DT-VLA: LIGHTWEIGHT CONTINUAL ADAPTATION OF DYNAMIC-TEMPORAL VISION-LANGUAGE-ACTION MODELS FOR EFFICIENT DEPLOYMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Today's vision-language-action (VLA) models must keep learning new skills as they are asked to perform new tasks. To date, the dominant recipe has been to fine-tune and redeploy the whole model for each new task, which is slow, memory-heavy, and can even overwrite earlier skills. We present an algorithm that introduces a dynamic and a temporal structure on top of a lightweight design. The dynamic structure brings each new task online by updating only a small task-local component while the shared backbone stays frozen. What earlier skills rely on is never touched, so forgetting caused by later updates is prevented by construction. A switch between skills moves only this small component instead of redeploying the whole model, sharply cutting deployment downtime. The temporal structure records every adaptation as it happens. At deployment it reads that record to schedule each incoming request to the right component, or to flag a request that matches none of them as a genuinely novel task. Therefore, a growing skill set is served without a supplied task identifier. Moreover, memory stays modest even after many skills accumulate, since each dynamic component is tiny. Finally, on a continual stream of held-out manipulation tasks, our algorithm raises the average success rate from 83.1% to 86.9% over the strongest published algorithm in our comparison. It also cuts the switching downtime of a $1{,}680$-request sequence from 6.1 hours to 70.5 seconds. Code is available at `https://anonymous.4open.science/r/DT-VLA/`.

1 INTRODUCTION

Background. In recent years, progress in multimodal large language models has led to many new works on robot control that combine vision, language understanding, and continuous action. Vision-Language-Action (VLA) models have emerged as a promising route: they show strong results on a range of robotic tasks such as object manipulation, navigation, and human-robot interaction.

Most existing VLA models use a Vision-Language Model (VLM) as the backbone to jointly represent the robot's visual observations and the language instruction, and then decode the resulting multimodal features into robot control commands. One study first showed that web-scale VLM knowledge can be transferred to robot control (Zitkovich et al., 2023). Later studies extended this idea to larger datasets and to chunks of continuous action (Kim et al., 2024; Black et al., 2025), and another study showed that a single model can control several robot arms (Bjorck et al., 2025).

Limitation. Current VLA models share three limits, one on each axis that matters for a deployed robot. In memory, they are large, often several billion parameters, so one model per task does not scale. In time, adapting to a new task needs costly retraining and a full redeployment. In accuracy, they keep one shared parameter set, so learning tasks one after another overwrites what earlier tasks rely on. Existing remedies address only one aspect at a time, leaving a need to learn new tasks without retraining the whole model or making earlier skills depend on a learned router.

Motivation. As VLA models leave controlled benchmarks and enter daily use, adapting on the fly without full retraining becomes urgent. Consider guide robots for people who are blind or have low vision: at least 2.2 billion people live with a vision impairment (World Health Organization, 2026), and trained guide dogs cannot close that gap, with about 17.31 million people with a vision impair-

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.

ment in China against around 200 dogs in active service (Guangdong Disabled Persons' Federation, 2023). Quadruped guide robots fill part of this gap and work well in routine settings, but when one meets an unfamiliar place, a newly opened subway station say, current models need an offline retraining and redeployment cycle before it can serve users there. Industry hits the same wall at a larger scale, with about 542,000 industrial robots installed (International Federation of Robotics, 2025a) and almost 200,000 professional service robots sold (International Federation of Robotics, 2025b) worldwide in 2024, where every switch to a new product on an assembly line means reprogramming and, today, retraining and redeploying a large model. In both cases the bottleneck is the same: a deployed robot cannot cheaply and quickly acquire a new skill, where a model supporting fine-grained capability updates would absorb one as a small local update, about an hour on a single GPU and no redeployment, with every earlier skill left untouched.

Contribution. We propose **DT-VLA** (**D**ynamic–**T**emporal **V**ision–**L**anguage–**A**ction Model), which addresses the two costs a deployed VLA model pays for every new task: the long downtime a full redeployment takes, and the earlier skills that redeployment overwrites. Experiments show that DT-VLA has superior performance in both accuracy and deployment time. Our main contributions cover the formulation, dynamic-temporal design, and evaluation:

- We formulate deploy-time continual adaptation as three measurable metrics: the accuracy an earlier task keeps, the downtime a task switch costs, and the storage each task adds. This turns a qualitative requirement into an experimentally testable objective.
- DT-VLA introduces a dynamic and a temporal structure, so that a new task can be learned without redeploying the whole model and without forgetting the tasks already learned.
- We evaluate on eight held-out manipulation tasks under a matched budget. DT-VLA raises the average success rate from 83.1% to 86.9% over the state-of-the-art tiny-scale VLA recipe (Wang et al., 2026). It leaves earlier skills exactly unchanged. It also cuts the switching downtime of a 1,680-request sequence from 6.1 hours to 70.5 seconds.

