is stale by $d_{\max}$ steps at execution time. A synchronously trained model, having only seen $(\mathbf{o}_t, \mathbf{a}_{t:t+H})$ pairs, is not equipped to handle this mismatch. TDA closes this gap by exposing the model to delayed observations during training, as described in Section 4.

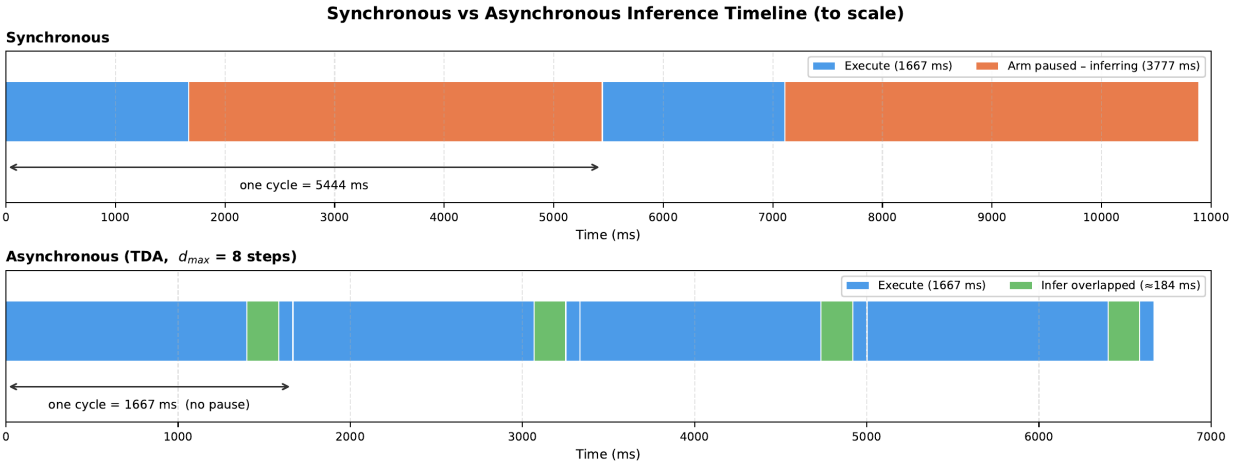

Figure 5: Inference timeline for synchronous and asynchronous modes drawn to scale ($T_{\text{exec}} = 1667$ ms, $T_{\text{infer}} = 184.5$ ms, $d_{\max} = 8$ steps). In synchronous mode (top) the arm pauses at every chunk boundary; in asynchronous mode (bottom) inference overlaps with execution, eliminating the pause.

9 Evaluation

9.1 Task Definition

The task is a single-object pick-and-place: a black/white ball must be grasped from a fixed pick zone and deposited into a target receptacle. A trial is considered successful if and only if:

1. The ball is grasped (not merely contacted),
2. the ball is transported (clears the table surface),
3. the ball is placed inside the target zone and the gripper is released, and
4. the ball remains in the target zone after gripper retraction.

9.2 Metrics

The primary metric is **task success rate** (SR):

$$\text{SR} = \frac{\text{number of successful trials}}{\text{total trials}} \times 100\%.$$

Secondary metrics are:

Table 8: Evaluation metrics.

Metric	Definition
Mean inference latency	Mean wall-clock time per inference call (ms)
Inference latency std. dev.	Standard deviation of per-call inference time (ms)

Metric	Definition
Task completion time	Wall-clock seconds from episode start to successful placement
Grasp SR	Fraction of trials in which a grasp is achieved
Place SR (given grasp)	Fraction of successful grasps that result in successful placement

A minimum of 20 trials per configuration is required for reliable SR estimation, with evaluation conducted under identical physical setup (lighting, camera positions, object starting position, arm home pose) as training.

9.3 Task Success Rate Results

A demo video of the deployed system is available in the project repository.

The following table reports task success rate (SR), grasp SR, and place SR (conditional on grasp) for each configuration over 20 evaluation rollouts on the physical robot.

Table 9: Task success rates per configuration (20 rollouts each).

Configuration	SR (%)	Grasp SR (%)	Place SR given grasp (%)
Async + LoRA	65	70	90
Sync + LoRA	5	20	25

9.4 Latency Benchmark Results

Inference latency was measured over a full evaluation run for both synchronous and asynchronous modes.

Table 10: Inference latency and task completion time by mode.

