STAGE-WISE DETERMINISTIC INFERENCE FOR VISION-LANGUAGE-ACTION MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Batched inference improves the efficiency of vision-language-action (VLA) models in shared serving, parallel evaluation, and policy-learning rollouts. However, changes in batch context can affect tensor shapes and kernel selection, altering the order of floating-point accumulation. Action outputs can therefore differ even when request inputs and sampling noise remain fixed, leading to divergent robot trajectories and changes in task outcomes. To address this reproducibility issue, we propose stage-wise deterministic inference (Stage). Stage groups and pads requests at independently selected fixed widths for vision encoding, language encoding, and action generation, allowing each stage to balance batching efficiency and padding overhead. This organization supports autoregressive decoding, parallel regression, and iterative generation, and extends to dynamic batching with fixed stage widths and per-request noise. Experiments with four VLA models show that Stage maintains byte-identical action outputs across the tested batch contexts. Stage achieves 1.32 to $2.66\times$ LLM-42 throughput at batch size 8 and 0.99 to $1.03\times$ the throughput of the standard batched inference implementation (Native) with stage widths selected for the target batch size, while maintaining comparable task success rates in closed-loop evaluation.

1 INTRODUCTION

Vision-language-action (VLA) models map visual observations and language instructions directly to robot actions, unifying perception, task interpretation, and action generation within a single policy. Based on the action-generation architectures, VLA models divide into autoregressive decoding, parallel regression, and iterative generation. Autoregressive models such as RT-2 (Zitkovich et al., 2023) and OpenVLA (Kim et al., 2025b) encode action values as discrete tokens and predict each token conditioned on the preceding tokens. Parallel regression predicts multiple action steps in one forward pass, as in OpenVLA-OFT (Kim et al., 2025a), which combines parallel decoding with a multi-layer perceptron (MLP) to output a continuous action chunk. Iterative models generate actions from noise through diffusion or flow matching, conditioned on visual observations and language instructions. Diffusion-based models refine actions through successive denoising steps, including Octo (Ghosh et al., 2024), RDT-1B (Liu et al., 2025), CogACT (Li et al., 2024), and TinyVLA (Wen et al., 2025). Flow-matching models update the action state using a learned velocity field, as in π_0 (Black et al., 2025b) and $\pi_{0.5}$ (Black et al., 2025a).

Onboard deployment of VLA policies is subject to computational, power, and thermal constraints. To reduce onboard resource demands and separate the software environments of motion control and model inference, a policy can run on a remote server, as supported by openpi (Physical Intelligence, 2025). When several robots use the same policy they can share one model instance, which avoids replicating the model and allows concurrent requests to be processed as a batch at higher GPU utilization. Batched inference is also used in policy evaluation, where frameworks such as LeRobot (Cadene et al., 2024) batch observations from parallel simulation environments to measure task success rates. Reinforcement learning also batches policy inference to collect interaction trajectories from parallel environments, as in RL-VLA^3 (Sun et al., 2026a), which dynamically batches asynchronous observations. Request arrival and scheduling in these settings can change the batch size, batch ordering, and the set of co-executed requests.

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

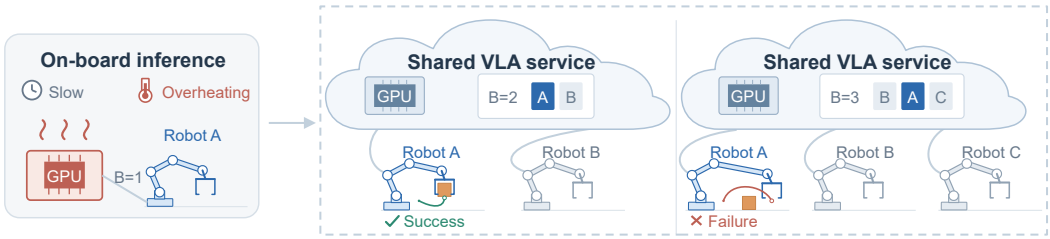


Figure 1: Illustration of VLA deployment and batch-context effects. Onboard inference can suffer from slow execution and overheating. In shared serving, changing the batch context can alter the actions and trajectories of robots, turning a successful execution into a failure.

However, batch context can change the accumulation order of floating-point reductions, which combine multiple values through operations such as summation. Floating-point addition is not associative, so different accumulation orders can produce different numerical results for the same request. These differences can affect action outputs and propagate through closed-loop execution to robot trajectories and task outcomes, as illustrated in Figure 1. We evaluate batch-context effects through paired closed-loop runs of $\pi_{0.5}$ on LIBERO-10 (Liu et al., 2023) at $B \in \{1, 2, 4, 8, 16\}$, holding the model, initial conditions, and sampling noise fixed. At every tested $B > 1$, committed actions and robot trajectories differ from the $B = 1$ reference, as shown in Table 1 and detailed in Table 4. Across these batch sizes, 5.2% to 6.6% of the reference successes become failures, while 32.0% to 40.0% of its failures become successes, indicating that batch context can alter 10–11% of episode outcomes even when aggregate success rates remain similar.

Table 1: Closed-loop outcomes of $\pi_{0.5}$ on LIBERO-10 episodes. Changes are measured relative to B=1, which succeeds in $425/500$ episodes.

Metric	B=4	B=16
Success rate	84.6%	85.4%
Successful episodes	423/500	427/500
Divergent trajectories	500/500	500/500
Success to failure	26/425	25/425
Failure to success	24/75	27/75
Outcome change rate	10.0%	10.4%

Operator-level deterministic inference uses batch-invariant kernels to constrain floating-point reductions across batch contexts. Inference frameworks such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) support this approach. We adopt this approach for the evaluated VLA models as the Invariant Kernels baseline. TBIK (Zhang et al., 2026) further aligns reductions across tensor-parallel sizes. For large language models (LLMs), determinism can also be obtained through candidate verification or execution-shape control. LLM-42 (Gond et al., 2026) retains a fast nondeterministic generation path and verifies candidate tokens at a fixed shape, rolling back when verification fails. CoRun (Zhao et al., 2026) isolates prefill per request and batches decode at fixed shapes, maintaining consistent outputs under a constant configuration.

We propose stage-wise deterministic inference (Stage), which assigns independent fixed widths V, P, and C to vision encoding, language encoding, and action generation. The widths are selected for the target workload and remain fixed throughout deployment. Each stage groups requests at its fixed width, pads incomplete groups, and discards padded outputs, as shown in Figure 3. Selecting V, P, and C independently allows a fixed configuration to balance stage-specific grouping and padding costs across batch sizes. We integrate Stage into SGLang (Zheng et al., 2024) for dynamic batching with fixed stage widths and per-request noise. Our contributions are as follows:

- **Batch-context effect.** We demonstrate that batch context can alter a VLA policy's actions, trajectories, and task outcomes. For $\pi_{0.5}$ on LIBERO-10, 10–11% of episode outcomes change despite similar aggregate success rates.
- **Stage-wise inference.** We decouple the execution widths of vision encoding, language encoding, and action generation. This organization supports VLA models based on autoregressive decoding, parallel regression, and iterative generation while preserving byte-identical action outputs across the tested batch contexts.

- **Consistent serving.** We extend Stage to dynamic batching of asynchronous requests while preserving fixed stage widths and per-request noise.
- **Evaluation.** We evaluate Stage on four VLA models across different batch contexts, with each request's action output byte-identical to its $B = 1$ reference under the same configuration. At $B = 8$, Stage achieves 1.32 to 2.66× the throughput of LLM-42. At $B = 1$ and $B = 8$, Stage achieves 0.99 to 1.03× the throughput of the standard batched inference implementation (Native), with stage widths selected for the target batch size.

2 Preliminaries and Motivation

2.1 Problem Formulation

VLA inference. VLA inference decomposes into vision encoding, language encoding, and action generation. For a request r, let I_r denote the visual observation, ℓ_r the language instruction, and s_r the robot state, and define the request input as $x_r = (I_r, \ell_r, s_r)$. Let ε_r denote the initial noise associated with request r. The three stages map x_r and ε_r to a continuous action chunk A_r according to

$$A_r = \mathcal{C}\big(\mathcal{P}(\mathcal{V}(I_r), \ell_r), s_r, \varepsilon_r\big). \tag{1}$$

Here $\mathcal{V}$, $\mathcal{P}$, and $\mathcal{C}$ denote vision encoding, language encoding, and action generation. The stages pass visual features and prefix conditions in request order, while the action stage retains each request's robot state and initial noise. The action stage takes sequential steps for autoregressive decoding, one forward pass for parallel regression, or repeated updates for iterative generation. Cross-request prefix caching is disabled because cache reuse makes the execution path dependent on prior requests.

Batch invariance. An execution context $\kappa = (B, \pi, \mathcal{S})$ specifies the batch size before padding, batch ordering, and co-executed request set. Under a fixed deployment configuration, let A_r^κ denote the action output of request r under κ. For the same x_r and ε_r, batch invariance requires any two batch contexts κ and κ' containing r to satisfy $A_r^\kappa \equiv_{\text{byte}} A_r^{\kappa'}$, where $\equiv_{\text{byte}}$ requires identical shapes, dtypes, element ordering, and serialized bytes. The notation is detailed in Table 3.