2 Related Work

Work on VLA models falls into four lines: naive VLA models (Kim et al., 2024), parameter-efficient adaptation (Hu et al., 2022), dynamic computation for multimodal models (Chen et al., 2024), and test-time training and adaptation (Sun et al., 2020). For each line, we examine its limitations in time, accuracy, and memory during continual adaptation to new tasks.

2.1 Naive Vision-Language-Action Models

VLA models control a robot by pairing a pretrained VLM with a policy that decodes multimodal features into actions. One study first transferred web-scale VLM knowledge to robot control (Zitkovich et al., 2023), and a later study extended the recipe to a large open model (Kim et al., 2024). Other studies carry the same recipe to flow-based continuous action (Black et al., 2025), to humanoid robots that control several arms (Bjorck et al., 2025), and down to models small enough for affordable hardware (Shukor et al., 2025; Wang et al., 2026). In this dominant recipe, adapting the model to a new task means fine-tuning the whole network on that task's demonstrations.

Limitation. This full-model recipe is costly on all three axes. In time, full fine-tuning even a single new task demands a multi-GPU server and many hours (Kim et al., 2024). In memory, the backbone is large, on the order of billions of parameters (Black et al., 2025), so one fine-tuned full model per task is impractical. In accuracy, the recipe keeps one shared parameter set, so learning tasks in sequence overwrites earlier ones. We therefore keep a single frozen backbone.

2.2 Parameter-Efficient Adaptation for Vision-Language-Action Models

Parameter-efficient adaptation freezes the original weights and trains only a small set of parameters: LoRA injects a low-rank increment into frozen weight matrices (Hu et al., 2022), later work allocates that budget across weight matrices by importance (Zhang et al., 2023), and in the VLA setting it adapts a large model on small demonstration datasets at a fraction of the trained weights (Kim et al., 2025; 2024). Carried into a stream of tasks, studies add modular adapters only where a

new task needs them and route without task labels (Römer et al., 2026), grow a library of low-rank experts under a learned router with replayed routing coefficients (Lei et al., 2026), retrieve a stored skill prototype's adapter from the current state (Lee et al., 2024), compose low-rank skills by merging them in parameter space (Liu et al., 2025), or attach a prompt to each task (Song et al., 2025). Classical continual learning instead penalizes changes to the weights earlier tasks rely on (Kirkpatrick et al., 2017) or replays stored data (Rolnick et al., 2019).

Limitation. These methods lower the cost of a *single* adaptation, and the ones built for a stream pay for it elsewhere. In memory, the streaming methods add a router, a prototype memory, or stored routing coefficients on top of the adapters, and replay-based methods also keep data from earlier tasks (Rolnick et al., 2019; Zhu et al., 2026a). In accuracy, updates for different tasks interfere when they share a parameter space (Yadav et al., 2023), which one study reports on-policy reinforcement learning can offset (Hu et al., 2026), and under a router an earlier task is only as accurate as the router, which must stay correct as the stream grows. In time, retraining a shared adapter repeats the training cost, and a router inside the forward pass is paid at every action. We therefore keep one increment per task and train it alone, so no earlier skill depends on a router or on stored data.

2.3 Dynamic Computation for Multimodal Models

A separate direction lowers inference cost by making computation depend on the input: pruning or sparsifying low-importance vision tokens at runtime (Chen et al., 2024; Huang et al., 2025), exiting early once a computation budget is met (Yue et al., 2024), or routing each input through a subset of experts (Lin et al., 2026; Fedus et al., 2022) or of layers (Zhang et al., 2026).

Limitation. These methods trade accuracy for time and, more importantly, are task-agnostic. In time, they cut computation with little accuracy loss (Chen et al., 2024; Huang et al., 2025); in accuracy, the trade is explicit, with early-exit criteria weighing success against compute (Yue et al., 2024); in memory, there is no task-isolated target. These choices follow instance-level signals with no notion of *which task* is being acquired: a runtime decision, not the task-level adaptation we study.