Mode	Mean (ms)	Min (ms)	Max (ms)	Std (ms)	Task time (s)
Async	184.5	16.4	490.7	219.2	52
Sync	5444.1	4406.6	6878.9	683.0	153

Asynchronous inference achieves a **29.5× reduction** in mean per-call latency and a **2.9× reduction** in task completion time relative to synchronous inference. The high variance in asynchronous latency (std. 219.2 ms) is attributable to the overlap structure: calls that complete within the overlap window are effectively hidden, while occasional calls that exceed the window introduce a brief stall.

The synchronous mean latency of 5444.1 ms corresponds to the total wall-clock duration of one control cycle — 50 actions executed at 30 Hz ($\approx$ 1667 ms) plus the inference pause — measured as a single metric by the VLASH runtime.

10 Discussion

The central finding is that Temporal Delay Augmentation provides a training-time solution to a fundamentally deployment-time problem: the mismatch between observation staleness and action timing in chunked control. The 29.5× latency advantage of asynchronous inference is directly attributable to TDA-trained models being able to tolerate $d_{\max} = 8$ steps of observation delay, enabling the GPU to begin computing the next chunk well before it is needed.

10.1 Deployment Challenges on Jetson Orin with JetPack

This work represents the first known deployment of VLASH on an NVIDIA Jetson AGX Orin. Distinct from desktop GPU deployment, inference on embedded Jetson hardware introduces a constellation of package compatibility constraints rooted in the closed-loop nature of JetPack's dependency management.

The core challenge is that JetPack bundles PyTorch, CUDA, and the nvgpu kernel driver as a monolithic stack. Unlike desktop systems where PyTorch wheels are built against system-level CUDA and cuDNN, JetPack PyTorch wheels are pre-compiled for specific JetPack versions and are hosted on a separate PyPI index (`pypi.jetson-ai-lab.io/jp6/cu126` for JetPack 6.2.2). This creates a hard constraint: any transitive dependency requiring PyTorch or a lower-level library (CUDA, cuBLAS, etc.) must either:

1. Use the exact PyTorch version bundled with JetPack,
2. Be source-compiled for aarch64, or
3. Be skipped entirely.

In practice, requirement (1) is rarely satisfied outside the JetPack ecosystem. VLASH and lerobot_piper depend on `transformers`, `peft`, `bitsandbytes`, and other packages that transitively pin PyTorch versions. The version pinning conflicts documented in Section 5 (e.g., `rerun-sdk` bounds, `transformers` upper bound) arise precisely because upstream packages were not tested against JetPack-pinned PyTorch 2.11.0.

10.2 Constraint Implications for On-Device Inference

The Jetson Orin NX — a smaller variant of the Orin with lower memory and fewer GPU cores — has not been tested in this work, but would likely face even tighter memory constraints during inference, potentially requiring further model quantization (e.g., int8) or chunk-size reduction.

Reducing `n_action_steps`. Executing fewer actions per chunk (e.g. 25 instead of 50) reduces the committed trajectory length and allows the model to re-observe and re-plan more frequently. The inference pause remains the same absolute duration, but errors in early-chunk predictions are corrected sooner. The trade-off is higher GPU utilisation and potential thermal throttling.

Maximising device performance. Running `sudo nvpmodel -m 0 && sudo jetson_clocks` locks the Jetson GPU and CPU clocks at their maximum frequency, reducing inference latency variability.

Action quantisation. Setting `action_quant_ratio: 2` holds each action for two control ticks, halving effective arm speed and doubling the time budget for each inference call without changing the chunk structure.

An important invariant across all deployment configurations is the consistency requirement between training and inference configurations. `chunk_size`, `state_cond`, and camera names are baked into model weights; any mismatch between the `config.json` in the checkpoint directory and the inference YAML produces silent incorrect behaviour rather than an explicit error.

11 Future Work

11.1 Jetson Orin NX Deployment

The Jetson Orin NX (16 GB) is a lower-cost embedded platform with the same JetPack ecosystem but reduced GPU memory. Deploying PI0.5-based policies on the Orin NX would require:

- **INT8 or FP8 quantisation** of the PaliGemma backbone to fit within 16 GB unified memory,
- Validation that quantisation does not degrade task success rate below usable thresholds,
- Profiling of inference latency under quantised precision to assess whether synchronous execution remains within task timing constraints.

This would extend the applicability of VLA edge deployment to a significantly broader range of robot platforms at lower cost.