2.2 Is Task Success Affected by Batch Context?

The paired evaluation in Table 4 compares $\pi_{0.5}$ on LIBERO-10 at $B \in \{2, 4, 8, 16\}$ against a $B = 1$ reference. For each episode, the paired runs use the same model, task, initial conditions, and initial noise for corresponding action chunks. At every tested $B > 1$, committed actions and robot trajectories differ from the reference, and some episodes switch between success and failure. Although aggregate success rates remain similar, the same episode can succeed in one batch context and fail in another. These results motivate batch-invariant action outputs for reproducible episode outcomes under different batch contexts.

2.3 Is Task Success Affected by Deterministic Inference?

Deterministic execution constrains numerical computation, so task performance must be evaluated separately. We compare Native, Invariant Kernels, LLM-42, and Stage on LIBERO-10 at $B \in \{1, 8\}$. Stage and LLM-42 use the fixed configurations from Section 4.3. Stage $(1, 1, 1)$ and $(8, 4, 8)$ achieve closed-loop success rates of 85.0% and 86.6%, as shown in Table 2. In this evaluation, Stage maintains action consistency across batch contexts while retaining success rates close to Native.

Table 2: Closed-loop success rates of $\pi_{0.5}$ on LIBERO-10 episodes. Stage and LLM-42 configurations are labeled by (V, P, C) and (K, W, G). Shared entries indicate identical success rates at $B = 1$ and $B = 8$. Invariant indicates byte-identical trajectories across batch sizes.

Method	$B = 1$	$B = 8$	Invariant
Native	85.0%	86.6%	–
Invariant Kernels	84.8%		✓
LLM-42 $(1, 2, 1)$	83.4%		✓
LLM-42 $(2, 4, 2)$	86.6%		✓
Stage $(1, 1, 1)$	85.0%		✓
Stage $(8, 4, 8)$	86.6%		✓

2.4 Is Prefill Isolation Required for Determinism?

LLM prompts vary widely in length, making padding to a common length expensive for short requests. CoRun (Zhao et al., 2026) therefore isolates prefill per request, preserving each prompt's length while removing the dependence of its execution shape on other requests. During decoding, each request reuses its key-value (KV) cache and contributes only one new query token per step. CoRun batches these query tokens at a fixed padded width, while constraining attention reductions to preserve each request's reduction order across batch contexts. CoRun is designed for autoregressive decoding and does not directly apply to parallel regression or iterative action generation.

VLA language instructions are typically short and have a limited length range, making padding to a fixed length practical. This permits language encoding to process multiple requests at a fixed execution width. Together with fixed-length visual tokens, the padded text also gives each request the same prefix shape, allowing action generation to use wider request groups.

We compare language-padding strategies and stage widths on $\pi_{0.5}$ using LIBERO-10 episode first frames at $B \in \{1, 2, 4, 8\}$, as shown in Figure 2. Padded Stage $(8, 8, 8)$ executes language encoding at width $P = 8$. For all requests, its action outputs at $B \in \{2, 4, 8\}$ are byte-identical to those obtained at $B = 1$ under the same configuration, demonstrating that grouped language encoding preserves batch invariance. At $B = 8$, this configuration achieves $1.53\times$ the throughput of unpadded Stage $(8, 1, 1)$, which processes language inputs and generates actions separately for each request. This comparison measures the combined effect of text padding and wider execution in both language encoding and action generation.

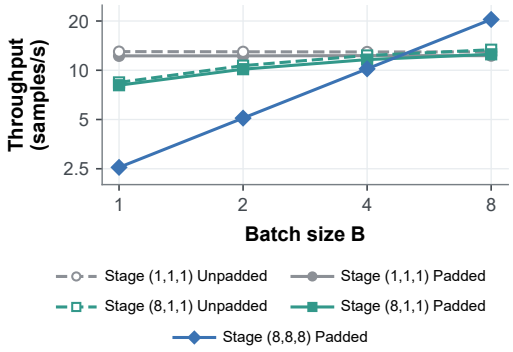


Figure 2: Forward throughput of $\pi_{0.5}$ on LIBERO-10 episode first frames under different padding strategies and stage widths.

3 Method

3.1 Stage-Wise Batch-Invariant Inference

Stage decomposition and execution. Vision encoding, language encoding, and action generation have different input structures and reuse patterns. Let V, P, and C denote their respective execution widths, each specifying the number of requests processed per call, including padded requests. These widths are positive integers selected independently for the target workload and fixed throughout deployment.

The three stages use the same grouping rule, differing only in width. At width w, they split the B requests in order into $\lceil B/w \rceil$ groups. If the final group has fewer than w requests, it is padded to width w. After execution, padded outputs are discarded and the remaining outputs are restored to request order. The call count and number of padded requests depend on batch size, while the execution width remains fixed.

Vision encoding executes $\lceil B/V \rceil$ times and returns features in the original request order. Language encoding groups requests independently of vision encoding and executes $\lceil B/P \rceil$ calls, padding shorter token sequences to a fixed token length. At $P = 1$, it can instead process each request at its original token length without sequence padding. Action generation groups prefix conditions, robot state, noise, and other required observations at width C, using the same request indices for all inputs. It takes sequential steps for autoregressive decoding, one forward pass for parallel regression, or repeated updates for iterative generation. Autoregressive KV-cache operations must also preserve request identity and independence. Successive generation steps use the same request grouping and padding pattern. After generation, padded outputs are discarded and postprocessing returns one action chunk per request.

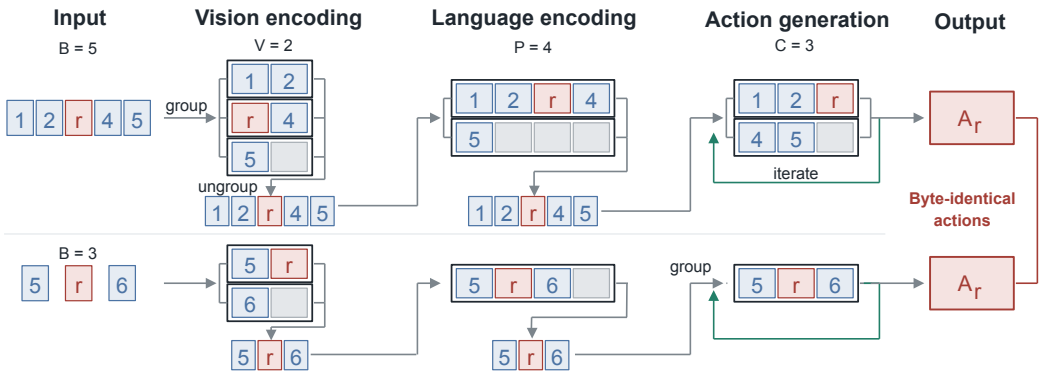


Figure 3: Stage-wise inference under different batch contexts. Both $B = 5$ and $B = 3$ use fixed stage widths $(V, P, C) = (2, 4, 3)$. Outlined groups represent stage calls, grey cells indicate padding, and highlighted cells track request r. Grouping and ungrouping preserve request identity between stages, while successive action-generation steps reuse the same grouping. Under the stated execution conditions, the same inputs and noise yield byte-identical actions for request r in both contexts.

Invariance analysis. We consider inference on one GPU without tensor or sequence parallelism, using a fixed language-padding policy and explicitly supplied per-request noise. Grouped language encoding uses a fixed padded token length, while unpadded execution at $P = 1$ retains each request's original token length across batch contexts. All other deployment settings remain fixed. Let $\mathcal{F}_{V,P,C}$ denote the Stage inference function, including grouping, padding, stage computation, output restoration, and postprocessing. Let $\mathbf{X}^{\kappa}$ denote the ordered request inputs and noise under κ, and let $i_{\kappa}(r)$ denote the output row index of request r. For any two batch contexts κ and κ' containing r with the same x_r and ε_r, the objective in Section 2.1 becomes

$$[\mathcal{F}_{V,P,C}(\mathbf{X}^{\kappa})]_{i_{\kappa}(r)} \equiv_{\text{byte}} \left[\mathcal{F}_{V,P,C}(\mathbf{X}^{\kappa'})\right]_{i_{\kappa'}(r)}. \tag{2}$$

The equality holds when fixed-shape operators execute reproducibly with the same per-request arithmetic across row positions, and computation and data movement preserve request independence. Section B.1 gives the conditions and proof.