2.4 Test-Time Training and Adaptation

Test-time training and adaptation update parameters at deploy time: an early study optimizes a self-supervised auxiliary task on each test input (Sun et al., 2020); in robotics, one reconstructs an unseen terrain and fine-tunes in simulation (Zhu et al., 2026b), another uses low-rank adapters as the trainable component for adapting the model at test time (Hu et al., 2025).

Limitation. Test-time updates are costly for interactive control on all three axes. In time, each unseen terrain needs its own on-site optimization (Zhu et al., 2026b), and an unmodified autoregressive VLA already runs below the control rates dexterous manipulation needs (Kim et al., 2025). In memory and accuracy, these methods need fresh target data per task (Hu et al., 2025), and repeatedly updating a shared base model online overwrites earlier skills. Our adaptation instead runs outside the control loop and trains each task's component in isolation.

3 Preliminary

VLA backbone. We build on the tiny-scale 0.5B VLA-Adapter architecture (Wang et al., 2026) and write the deployed policy as $\pi_\theta(a \mid o, \ell)$, where o is the observation, ℓ the language instruction, and θ the full set of backbone and policy parameters. Where the usual recipe updates θ for every new task, DT-VLA keeps it frozen; we write the frozen parameters as θ_0, the vision-language model together with the action head, with $P_0 = |\theta_0|$ their count.

Continual adaptation setting. A robot is given a stream of tasks $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_K$ that arrive one at a time. When task $\mathcal{T}_k$ arrives, the model may train on its demonstrations but cannot revisit the data of any earlier task $\mathcal{T}_{j<k}$; this is the exemplar-free constraint, and a method is judged on how well it learns each new task (*plasticity*) against how much earlier behavior it keeps (*stability*). At test time, the identity of the task being served is given; this is the *task-incremental* scenario of continual learning (van de Ven et al., 2022), and every compared method is evaluated under it, while our temporal module additionally handles requests without a supplied task identity by selecting a component from the accumulated adaptation log, which we evaluate separately.

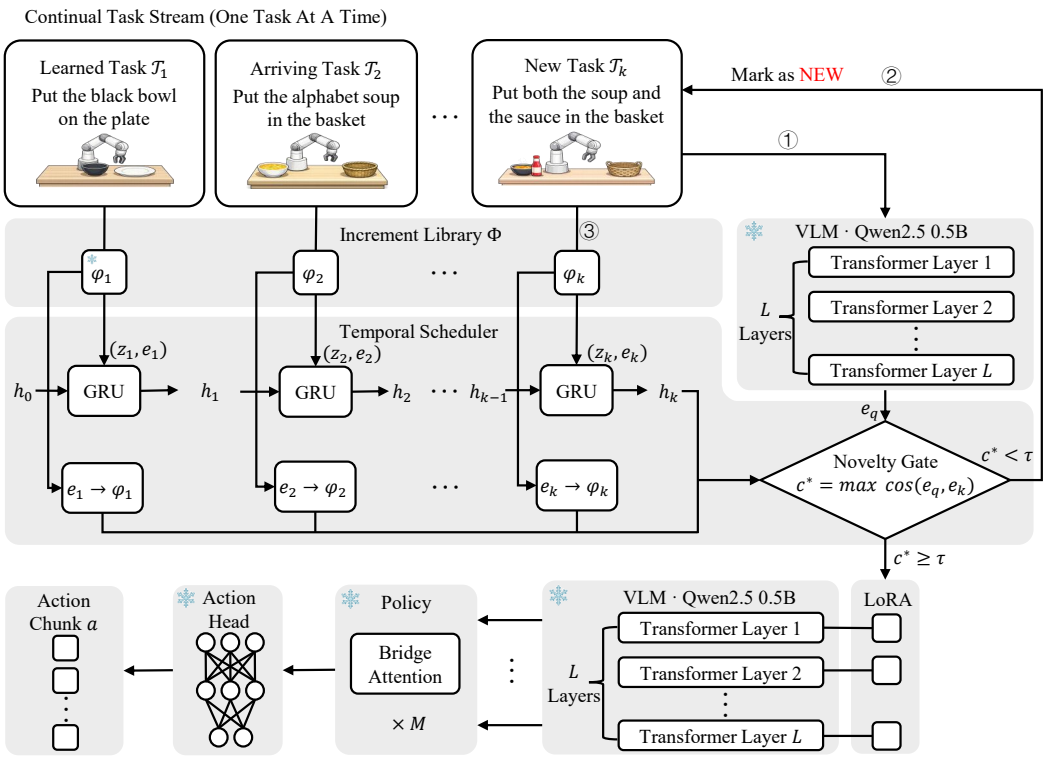


Figure 1: Overview of DT-VLA. Task cards show illustrative instructions; the backbone is drawn twice only to separate the query pass from the serving pass.