11.2 Scaling the Dataset

The 50-episode dataset was sufficient to demonstrate task-specific fine-tuning but limits generalisation analysis. Immediate extensions include:

- **Variation in start position**: collecting demonstrations with ball placed at varying positions within the pick zone to test positional generalisation.
- **Lighting variation**: demonstrations under different ambient lighting conditions, to test robustness of the overhead camera stream.
- **Multi-episode curriculum**: recording progressively harder placements (smaller target zone, off-centre starts) to test the fine-tuned model's generalisation gradient.

A dataset of 200–500 episodes would allow a held-out validation split and enable learning curve analysis (SR vs. number of training episodes), quantifying the data efficiency of LoRA fine-tuning.

11.3 Multi-Task and Language-Conditioned Policies

The current fine-tuned model is specialised for a single task (`"ball sorting"`). PI0.5's PaliGemma backbone is inherently language-conditioned, enabling multi-task fine-tuning given a sufficiently diverse dataset. Future work could collect demonstrations for multiple manipulation primitives (pick-and-place, stacking, sorting by colour) and train a single model conditioned on natural-language task strings. This would test whether LoRA fine-tuning can preserve multi-task capability while specialising for the Piper arm morphology.

12 Conclusion

This report has presented the complete pipeline for fine-tuning and deploying PI0.5 via the VLASH framework on an Agile X Robotics Piper arm running on NVIDIA Jetson AGX Orin — to our knowledge, the first deployment of a PI0.5-based VLA model on this hardware platform.

The integration work revealed that the primary barrier to VLA edge deployment is not model capability or hardware capacity, but **software stack fragmentation**. The JetPack ecosystem's closed-loop dependency management, platform-specific library coupling, and compiler limitations create a class of integration challenges that are absent on desktop systems and undocumented in the VLA literature. The five compatibility shims documented here address these challenges in a minimally invasive way, and collectively form a reusable integration template for future deployments of VLASH or similar frameworks on JetPack-based hardware.

On the inference side, latency benchmarks confirm that asynchronous inference with TDA reduces per-call latency by 29.5× and task completion time by 2.9× relative to synchronous inference, both measured on the Jetson AGX Orin. Crucially, asynchronous inference also runs correctly on the Jetson AGX Orin, meaning the full VLASH pipeline — TDA training, LoRA fine-tuning, and async deployment — is validated end-to-end on embedded hardware.

The broader implication is that **state-of-the-art VLA policies with asynchronous inference are viable on consumer embedded hardware today**. The training infrastructure, integration shims, and latency characterisation presented here form a reusable foundation for future VLA edge deployments beyond the Piper arm platform.

References

[1] Physical Intelligence, K. Black, N. Brown, J. Darpinian, *et al.*, "$\pi_{0.5}$: A vision-language-action model with open-world generalization," *arXiv preprint arXiv:2504.16054*, 2025.

[2] S. Han, X. Peng, *et al.*, "VLASH: Asynchronous visuomotor policy execution via temporal delay augmentation," *MIT HAN Lab*, 2025.

[3] M. J. Kim, K. Pertsch, S. Karamcheti, O. Mees, C. Finn, and S. Levine, "OpenVLA: An open-source vision-language-action model," *arXiv preprint arXiv:2406.09246*, 2024, Available: https://arxiv.org/abs/2406.09246

[4] A. Brohan *et al.*, "RT-2: Vision-language-action models transfer web knowledge to robotic control," in *Conference on robot learning (CoRL)*, 2023. Available: https://arxiv.org/abs/2307.15818

[5] O. M. Team *et al.*, "Octo: An open-source generalist robot policy," *arXiv preprint arXiv:2405.12213*, 2024, Available: https://arxiv.org/abs/2405.12213

[6] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn, "Learning fine-grained bimanual manipulation with low-cost hardware," in *Robotics: Science and systems (RSS)*, 2023. doi: 10.15607/RSS.2023.XIX.016.

[7] R. Cadene, S. Alibert, A. Soare, Q. Gallouedec, A. Zouitine, and T. Wolf, "LeRobot: Making AI for robotics more accessible." GitHub repository, 2024. Available: https://github.com/huggingface/lerobot

[8] G. Team and Google, "Gemma: Open models based on gemini research and technology," *arXiv preprint arXiv:2403.08295*, 2024.

[9] E. A. Hu, Y. Shen, P. Wallis, *et al.*, "LoRA: Low-rank adaptation of large language models," in *International conference on learning representations*, 2022.

[10] A. Ming, "Lerobot_piper: LeRobot integration for piper arm." https://github.com/WeGo-Robotics/lerobot_piper, 2024.