Fixed-shape execution analysis. Let d identify the vision, language, or action stage. For stage d at width w and a batch of B requests, define the number of calls, total batch size after padding, and number of additional padding rows as

$$n_{d,w}(B) = \left\lceil \frac{B}{w} \right\rceil, \qquad \widetilde{B}_w = w \cdot n_{d,w}(B), \qquad \Delta_B(B, w) = \widetilde{B}_w - B. \tag{3}$$

Here $\widetilde{B}_w$ is the total number of request rows across all padded groups. A smaller width usually reduces $\Delta_B(B, w)$ but increases $n_{d,w}(B)$. A larger width usually reduces $n_{d,w}(B)$ and amortizes fixed call overhead, but can increase $\Delta_B(B, w)$ in a partial final group. Language encoding execution also incurs sequence padding. For request token lengths $\mathbf{L} = (L_1, \ldots, L_B)$ and a fixed token length after padding $\widetilde{L} \geq \max_i L_i$, the number of additional padding tokens for the B requests is

$$\Delta_L(\mathbf{L}, \widetilde{L}) = \sum_{i=1}^{B} (\widetilde{L} - L_i). \tag{4}$$

When $P = 1$ and each request retains its token length, language encoding computation avoids padding shorter sequences but runs once per request. Let $\tau_d(w)$ be the latency of one call to stage d at width w. The total stage latency for a batch of B requests is modeled as

$$T_d(B, w) = n_{d,w}(B) \cdot \tau_d(w). \tag{5}$$

$\Delta_B(B, w)$ counts additional request positions introduced by group padding, while $\Delta_L(\mathbf{L}, \widetilde{L})$ counts additional token positions introduced by language token padding. The stages have different call times and padding structures, which motivates selecting V, P, and C independently. Section B.2 provides the detailed cost model and width-selection analysis. Different width configurations need not generate identical actions.

3.2 BATCH-INVARIANT VLA SERVING

Batch-invariant execution. The service collects asynchronous requests within a batching window. Each dispatched batch uses the same fixed stage widths V, P, and C. Scheduling changes batch composition while preserving every stage's execution width. Under the conditions in Section 3.1, fixed inputs and noise therefore generate byte-identical actions across these batch contexts.

Schedule-invariant sampling. Each request supplies its initial noise before batching. Generating noise according to batch shape or row position could give the same request different initial states across schedules. The service keeps each request's observations and initial noise paired throughout batching, padding, and ungrouping. Repeated execution of the same request uses the same initial noise.

Let σ denote an arrival and dispatch schedule, and $\kappa_r(\sigma)$ the execution context it forms for request r. For fixed request inputs and noise within the same deployment, the stage conditions and noise handling above give $A_r^{\kappa_r(\sigma)} \equiv_{\text{byte}} A_r^{\kappa_r(\sigma')}$.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

The experiments compare Stage with Native, Invariant Kernels, and model-specific adaptations of LLM-42. Section 4.2 evaluates the effects of stage widths on forward throughput and selects fixed Stage configurations for subsequent comparisons. Section 4.3 compares forward throughput on four VLA models and evaluates action consistency across batch contexts. Section 4.4 compares the number of requests waiting for dispatch under different request arrival patterns.

All experiments run on a single RTX 4090. The $\pi_{0.5}$ implementation uses SGLang, while OpenVLA, OpenVLA-OFT, and RDT-1B use their respective PyTorch implementations. Each model is evaluated on the first observation from each of 500 episodes, using Bridge for RDT-1B and LIBERO-10 for the other models. Request inputs remain fixed across methods and batch contexts, with initial noise also fixed for iterative generators. For $\pi_{0.5}$, the language token tensor has width 200, and 10 denoising steps generate a 50-step action chunk. Model implementations and dataset details are provided in Section C.3.

4.2 STAGE-WIDTH SELECTION

The experiment evaluates all combinations of $V, P, C \in \{1, 4, 8\}$ at batch sizes $B \in \{1, 4, 8, 16\}$. For each fixed configuration, all 500 requests return byte-identical action outputs relative to its independently executed $B = 1$ reference at every tested batch size. Figure 4 reports forward throughput as the reciprocal of amortized forward latency in seconds.

The effect of execution width on throughput differs across stages. At $B = 8$, increasing only V from 1 to 8 raises throughput by 2.7% relative to Stage $(1, 1, 1)$, while increasing only P reduces it by 0.3%. Increasing only C to 8 raises throughput by 61.0%. In this comparison, widening action generation provides the largest throughput gain, while widening vision or language encoding has a limited effect.

The highest-throughput configuration depends on batch size. Stage $(1, 1, 4)$ retains 86.4% to 96.2% of the highest observed throughput at the same batch size. Among the tested configurations with $V = P = C$, $(1, 1, 1)$ achieves the highest minimum relative throughput across the same batch sizes at 58.7%. Stage $(8, 4, 8)$ is fastest at $B = 8$ and $B = 16$, but retains only 32.4% of the highest observed throughput at $B = 1$. Independent stage widths therefore allow a fixed configuration to balance inference performance across batch sizes. The subsequent method comparisons retain $(1, 1, 1)$ and $(8, 4, 8)$, which achieve the highest observed throughput at $B = 1$ and $B = 8$. Each configuration remains fixed as batch size varies.

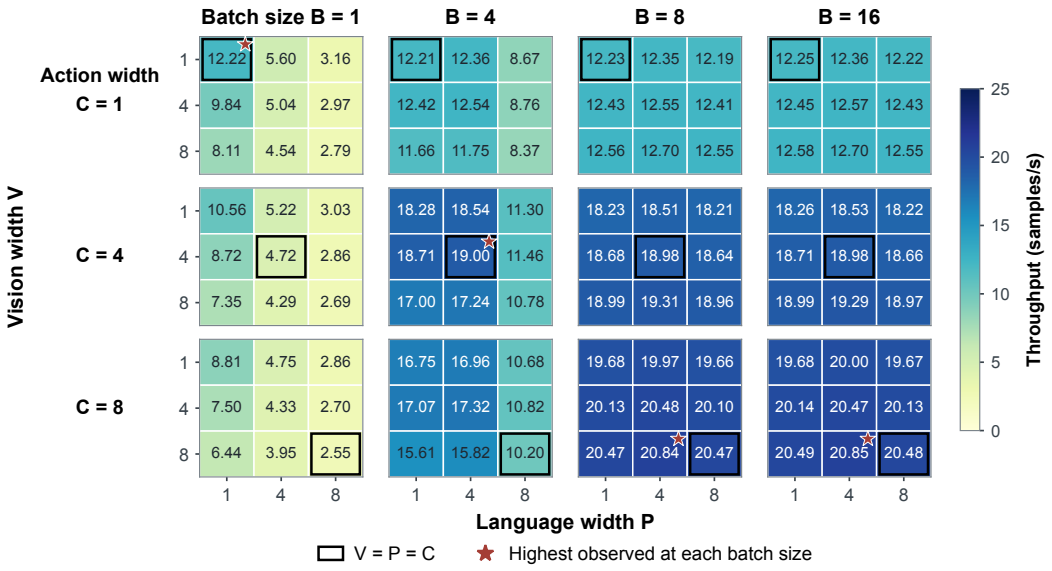


Figure 4: Forward throughput of $\pi_{0.5}$ in samples per second. Rows fix action width C and columns fix batch size B, with language width P horizontal and vision width V vertical. Black outlines mark $V = P = C$, and stars indicate the highest observed throughput at each batch size.

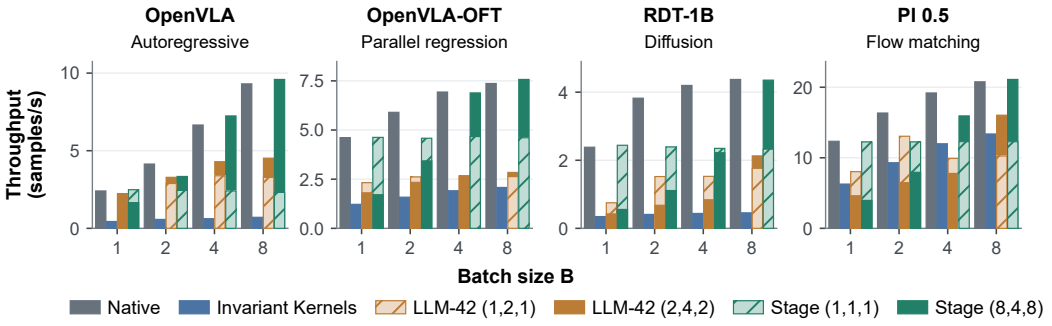


Figure 5: Forward throughput in samples per second across four VLA models. Each model uses 500 episode first frames, from Bridge for RDT-1B and LIBERO-10 otherwise. Labels give Stage (V, P, C) and LLM-42 (K, W, G), with (K, Q, G) used for OpenVLA-OFT.

4.3 Forward Inference Throughput

Figure 5 compares Native, Invariant Kernels, LLM-42, and Stage on OpenVLA, OpenVLA-OFT, RDT-1B, and $\pi_{0.5}$ at $B \in \{1, 2, 4, 8\}$. Stage uses $(V, P, C) = (1, 1, 1)$ and $(8, 4, 8)$. LLM-42 uses $(K, W, G) = (1, 2, 1)$ and $(2, 4, 2)$, with (K, Q, G) replacing (K, W, G) for the OpenVLA-OFT block-verification adaptation. LLM-42 configuration selection is described in Section C.2. Implementation details for the four models and their adaptations of Invariant Kernels and LLM-42 are provided in Section C.3.

On all four models, every Invariant Kernels, LLM-42, and Stage configuration returns byte-identical action outputs for all requests relative to its independently executed $B = 1$ reference at every tested batch size. Each comparison uses the reference for the same fixed configuration.

At $B = 1$, Stage $(1, 1, 1)$ achieves 0.99 to 1.03× Native throughput over the four models. At $B = 8$, Stage $(8, 4, 8)$ achieves approximately 1.00 to 1.03× Native throughput. The two configurations therefore retain throughput close to Native at their respective target batch sizes.