4 METHOD

4.1 OVERVIEW

DT-VLA turns continual adaptation of a VLA model into a sequence of small, isolated, deploy-time local updates over one frozen backbone. Its **Dynamic** part (D) enables task-local adaptation by training one small low-rank increment for each new task while preserving the shared backbone and all earlier increments. Its **Temporal** part (T) records acquired increments and schedules identity-free requests: it reads the adaptation log to select an increment or detect a new task, operating outside the action-generation path. The method runs in three stages, illustrated in Figure 1. **Step 1: Construction.** We train a general backbone, freeze it, and initialize an increment library, an adaptation log, and its recorder. **Step 2: Update.** Each arriving task trains a fresh increment, which joins the library together with its log entry. **Step 3: Query.** The requested task's increment is attached to the backbone for action generation. When no task identity is supplied, T consults the adaptation log to select an existing increment or flag the request for task-local adaptation.

4.2 CONSTRUCTION STAGE

General frozen backbone. Using the 0.5B VLA-Adapter architecture (Wang et al., 2026), we train our own *general* backbone jointly on held-in tasks spanning several capability types, then freeze its parameters θ_0. Training across task families provides shared representations for subsequent task-local adaptation. The backbone ablation compares mismatched, general, and matched backbones under the same task-local adaptation recipe to examine this design choice.

Increment library, log, and recorder. We initialize an empty increment library $\Phi = \{\}$, keyed by task identity, an empty log $\mathcal{H} = \{\}$ maintained by T, and a temporal recorder g_ψ with an initial hidden state h_0 to summarize the subsequent stream of increment fingerprints.

4.3 UPDATE STAGE

Per-task low-rank increment. Let $W_0 \in \mathbb{R}^{d\times d}$ be a frozen weight matrix of the backbone (an attention projection inside the VLM). For task $\mathcal{T}_k$, D learns a low-rank increment $\Delta W_k = B_k A_k$, with $B_k \in \mathbb{R}^{d\times r}$, $A_k \in \mathbb{R}^{r\times d}$ and $r \ll d$, so that the effective weight while serving $\mathcal{T}_k$ is $W_0 + \Delta W_k$ (Hu et al., 2022). Write $\varphi_k = (A_k, B_k)$ for this pair and π_k for the policy that serves $\mathcal{T}_k$, that is, the frozen backbone with φ_k attached. Only φ_k is trained, on the demonstrations of $\mathcal{T}_k$ with the behavior-cloning objective

$$\min_{\varphi_k} \mathbb{E}_{\mathcal{T}_k} \left[\mathcal{L}\big(\pi_k(o,\ell),\, a\big) \right], \tag{1}$$

the expectation being over demonstration triples (o, ℓ, a) of that task, with every other task's increment held fixed. The learned increment then joins the library Φ.

Logging the task. After task $\mathcal{T}_k$ is learned and evaluated, T appends one entry (e_k, z_k, s_k, t_k, k) to its log: the task embedding pools the backbone's last-layer features over the non-visual part of the sequence, averaged over a few demonstration frames, the fingerprint is a compact norm-based summary of φ_k, and t_k is its acquisition time. A recurrent module summarizes the stream of fingerprints,

$$h_k = g_\psi(z_k, h_{k-1}), \tag{2}$$

which records the specialization process as one trajectory. The recorder requires no separate training objective. T operates as a *bypass* module for request-level scheduling: $\mathcal{H}$ indexes the available increments, and h_k sets the novelty threshold. Once an increment is selected, action generation uses the frozen backbone and that increment without reading the log or recorder state.

Structural, exemplar-free zero-forgetting. Because each task owns its own increment and the backbone is never updated, learning $\mathcal{T}_k$ cannot perturb the parameters that serve any earlier task $\mathcal{T}_{j<k}$, which preserves earlier behavior by construction. Among the routes to stability reviewed in Section 2, this is parameter isolation: no stored data and no penalty term. Section 4.5 formalizes this guarantee using the disjoint parameters assigned to each task.

4.4 QUERY STAGE

Serving with the task identity. Given the currently active task $\mathcal{T}_k$, we read its increment φ_k from the library and attach it to the frozen backbone. The input is the current observation o and the language instruction ℓ; the output is a chunk of low-level actions, $a = \pi_k(o, \ell)$. Serving the whole stream therefore needs only the one frozen backbone plus the small library of increments.