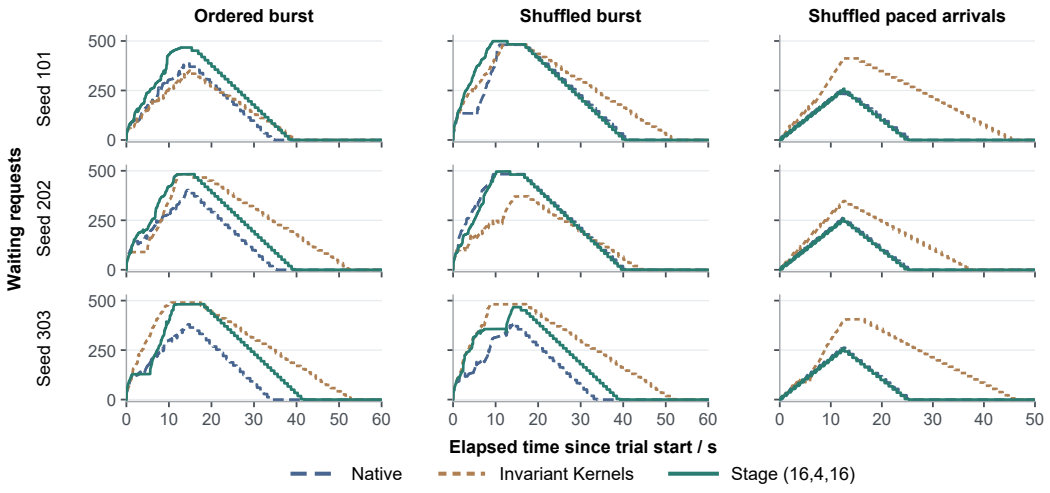


Figure 6: Requests waiting for dispatch over time. Rows correspond to scheduling seeds, and columns show ordered bursts, shuffled bursts, and shuffled paced arrivals. Shuffled paced arrivals use a planned interarrival interval of 25 ms.

Compared with the deterministic baselines at $B = 8$, Stage $(8, 4, 8)$ achieves 1.32 to $2.66\times$ the throughput of the faster evaluated LLM-42 configuration and 1.58 to $13.47\times$ that of Invariant Kernels. The largest observed improvement over LLM-42 occurs on RDT-1B at $B = 1$, where Stage $(1, 1, 1)$ achieves $3.25\times$ its throughput.

The relative performance of the methods varies with batch size. On $\pi_{0.5}$ at $B = 2$, Stage $(1, 1, 1)$ achieves $0.75\times$ Native throughput and $0.94\times$ the throughput of LLM-42 $(1, 2, 1)$. At $B = 4$, Stage $(8, 4, 8)$ achieves $0.83\times$ Native throughput and $1.61\times$ the throughput of LLM-42 $(1, 2, 1)$. These results indicate that method and configuration selection should account for the batch sizes of the target workload.

4.4 Dynamic Serving Performance

The dynamic-serving experiment compares Native, Invariant Kernels, and Stage on $\pi_{0.5}$ in SGLang, using 500 LIBERO-10 first frames as independent requests. Stage uses all configurations with $V, P \in \{1, 4, 16\}$ and $C = 16$. The server batches at most 16 requests with a waiting window of 5 ms. The experiment evaluates ordered bursts, shuffled bursts and shuffled paced arrivals under three scheduling seeds. All methods use the same planned arrivals for each scenario and seed, with request inputs and initial noise fixed. The server determines the dispatched batch composition during execution.

All Stage configurations and Invariant Kernels return byte-identical action outputs relative to their independently executed $B = 1$ references for all requests in every tested dynamic schedule. In these trials, Native matches its reference only for requests dispatched at $B = 1$.

Figure 6 compares Native, Invariant Kernels, and Stage $(16, 4, 16)$ under ordered bursts, shuffled bursts, and shuffled paced arrivals. The figure reports the number of requests initiated by the client but not yet dispatched by the server. The remaining Stage configurations are reported in Section C.4. Stage $(16, 4, 16)$ remains fixed across all arrival patterns. Completed-request throughput and P99 HTTP response latency are reported in Table 10. Under paced arrivals, Stage $(16, 4, 16)$ achieves 1.01 to $1.05\times$ Native completed-request throughput. Under ordered bursts, its throughput is 0.83 to $0.89\times$ that of Native, while its P99 HTTP response latency is 1.32 to $1.50\times$ that of Native. The fixed configuration preserves action consistency across the tested schedules, but its performance relative to Native depends on the arrival pattern.

5 Related Work

5.1 Deterministic and Batch-Invariant Inference

Prior work on deterministic inference studies the reproducibility of model outputs under different execution configurations. Request-level comparisons reveal differences in individual outputs that aggregate metrics may conceal (Fang et al., 2026). Existing approaches constrain numerical computation within operators, verify candidates before committing outputs, or fix execution shapes on the main inference path. At the level of numerical operations, raising precision mitigates rounding error, and LayerCast (Yuan et al., 2025) stores weights in low precision and computes in 32-bit floating point. Reproducible numerical computation further constrains accumulation, and Ahrens et al. (2020) use binned accumulators to make summation independent of accumulation order. TBIK (Zhang et al., 2026) builds on existing batch-invariant operators and aligns the intra-GPU and inter-GPU reduction trees, achieving bitwise consistency across the tensor-parallel configurations it supports.

For LLM inference, candidate verification with rollback separates fast generation from the consistency check. LLM-42 (Gond et al., 2026) verifies candidate tokens at a fixed execution shape and commits the prefix accepted by the verifier. When a mismatch occurs, it rolls back generation and repairs the cache. Verification can also be triggered on demand, and MarginGate (Chu et al., 2026) selects which steps to verify according to the logit margin, reducing the number of verification calls.

Execution-shape control places the constraint on the main inference path, so that different batches of requests run at a predetermined computational shape. In LLM serving, CoRun (Zhao et al., 2026) isolates prefill per request, executes decode in fixed-shape batches, and binds the random state to each request. This organization preserves identical outputs across batch contexts under a fixed deployment configuration when operator execution is reproducible, position-invariant, and request-independent.

5.2 Efficient VLA Inference and Serving

Batched VLA inference supports simulation evaluation, policy-learning rollouts, and shared robot serving. Benchmarks such as LIBERO (Liu et al., 2023) and RoboCasa (Nasiriany et al., 2024) provide simulated tasks, while evaluation frameworks batch observations from parallel environments for policy inference. During policy learning, RL-VLA^3 (Sun et al., 2026a) dynamically batches asynchronous simulation observations to improve rollout efficiency. Shared serving also aggregates requests across clients. JoyNexus (Sun et al., 2026b) supports training, rollout, and evaluation through shared model services, grouping inputs for backbone computation and routing the resulting features to the corresponding action modules. Serving systems use batching and scheduling to balance throughput and latency. Clipper (Crankshaw et al., 2017) adapts batch size to a latency target and delays execution to aggregate requests. For autoregressive inference, Orca (Yu et al., 2022) schedules requests between decoding iterations, while SGLang (Zheng et al., 2024) combines continuous batching with cache-aware scheduling. Robot policy serving also considers the execution of previously generated actions. Armory (Bansal et al., 2026) selects batches based on the remaining execution time of queued actions to reduce periods when robots have no actions to execute.

6 Conclusion

We show that batch context can alter a VLA policy's committed actions, robot trajectories, and task outcomes. To address this reproducibility issue, we propose Stage, a stage-wise deterministic inference method. Stage assigns independent fixed execution widths to vision encoding, language encoding, and action generation. It supports autoregressive decoding, parallel regression, and iterative generation, and extends to dynamic batching with fixed stage widths and per-request noise. Evaluations on four VLA models and dynamic-serving trials on $\pi_{0.5}$ yield byte-identical action outputs relative to each configuration's $B = 1$ reference in all tested batch contexts. At $B = 8$, Stage achieves 1.32 to 2.66$\times$ the throughput of LLM-42. The configurations selected for $B = 1$ and $B = 8$ retain throughput close to Native at their respective target batch sizes. Closed-loop evaluation of $\pi_{0.5}$ on LIBERO-10 further demonstrates action consistency alongside task success rates close to Native.

AI USE STATEMENT

We used generative AI for preliminary literature searches and experimental exploration. The authors organized and verified the cited references and subsequently refined the experimental designs. We also used AI tools to polish the manuscript.

REFERENCES

Willow Ahrens, James Demmel, and Hong Diep Nguyen. Algorithms for efficient reproducible floating point summation. *ACM Transactions on Mathematical Software*, 46(3):22, 2020. doi: 10.1145/3389360.

Rohan Bansal, David He, Nadun Ranawaka Arachchige, Zhenyang Chen, Soobum Kim, Kexin Rong, and Danfei Xu. Action chunk scheduling for batched robot policy serving. *arXiv*, 2026. URL https://arxiv.org/abs/2608.00337.

Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: a vision-language-action model with open-world generalization. In *Proceedings of The 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pp. 17–40. PMLR, 2025a. URL https://proceedings.mlr.press/v305/black25a.html.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, Laura Smith, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A vision-language-action flow model for general robot control. In *Proceedings of Robotics: Science and Systems*, 2025b. doi: 10.15607/RSS.2025.XXI.010.

Remi Cadene, Simon Alibert, Alexander Soare, Quentin Gallouedec, Adil Zouitine, Steven Palma, Pepijn Kooijmans, Michel Aractingi, Mustafa Shukor, Dana Aubakirova, Martino Russi, Francesco Capuano, Caroline Pascal, Jade Choghari, Khalil Meftah, Maxime Ellerbach, Jess Moss, and Thomas Wolf. LeRobot: State-of-the-art machine learning for real-world robotics in PyTorch, 2024. URL https://github.com/huggingface/lerobot.

Kexin Chu, Yang Zhou, and Wei Zhang. MarginGate: Sparse margin-triggered verification for batch-invariant LLM inference. *arXiv*, 2026. URL https://arxiv.org/abs/2605.30218.

Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation*, pp. 613–627. USENIX Association, 2017. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

Zhengyu Fang, Zhimeng Jiang, Huiyuan Chen, Xiaoge Zhang, Tianyi Li, Kaiyu Tang, Xiao Li, and Jing Li. Dataset-level metrics attenuate non-determinism: A fine-grained non-determinism evaluation in diffusion language models. *arXiv*, 2026. URL https://arxiv.org/abs/2604.13413.

Dibya Ghosh, Homer Rich Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, Jianlan Luo, You Liang Tan, Lawrence Yunliang Chen, Quan Vuong, Ted Xiao, Pannag R Sanketi, Dorsa Sadigh, Chelsea Finn, and Sergey Levine. Octo: An open-source generalist robot policy. In *Proceedings of Robotics: Science and Systems*, 2024. doi: 10.15607/RSS.2024.XX.090.

Raja Gond, Aditya K Kamath, Ramachandran Ramjee, and Ashish Panwar. LLM-42: Enabling determinism in LLM inference with verified speculation. In *ACM Symposium on Operating Systems*

Principles, 2026. URL https://www.microsoft.com/en-us/research/publication/llm-42-enabling-determinism-in-llm-inference-with-verified-speculation.

Moo Jin Kim, Chelsea Finn, and Percy Liang. Fine-tuning vision-language-action models: Optimizing speed and success. In *Proceedings of Robotics: Science and Systems*, 2025a. doi: 10.15607/RSS.2025.XXI.017.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P Foster, Pannag R Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. OpenVLA: An open-source vision-language-action model. In *Proceedings of The 8th Conference on Robot Learning*, volume 270 of *Proceedings of Machine Learning Research*, pp. 2679–2713. PMLR, 2025b. URL https://proceedings.mlr.press/v270/kim25c.html.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.

Qixiu Li, Yaobo Liang, Zeyu Wang, Lin Luo, Xi Chen, Mozheng Liao, Fangyun Wei, Yu Deng, Sicheng Xu, Yizhong Zhang, Xiaofan Wang, Bei Liu, Jianlong Fu, Jianmin Bao, Dong Chen, Yuanchun Shi, Jiaolong Yang, and Baining Guo. CogACT: A foundational vision-language-action model for synergizing cognition and action in robotic manipulation. *arXiv*, 2024. URL https://arxiv.org/abs/2411.19650.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 44776–44791. Curran Associates, Inc., 2023. doi: 10.52202/075280-1939.

Songming Liu, Lingxuan Wu, Bangguo Li, Hengkai Tan, Huayu Chen, Zhengyi Wang, Ke Xu, Hang Su, and Jun Zhu. RDT-1B: a diffusion foundation model for bimanual manipulation. In *International Conference on Learning Representations*, pp. 29982–30009, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/49f80e4d2471ad4f2edf4f5f1ab62339-Paper-Conference.pdf.

Soroush Nasiriany, Abhiram Maddukuri, Lance Zhang, Adeet Parikh, Aaron Lo, Abhishek Joshi, Ajay Mandlekar, and Yuke Zhu. RoboCasa: Large-scale simulation of household tasks for generalist robots. In *Proceedings of Robotics: Science and Systems*, 2024. doi: 10.15607/RSS.2024.XX.050.

Physical Intelligence. openpi, 2025. URL https://github.com/Physical-Intelligence/openpi.

Haoran Sun, Yongjian Guo, Zhong Guan, Shuai Di, Xiaodong Bai, Jing Long, Tianyun Zhao, Mingxi Luo, Hongke Zhao, Likang Wu, Xiaotie Deng, Xu Chu, Xi Xiao, Sheng Wen, Yicheng Gong, and Junwu Xiong. RL-VLA^3: A flexible and asynchronous reinforcement learning framework for VLA training. *arXiv*, 2026a. URL https://arxiv.org/abs/2602.05765.

Haoran Sun, Wentao Zhang, Junyang Hua, Hedan Yang, Yongjian Guo, Yifei Zhang, Xiaolong Xiang, Mingxi Luo, Jing Long, Chen Zhao, Chen Zhou, Wanting Xu, Qiming Yang, Hui Zhang, Song Wang, Xiaodong Bai, Shuai Di, Xu Chu, Xiaotie Deng, Yicheng Gong, and Junwu Xiong. JoyNexus: Service-oriented multi-tenant post-training for VLA models. *arXiv*, 2026b. URL https://arxiv.org/abs/2607.16074.

Junjie Wen, Yichen Zhu, Jinming Li, Minjie Zhu, Zhibin Tang, Kun Wu, Zhiyuan Xu, Ning Liu, Ran Cheng, Chaomin Shen, Yaxin Peng, Feifei Feng, and Jian Tang. TinyVLA: Toward fast, data-efficient vision-language-action models for robotic manipulation. *IEEE Robotics and Automation Letters*, 10(4):3988–3995, 2025. doi: 10.1109/lra.2025.3544909.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pp. 521–538. USENIX Association, 2022. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. Understanding and mitigating numerical sources of nondeterminism in LLM inference. In *Advances in Neural Information Processing Systems*, volume 38, pp. 169819–169851. Curran Associates, Inc., 2025. doi: 10.52202/085713-5653.

Ziyang Zhang, Xinheng Ding, Jiayi Yuan, Rixin Liu, Huizi Mao, Jiarong Xing, and Zirui Liu. Deterministic inference across tensor parallel sizes that eliminates training-inference mismatch. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/66224.

Shiju Zhao, Jiacheng Yang, Qihang Chen, Junhao Hu, Jiaqi Zheng, Guihai Chen, and Xusheng Chen. CoRun: Padding is simple and efficient for deterministic LLM inference. *arXiv*, 2026. URL https://arxiv.org/abs/2608.14376.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000.

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 *Proceedings of The 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pp. 2165–2183. PMLR, 2023. URL https://proceedings.mlr.press/v229/zitkovich23a.html.

A NOTATION

Table 3 summarizes the notation used in the problem formulation, method, and experiments.

Table 3: Notation used in the method and experiments.

Symbol	Meaning
r	request identifier
I_r, ℓ_r, s_r	visual observation, language instruction, robot state
x_r	request input, comprising I_r, ℓ_r, s_r
ε_r	explicitly supplied per-request initial noise
A_r	continuous action chunk generated for the request
$\mathcal{V}, \mathcal{P}, \mathcal{C}$	vision encoding, language encoding, and action generation
$B, \pi, \mathcal{S}$	batch size, batch ordering, and co-executed request set
κ	execution context, comprising $B, \pi, \mathcal{S}$
V, P, C	fixed execution widths of vision, language, and action groups in Stage
$\mathcal{F}_{V,P,C}$	Stage inference function with fixed stage widths
$\mathbf{X}^{\kappa}, i_{\kappa}(r)$	ordered request inputs and noise in context κ, and output row index of request r
d	stage label
w	generic fixed stage width
$n_{d,w}(B)$	number of calls to stage d for B requests at width w
$\widetilde{B}_w, \Delta_B(B, w)$	total batch size after padding and number of additional padding rows at width w
$L_r, \mathbf{L}$	token length of request r and token-length vector of the batch
$\widetilde{L}, \Delta_L(\mathbf{L}, \widetilde{L})$	fixed token length after padding and total number of additional padding tokens
$\tau_d(w), T_d(B, w)$	per-call latency and total latency of stage d
K, W, G	number of concurrent workers, speculative window length, and fixed verifier request width in LLM-42
Q	maximum candidate request group size in the OpenVLA-OFT adaptation of LLM-42
$\equiv_{\text{byte}}$	identical shapes, dtypes, element ordering, and serialized bytes
$\sigma, \kappa_r(\sigma)$	arrival and dispatch schedule, and execution context of request r under σ

B Proof and Cost Analysis

B.1 Batch Invariance Analysis

This appendix establishes the batch-invariance objective in Section 3.1 under fixed execution and sampling conditions. For a request r with the same input x_r and initial noise ε_r in contexts κ and κ', the required equality is

$$[\mathcal{F}_{V,P,C}(\mathbf{X}^{\kappa})]_{i_\kappa(r)} \equiv_{\text{byte}} \left[\mathcal{F}_{V,P,C}(\mathbf{X}^{\kappa'})\right]_{i_{\kappa'}(r)}. \tag{6}$$

The notation follows Table 3.