Scheduling without the task identity. When a request arrives without a task identity, T resolves it from the log. If the request names when the task was acquired, the timestamps in $\mathcal{H}$ keep only the matching entries as candidates; otherwise all entries are. From the episode's first frame we compute a query embedding e_q, with the same features used for the index, and score it against the candidates,

$$k^* = \arg\max_k \cos(e_q, e_k), \qquad c^* = \max_k \cos(e_q, e_k). \tag{3}$$

This decision is implemented as a *novelty gate* whose threshold is read out from the recorder state, $\tau = \tau_0 + w\,\sigma(u^\top h_K)$, where h_K follows the most recent entry, so the threshold is a property of the library rather than of any candidate; σ is the logistic function and (u, τ_0, w) are calibrated once on held-in data. If $c^* \geq \tau$, the increment φ_{k^*} is attached and serving proceeds as above; if $c^* < \tau$, the gate declares a new task and re-enters the Update stage, with the nearest increment k^* serving as a fallback until the new one is ready. A request the timestamps resolve to one entry is settled directly by the log. Request resolution reads $\mathcal{H}$ and the current recorder state once, before action generation. The query pass runs on the same resident frozen backbone with no increment attached. The selected increment is then attached to this backbone to generate the requested action chunk.

4.5 ANALYSIS

We state the algorithm's core properties as formal claims. Let d be the weight dimension, r the increment rank, L the number of adapted matrices, and $P_0 = |\theta_0|$ the number of backbone parameters. We consider $r \ll d$ and $rdL \ll P_0$, so each task-local increment remains small relative to the frozen backbone and contributes only a compact set of additional parameters for its task.

Algorithm 1 DT-VLA: continual local updates over a frozen backbone

Construction: train general backbone θ_0 on held-in tasks; **freeze** θ_0
initialize library $\Phi \leftarrow \{\}$; log $\mathcal{H} \leftarrow \{\}$; recorder state h_0
for each arriving task $\mathcal{T}_k$ in the stream **do**
  Update: initialize a fresh increment φ_k, $r \ll d$
  train only φ_k on $\mathcal{T}_k$ by Eq. (1); keep θ_0 and $\{\varphi_j\}_{j \neq k}$ fixed
  $\Phi \leftarrow \Phi \cup \{\varphi_k\}$
  append (e_k, z_k, s_k, t_k, k) to $\mathcal{H}$; update recorder $h_k \leftarrow g_\psi(z_k, h_{k-1})$ // bypass, off the action path
end for
Query (identity k given): attach φ_k to θ_0; output $a = \pi_k(o, \ell)$
Query (identity not given): restrict candidates by any time reference; compute e_q from the first frame; get (k^*, c^*) by Eq. (3)
  if the log resolves the request, or $c^* \geq \tau$: serve with φ_{k^*}
  else: declare a new task, serve with φ_{k^*} as a fallback, and run **Update** for it

Lemma 1 (Per-task update size)**.** *For every task $\mathcal{T}_k$, the trained increment has size $|\varphi_k| = 2rdL = \mathcal{O}(rdL) \ll P_0$, independent of k and of the number of tasks already learned.*

Proof. Each adapted matrix contributes $rd + dr$ trainable parameters from its two low-rank factors. Summing over the L matrices gives

$$|\varphi_k| = \sum_{m=1}^{L} (rd + dr) = 2rdL. \tag{4}$$

□

Lemma 2 (Parameter disjointness)**.** *The increment φ_k is disjoint from θ_0 and from every φ_j with $j \neq k$. Optimizing Eq. (1) updates only φ_k and leaves these other parameter sets fixed.*

Theorem 1 (Structural, exemplar-free zero-forgetting)**.** *Learning any task $\mathcal{T}_k$ leaves the parameters, and hence the input-output behavior, of every earlier task $\mathcal{T}_{j<k}$ exactly unchanged, without storing exemplars from earlier tasks or using a penalty term in the update objective.*

Proof. Write superscripts $-$ and $+$ for parameters before and after learning $\mathcal{T}_k$. Lemma 2 gives

$$\theta_0^+ = \theta_0^-, \qquad \varphi_j^+ = \varphi_j^- \quad (j < k). \tag{5}$$

An earlier policy uses θ_0 and its own increment φ_j; with both unchanged, its action mapping $\pi_j(o, \ell)$ is identical before and after learning $\mathcal{T}_k$ for any fixed observation and instruction. □