Execution conditions and stage inputs. The analysis assumes the fixed deployment settings in Section 3.1. For each stage, the execution shapes used for the same request at corresponding steps remain unchanged across batch contexts. For unpadded language encoding at $P = 1$, token lengths may differ between requests but remain fixed for each request. At these shapes, each operator must execute reproducibly and apply the same floating-point operations, including reduction order and precision, to a request regardless of its row position or companion requests.

Grouping and padding preserve each request's inputs. After execution, padded rows are removed and outputs are restored to request order without changing their bytes. Under the stated operator conditions, the vision stage produces identical features for request r in both contexts. The language stage receives the same instruction and any visual features or robot state required by the model, so its output is also identical. This argument applies whether the model uses a multimodal prefix or separate image and text encoders. Regrouping keeps these outputs associated with their requests, so action generation receives the same conditions without requiring $V = P = C$ or matching group boundaries across stages.

Action generation across architectures. Action generation uses the same conditions, robot state, and initial noise for r in both contexts. For a multistep generator, let $z_r^{(j),\kappa}$ denote its complete request state after step j, including any cache or sampler history needed by later steps. The proof starts from equal initial states and establishes the implication

$$z_r^{(j),\kappa} \equiv_{\text{byte}} z_r^{(j),\kappa'} \quad \Longrightarrow \quad z_r^{(j+1),\kappa} \equiv_{\text{byte}} z_r^{(j+1),\kappa'}. \tag{7}$$

Autoregressive decoding. Equal prefix results initialize equal token and KV states. At each step, equal states produce equal logits under the execution conditions above. Greedy decoding therefore selects the same token, and the same cache writes preserve equal KV states. Cache positions belong to the request, and completed requests do not change the execution shapes used by other requests. Induction therefore gives the same generated tokens and decoded actions.

Parallel regression. The action head receives equal conditions and performs one forward pass at width C. Applying the same argument gives the same complete action block directly. This path requires no sampling noise when the regression head is deterministic.

Diffusion. The same ε_r initializes the same noisy action, and each request starts with the same sampler history. At a fixed sampling step, equal action states and conditions produce equal model predictions. Applying the same sampler update to identical states and predictions produces identical next states, which establishes Equation 7. This covers deterministic sampling such as the evaluated RDT-1B solver. If sampling introduces additional noise, the noise at each step must be identical for the same request across batch contexts.

Flow matching. The same ε_r initializes the action state. At each integration step, equal states, conditions, and times produce equal velocity predictions. The fixed integration rule applies the same floating-point updates, establishing Equation 7. Repeating this argument over the fixed integration schedule gives equal final actions.

All four paths preserve equal outputs through the fixed per-request postprocessing, establishing Equation 6. For each fixed configuration (V, P, C), the same request produces byte-identical actions across batch contexts. Changing the widths or switching to Native may produce different actions.

B.2 Stage Width Analysis

The cost model in Section 3.1 compares the number of stage calls with the latency of each call. For stage d, one call processes a group at width w. An action-stage call includes the complete generation procedure for that group, so its latency already includes all decoding or sampling steps.

Fixed-shape execution analysis. A batch of B requests requires

$$n_{d,w}(B) = \left\lceil \frac{B}{w} \right\rceil, \qquad \widetilde{B}_w = w \cdot n_{d,w}(B), \qquad \Delta_B(B, w) = \widetilde{B}_w - B. \tag{8}$$

Let τ_d^{call} denote the fixed per-call overhead of stage d and $\tau_d^{\text{exec}}(w)$ its execution time at width w. The per-call and total stage latencies are modeled as

$$\tau_d(w) \approx \tau_d^{\text{call}} + \tau_d^{\text{exec}}(w), \qquad T_d(B, w) \approx n_{d,w}(B) \cdot \tau_d(w). \tag{9}$$

Padding increases the work performed within each call. Request padding is counted by $\Delta_B(B, w)$, while $\Delta_L(\mathbf{L}, \widetilde{L})$ counts sequence padding within the real requests. These counts describe additional input positions.

Width selection. For sequential execution with a fixed language-padding policy, the total stage-computation latency is modeled as

$$T_{\text{total}}(B; V, P, C) \approx T_{\text{vision}}(B, V) + T_{\text{language}}(B, P) + T_{\text{action}}(B, C), \tag{10}$$

where each stage contributes $T_d(B, w) \approx n_{d,w}(B) \cdot \tau_d(w)$. In this additive model, each term depends on its own stage width, so the three terms can be minimized independently over their respective candidate widths.

A larger width can reduce the number of calls but may increase the cost of each call. In our implementation based on SGLang, processing $B = 8$ requests requires two language-stage calls at $P = 4$ and one at $P = 8$. The smaller width is faster when

$$2 \cdot \tau_{\text{language}}(4) < \tau_{\text{language}}(8). \tag{11}$$

Thus, fewer calls do not necessarily imply lower latency. The widths are selected for the target workload and remain fixed during deployment.

C Additional Experimental Results

This appendix expands the closed-loop batch-context results in Section C.1. LLM-42 configuration selection and four-model numerical results appear in Section C.2 and Section C.3, followed by additional dynamic-serving configurations in Section C.4.

C.1 Complete Batch-Context Results

Table 4 expands Table 1 to all tested batch sizes. An independent $B{=}1$ repeat reproduces the committed actions, trajectories, and outcomes of all 500 episodes.

Table 4: Closed-loop outcomes of $\pi_{0.5}$ on 500 LIBERO-10 episodes at different batch sizes. Trajectory divergence and outcome changes are measured relative to the $B{=}1$ reference.

Metric	$B{=}1$	$B{=}2$	$B{=}4$	$B{=}8$	$B{=}16$
Success rate	85.0%	84.8%	84.6%	86.6%	85.4%
Successful episodes	$425/500$	$424/500$	$423/500$	$433/500$	$427/500$
Divergent trajectories	-	$500/500$	$500/500$	$500/500$	$500/500$
Success to failure	-	$28/425$	$26/425$	$22/425$	$25/425$
Failure to success	-	$27/75$	$24/75$	$30/75$	$27/75$
Outcome change rate	-	11.0%	10.0%	10.4%	10.4%

C.2 LLM-42 Configuration Selection

Table 5 reports the three LLM-42 candidates evaluated on $\pi_{0.5}$. The comparison retains $(1, 2, 1)$ and $(2, 4, 2)$, which attain the highest throughput at $B = 1$ and $B = 8$.

Table 5: LLM-42 forward throughput in samples/s on 500 LIBERO-10 episode first frames with $\pi_{0.5}$ on one RTX 4090. Labels give (K, W, G). Throughput is batch size divided by mean forward latency in seconds, including padded samples.

Configuration	$B = 1$	$B = 2$	$B = 4$	$B = 8$
LLM-42 $(1, 2, 1)$	8.03	13.05	9.92	10.26
LLM-42 $(2, 4, 2)$	4.62	6.47	7.78	16.04
LLM-42 $(4, 8, 2)$	2.80	4.05	5.01	5.73

C.3 Model Implementations and Evaluation

The $\pi_{0.5}$ experiments use SGLang, while OpenVLA and OpenVLA-OFT use their official PyTorch implementations with Transformers. RDT-1B uses its official PyTorch implementation with Transformers encoders and a Diffusers sampler. Each model is evaluated on one RTX 4090 using the first frame from each of 500 episodes. OpenVLA, OpenVLA-OFT, and $\pi_{0.5}$ share the LIBERO-10 inputs from ten tasks with 50 initial states each, while RDT-1B uses the first 500 Bridge episodes.

For Invariant Kernels, we replace matrix multiplication, normalization, and softmax with batch-invariant implementations. Matrix multiplication uses a fixed reduction order without split-K, while normalization and attention retain a reduction structure independent of batch size.

For LLM-42, we retain a fast candidate-generation path and adapt fixed-shape verification to each model. OpenVLA verifies candidate token sequences, OpenVLA-OFT verifies complete action blocks, and RDT-1B and $\pi_{0.5}$ verify iterative generation states. The window length W specifies the number of candidate tokens for autoregressive decoding or generation steps for iterative models. Each configuration fixes the verifier shape and preserves the cache or sampler state associated with each request. When verification fails, our adaptations discard inconsistent candidates and use the verifier outputs to correct the token sequence, action block, or iterative state.

Table 6, Table 7, Table 8, and Table 9 give the numerical values plotted in Figure 5. Throughput is batch size divided by mean forward latency in seconds, including padded samples.

Table 6: OpenVLA forward throughput in samples/s on 500 LIBERO-10 episode first frames on one RTX 4090. Labels give Stage (V, P, C) and LLM-42 (K, W, G).

Method	$B = 1$	$B = 2$	$B = 4$	$B = 8$
Native	2.41	4.14	6.66	9.31
Invariant Kernels	0.44	0.58	0.62	0.71
LLM-42 $(1, 2, 1)$	2.24	2.89	3.39	3.28
LLM-42 $(2, 4, 2)$	2.17	3.29	4.29	4.52
Stage $(1, 1, 1)$	2.49	2.44	2.41	2.31
Stage $(8, 4, 8)$	1.66	3.34	7.24	9.59

Table 7: OpenVLA-OFT forward throughput in samples/s on 500 LIBERO-10 episode first frames on one RTX 4090. Labels give Stage (V, P, C) and LLM-42 (K, Q, G) for block verification.