Corollary 1 (Task-switch parameter writes)**.** *Switching by loading the selected increment over the resident frozen backbone writes $\mathcal{O}(rdL)$ parameter values, compared with $\mathcal{O}(P_0)$ for a full-model reload, a reduction by a factor of order $P_0/(rdL)$ in the number of parameters written.*

Proof. A switch loads only $|\varphi_k| = 2rdL$ values by Lemma 1, while a full-model reload writes P_0 parameters. The shared backbone is retained throughout the switch. □

Proposition 1 (Model-parameter storage)**.** *After K tasks, the total number of stored backbone and increment parameters is*

$$M(K) = P_0 + \sum_{k=1}^{K} |\varphi_k| = P_0 + 2KrdL. \tag{6}$$

Each task adds only $2rdL$ parameters to the stored model, whereas keeping a separate full model for every task duplicates the backbone and requires KP_0 parameters.

Proof. The backbone is stored once, and each task contributes one increment of size $2rdL$. Summing these disjoint increments yields the expression above. □

Appendix A.1 provides the proof of Lemma 2 and the low-rank approximation analysis; Appendix A.4 evaluates the resulting accuracy-storage trade-off.

5 EXPERIMENTAL STUDY

5.1 EXPERIMENTAL SETUP

All experiments run on one workstation with a single NVIDIA GeForce RTX 5090 GPU. Our training recipe follows VLA-Adapter (Wang et al., 2026) and our evaluation adopts its success-rate metric; for the remaining implementation details see Appendix A.2. Each increment is trained for $15{,}000$ steps on its own task only, at rank $r = 16$: about 1.08M parameters and 2.2 MB per task. This default balances accuracy and per-task storage in the rank study (Appendix A.4).

We evaluate a custom eight-task continual stream spanning all four LIBERO suites, modeling arrivals from different capability types. All compared methods use the same stream. The official LIBERO lifelong protocol orders ten tasks within a single suite (Liu et al., 2023).

5.1.1 DATASET

We use the LIBERO dataset (Liu et al., 2023) in the processed form of a prior study (Kim et al., 2024): 40 manipulation tasks in four suites of ten, Spatial, Object, Goal, and Long, one natural-language instruction each, and $1{,}693$ demonstrations (Appendix A.2). We hold out two tasks per suite, so 32 tasks train the general backbone and the remaining 8 form the continual stream, never seen during backbone training; they arrive in a fixed order and we call them $\mathcal{T}_1$–$\mathcal{T}_8$ throughout.

5.1.2 ALGORITHMS

We compare five algorithms: DT-VLA and four baselines. (1) **VLA-Adapter** (Wang et al., 2026), the state-of-the-art tiny-scale VLA recipe and the strongest published algorithm here, which adapts to each task by fully fine-tuning one complete model; in our setup that is 2.95 GB per task; in every figure and table, VLA-Adapter denotes this recipe. (2) **DMPEL** (Lei et al., 2026), a library of low-rank experts under a learned router with replayed routing coefficients. (3) **IsCiL** (Lee et al., 2024), which retrieves a stored skill prototype's adapter from the current state. (4) **LoRA** (Hu et al., 2022), our frozen backbone with one *shared* low-rank increment trained sequentially across the whole stream. All five see the same demonstrations and the same per-task budget of 120k samples (Appendix A.2). VLA-Adapter and LoRA share our backbone; DMPEL and IsCiL use their official backbones and pretrained checkpoints for system-level comparison.

5.1.3 FACTORS

We vary two factors. (1) The **number of subsequently learned tasks** (0 to 7): how many tasks the model has learned after a given task was acquired, which probes whether later learning changes the behavior of an earlier task. (2) The **number of task-switch requests** (M from 1 to $1{,}680$): M is the length of the served request sequence, with $M = 1$ measuring a single request and $M = 240$ and $M = 1{,}680$ measuring cumulative downtime over longer request sequences.

5.1.4 MEASUREMENTS

We measure three kinds of outcomes. (1) **Accuracy**: the success rate (%) of a task over 20 evaluation episodes, read at two moments, the *success rate right after learning* and the *success rate at the end of the stream*, re-measured once all eight tasks have been learned; the average of the latter over the eight tasks is a method's final accuracy. (2) **Time**: the total switching downtime, the sum of every task-switch stall accumulated over a served request sequence, measured end to end. (3) **Memory**: the persistent storage a method adds per task in order to serve it, in megabytes; what that artifact is for each method is listed in Appendix A.2. The accuracy comparison uses supplied task identities; Section 5.2.3 evaluates identity-free routing, and Appendix A.7 evaluates end-to-end serving, including novelty detection and task adaptation.