Method	$B = 1$	$B = 2$	$B = 4$	$B = 8$
Native	4.61	5.90	6.93	7.37
Invariant Kernels	1.22	1.59	1.91	2.08
LLM-42 $(1, 2, 1)$	2.32	2.62	2.67	2.63
LLM-42 $(2, 4, 2)$	1.81	2.34	2.66	2.84
Stage $(1, 1, 1)$	4.63	4.58	4.67	4.61
Stage $(8, 4, 8)$	1.71	3.43	6.89	7.57

Table 8: RDT-1B forward throughput in samples/s on the first frames of 500 Bridge episodes on one RTX 4090. Labels give Stage (V, P, C) and LLM-42 (K, W, G).

Method	$B = 1$	$B = 2$	$B = 4$	$B = 8$
Native	2.38	3.82	4.20	4.38
Invariant Kernels	0.35	0.41	0.44	0.46
LLM-42 $(1, 2, 1)$	0.75	1.52	1.53	1.76
LLM-42 $(2, 4, 2)$	0.42	0.68	0.84	2.13
Stage $(1, 1, 1)$	2.44	2.39	2.35	2.33
Stage $(8, 4, 8)$	0.55	1.11	2.22	4.35

Table 9: $\pi_{0.5}$ forward throughput in samples/s on 500 LIBERO-10 episode first frames on one RTX 4090. Labels give Stage (V, P, C) and LLM-42 (K, W, G).

Method	$B = 1$	$B = 2$	$B = 4$	$B = 8$
Native	12.34	16.35	19.19	20.77
Invariant Kernels	6.28	9.30	11.99	13.38
LLM-42 $(1, 2, 1)$	8.03	13.05	9.92	10.26
LLM-42 $(2, 4, 2)$	4.62	6.47	7.78	16.04
Stage $(1, 1, 1)$	12.25	12.25	12.29	12.28
Stage $(8, 4, 8)$	3.96	7.94	15.94	21.11

Table 10: Completed-request throughput in requests/s and P99 HTTP response latency in seconds for $\pi_{0.5}$ on LIBERO-10 first-frame requests. Values give the minimum to maximum over scheduling seeds 101, 202, and 303.

Arrival pattern	Method	Throughput	P99 latency
Ordered burst	Native	$14.04 \sim 14.25$	$20.76 \sim 21.37$
	Invariant Kernels	$9.22 \sim 11.98$	$27.25 \sim 44.22$
	Stage $(16, 4, 16)$	$11.84 \sim 12.65$	$28.25 \sim 31.15$
Shuffled burst	Native	$12.01 \sim 14.68$	$20.50 \sim 32.76$
	Invariant Kernels	$9.43 \sim 11.10$	$31.19 \sim 44.68$
	Stage $(16, 4, 16)$	$12.21 \sim 12.61$	$30.43 \sim 32.75$
Shuffled paced arrivals	Native	$18.66 \sim 19.25$	$13.49 \sim 14.21$
	Invariant Kernels	$10.62 \sim 12.75$	$26.85 \sim 34.74$
	Stage $(16, 4, 16)$	$19.48 \sim 19.52$	$13.14 \sim 13.19$

C.4 Additional Dynamic-Serving Configurations

Table 10 reports response metrics for the three methods shown in Figure 6 using the same local HTTP service. Throughput divides the completed requests by the time from trial start to the last complete response. HTTP response latency covers the interval from request initiation until the complete response is received.

The dynamic-serving experiment evaluates all nine combinations of $V, P \in \{1, 4, 16\}$ at $C = 16$, with Stage $(V, P, C) = (16, 4, 16)$ shown in Figure 6. The remaining eight configurations appear in Figure 7, Figure 8, Figure 9, Figure 10, Figure 11, Figure 12, Figure 13, and Figure 14, using the same arrival patterns, scheduling seeds, and Native and Invariant Kernels control traces.

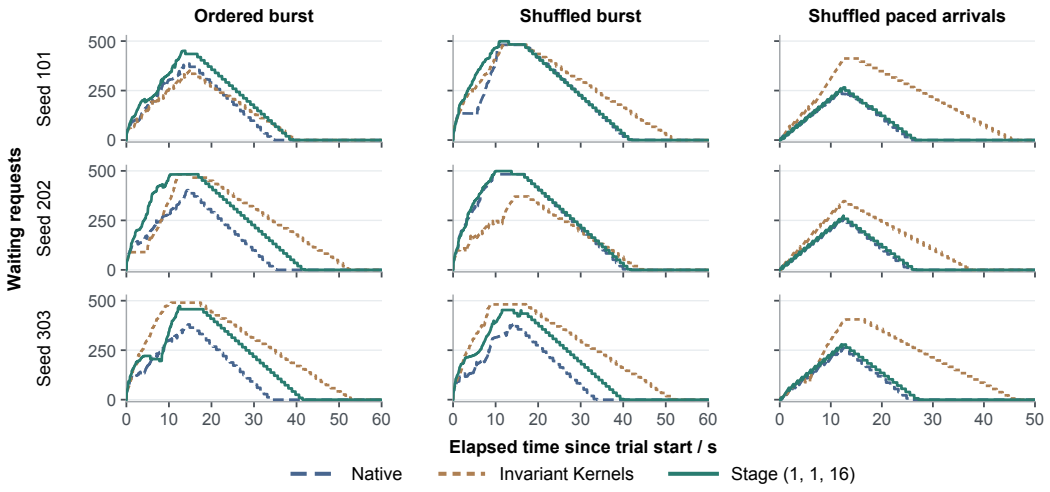


Figure 7: Waiting requests over time for Stage $(1, 1, 16)$, Native, and Invariant Kernels.

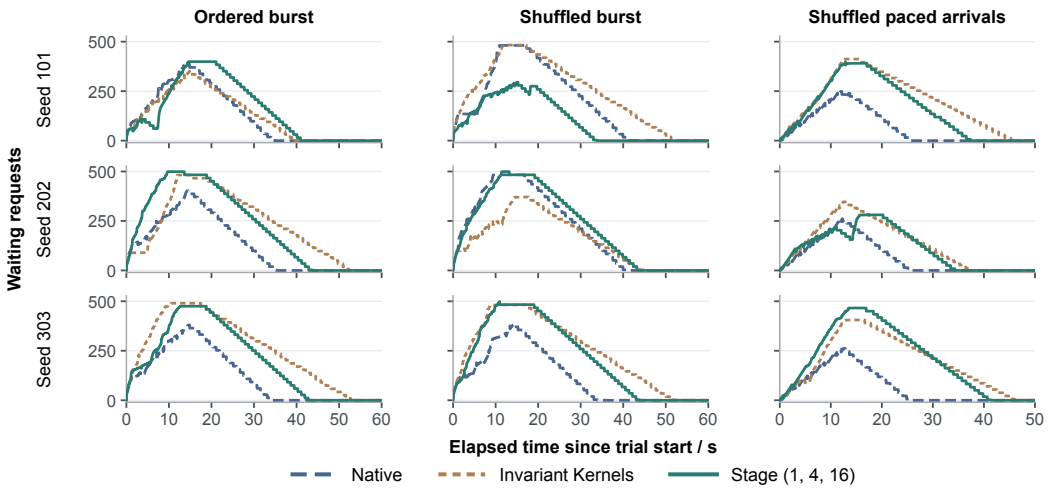


Figure 8: Waiting requests over time for Stage $(1, 4, 16)$, Native, and Invariant Kernels.

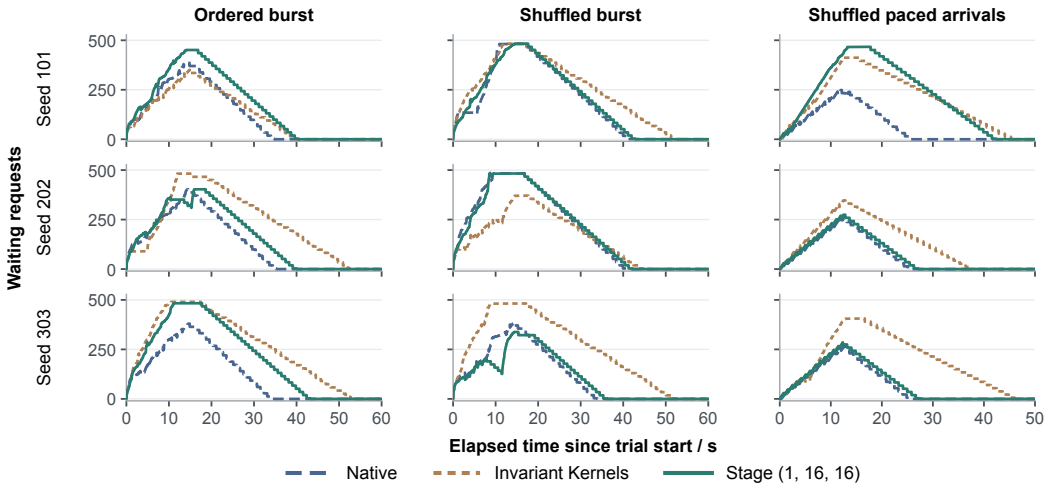


Figure 9: Waiting requests over time for Stage $(1, 16, 16)$, Native, and Invariant Kernels.