5.2 EXPERIMENTAL RESULTS

Figure 2 summarizes task retention as later tasks are learned and switching downtime as requests accumulate; Appendix A.3 presents additional scaling results.

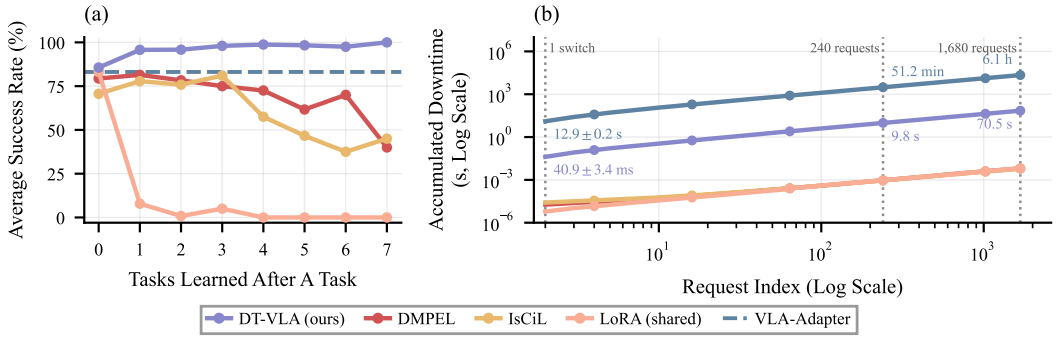


Figure 2: Main results. (a) Accuracy against tasks learned afterwards. (b) Switching downtime accumulated over a request sequence, with dotted lines at 1, 240, and 1,680 requests.

5.2.1 Effect of the Number of Subsequently Learned Tasks

DT-VLA maintains its average success rate across the stream: 86.25 immediately after learning and 86.9 at the end (Figure 2(a)). The parameters serving each task remain bit-exact unchanged; the observed difference reflects evaluation stochasticity. The router-based baseline drops from 79.4 to 77.5 and the retrieval-based baseline from 70.6 to 66.9: both keep shared machinery that must stay correct as the stream grows. The shared LoRA baseline updates one increment for every task: it falls from 83.1 to 5.0, and the seven tasks that later learning overwrites lose 89.3 points on average. The strongest baseline trains one full model per task, so later learning cannot change earlier tasks; it holds 83.1 at both readings, using 2.95 GB of storage per task.

Task-local adaptation enables the frozen backbone to solve the held-out tasks, improving its zero-shot success rate of 0.0 on all eight tasks to an average of 86.9. Appendix A.4 examines the effect of increment capacity on performance across these held-out tasks.

5.2.2 Effect of the Deployment Scale

We measure cumulative switching downtime over fixed request sequences executed end to end for DT-VLA and the full-model baseline (Figure 2(b)). Over 240 requests the full-model baseline accumulates 51.2 minutes of downtime against 9.8 seconds for DT-VLA, about 315× less; over 1,680 requests the gap is 6.1 hours against 70.5 seconds (311×). Per request, reloading a full model takes 12.9 ± 0.2 seconds while swapping one increment over the resident backbone takes 40.9 ± 3.4 milliseconds. Both methods have similar initial startup times (13.6 s against 13.8 s) and identical inference speed. Both methods are timed with warm caches on the same workstation, using the same deployment code path for model loading and task switching.

Memory scales the same way: N tasks cost $N \times 2.95$ GB for the full-model recipe against one resident backbone plus $N \times 2.2$ MB for ours. The other three baselines, measured on the same sequence, serve from a resident state and move no parameters at a switch, so their per-switch stall is bookkeeping only, about 0.004 milliseconds. Their retained accuracy is reported in Section 5.2.1. Small increments also reduce the amount of data transferred when distributing trained skills to individual robots or an entire robot fleet, as illustrated in Section 5.2.4.

5.2.3 Ablation Study

Ablations show that each DT-VLA component is needed (Figure 3; details in Appendix A.5).

The general backbone. A controlled diagnostic varies only the frozen backbone, with a mismatched, a general, and a matched arm (Figure 3(b)). The matched arm confirms that a local increment reweights the features a backbone already exposes. What the general backbone buys is deployability: no prior knowledge of an incoming task's family is needed, and tasks outside the specialized families can still be adapted using the same resident backbone.