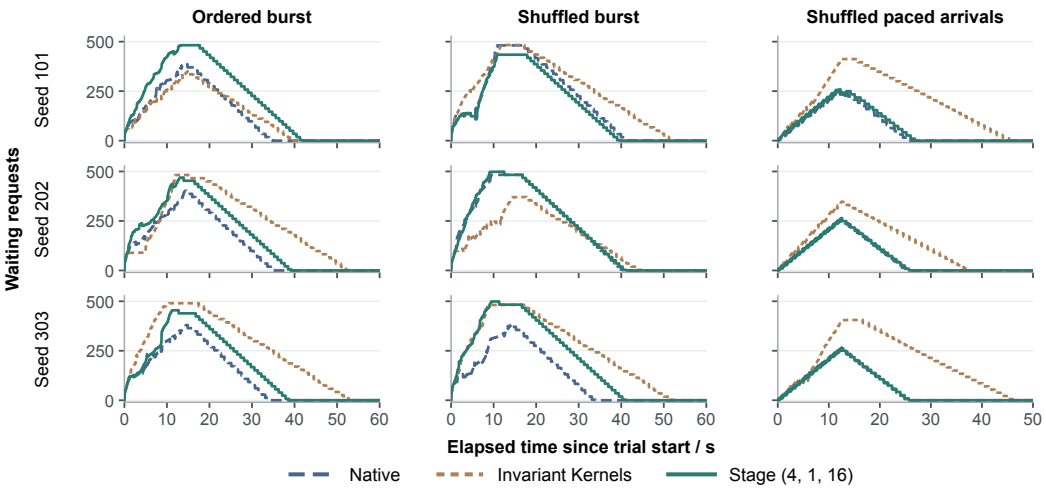


Figure 10: Waiting requests over time for Stage $(4, 1, 16)$, Native, and Invariant Kernels.

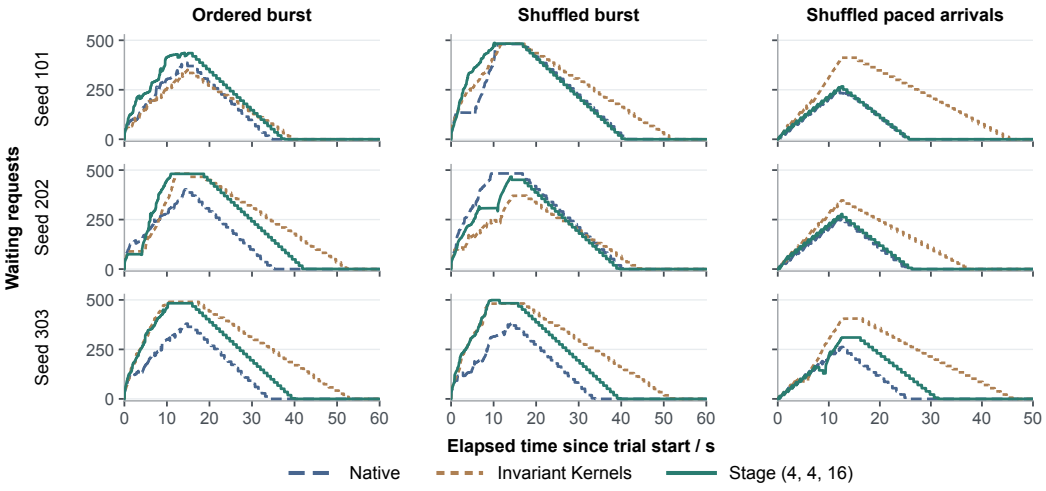


Figure 11: Waiting requests over time for Stage $(4, 4, 16)$, Native, and Invariant Kernels.

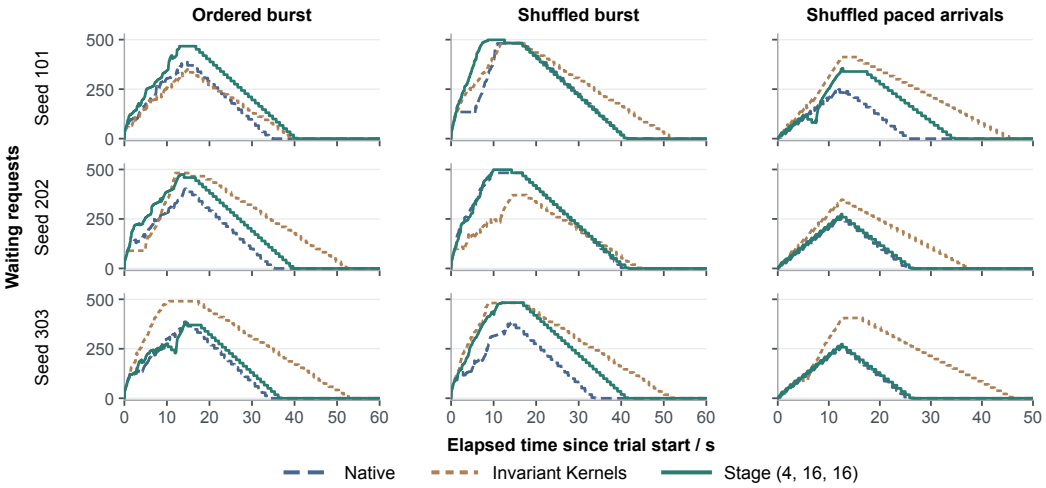


Figure 12: Waiting requests over time for Stage $(4, 16, 16)$, Native, and Invariant Kernels.

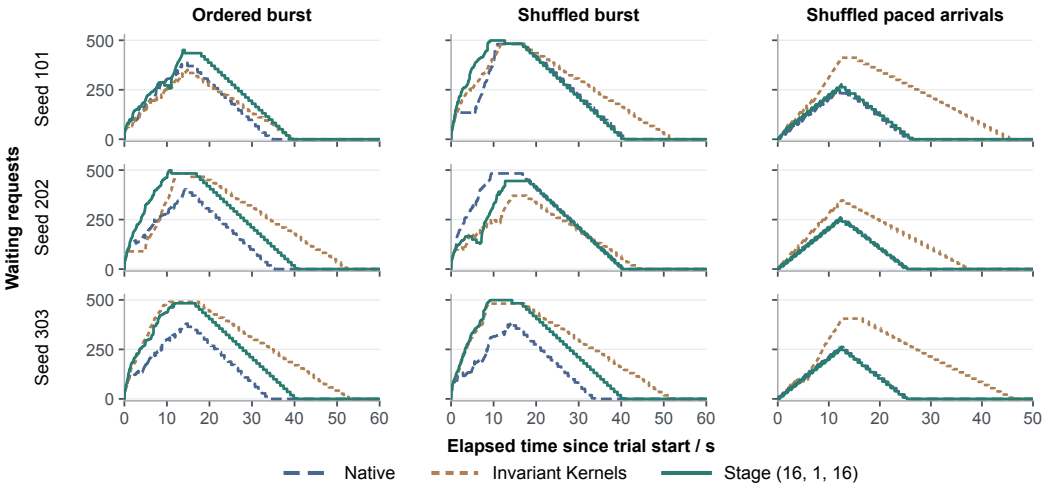


Figure 13: Waiting requests over time for Stage $(16, 1, 16)$, Native, and Invariant Kernels.

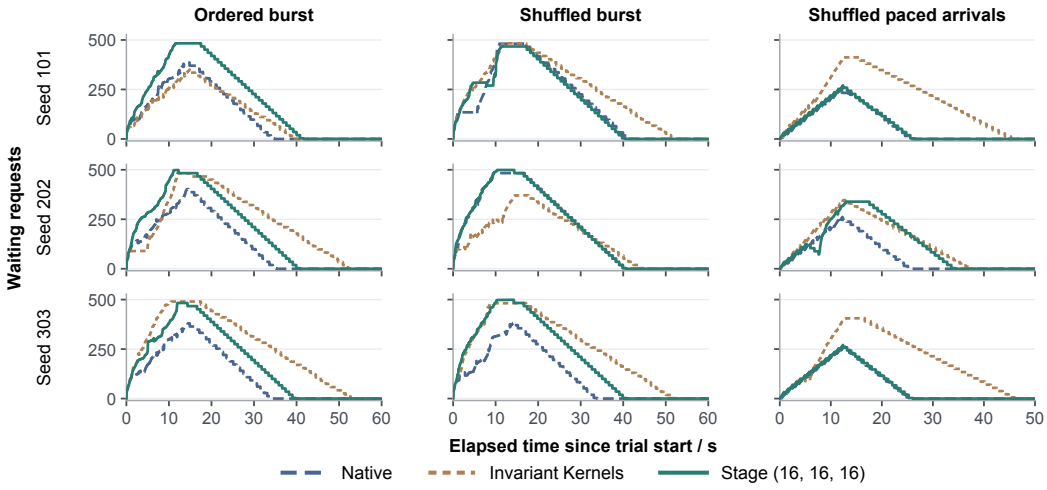


Figure 14: Waiting requests over time for Stage $(16, 16, 16)$, Native, and Invariant Kernels.