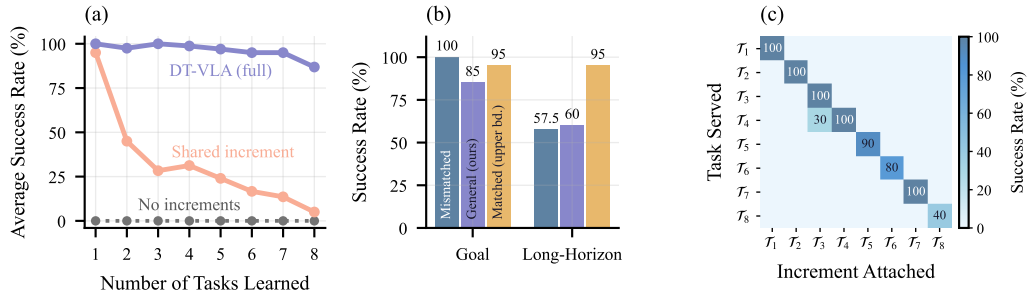


Figure 3: Ablations. (a) Accuracy along the stream: full method, one shared increment, none. (b) The same adaptation over three frozen backbones; goal task and the two long-horizon tasks averaged, the matched arm an upper bound. (c) Each task (row) served by each increment (column).

Per-task increments. Figure 3(a) compares independent per-task increments, one shared increment, and the frozen backbone alone, isolating the effects of task separation and adaptation. Without the increments the frozen backbone scores 0.0 zero-shot on every held-out task. Removing only their *isolation* gives the shared LoRA baseline, which collapses to 5.0 at the end of the stream. Cross-task attachment further shows that each increment carries task-specific behavior: a task reaches 88.8 with its own increment and 0.5 with another's (Figure 3(c)).

Increment coverage. Spreading a lower rank across all linear layers, instead of concentrating $r = 16$ on the attention projections, reaches 81.9 at the end of the stream with zero drop but at 8.9 MB per task. With the rank study, neither wider coverage nor more depth wins consistently, so we keep the concentrated default at the smallest per-task artifact.

The temporal module. For identity-free requests, T correctly routes all 40 in-library queries and achieves a novelty-detection AUROC of 0.9992. The detection threshold is calibrated on held-in data; Appendix A.5 details the evaluation protocol.

5.2.4 Case Studies

We illustrate two deployment scenarios using the measured switching times and artifact sizes. Transmission times are estimated at the stated link bandwidths (Appendix A.6).

A guide robot meets a new place. A subway station opens. Distributing a trained skill requires a 2.95 GB model under the full-model recipe, about 40 minutes over a 10 Mbps link, versus a 2.2 MB increment in about 2 seconds for DT-VLA. We also evaluated the adaptation loop end to end on LIBERO without supplied task identities: the gate detected a withheld task, one adaptation followed, and the task achieved 100% success while the stored policies remained unchanged (Appendix A.7).

An industrial line switches products. The 240-request workload of Section 5.2.2 models repeated product changes on an industrial line, where switching stalls would interrupt production. The gap grows with fleet scale: pushing ten skills to fifty robots over a shared 100 Mbps uplink takes about 34 hours for full models against 1.5 minutes for increments. Compact increments also reduce skill-library storage and increase the number of tasks an edge device can hold (Appendix A.6).

6 Conclusion

We presented DT-VLA, which adapts a VLA model to a stream of new tasks by training one small low-rank increment per task over one frozen general backbone, while a bypass module logs each increment and routes requests, so deployment needs no oracle task identity. On eight held-out LIBERO tasks it raises final accuracy from the 83.1 of the strongest published algorithm in our comparison to 86.9, leaves earlier tasks exactly unchanged, and cuts the switching downtime of a $1{,}680$-request sequence from 6.1 hours to 70.5 seconds. Future work will harden the scheduler's novelty boundary and scale the backbone for long-horizon accuracy.

REPRODUCIBILITY STATEMENT

We describe the DT-VLA architecture and its construction, update, and query procedures in Section 4 and Algorithm 1. The assumptions and proofs underlying the theoretical results are provided in Section 4.5 and Appendix A.1. Section 5.1 and Appendix A.2 document the dataset, task split and ordering, training budgets, implementation details, hardware, and evaluation metrics. Appendix A.5 provides additional details on task routing and novelty detection, including threshold calibration. Appendices A.7 and A.8 describe the end-to-end evaluation without task identities and the adaptation-time measurements. An anonymous implementation repository is linked in the abstract.

ETHICS STATEMENT

This work investigates continual adaptation for robotic manipulation using the LIBERO benchmark. The guide-robot and industrial scenarios illustrate potential applications and projected deployment benefits; they do not establish the safety or effectiveness of deployment in these settings. Parameter isolation preserves previously learned policies but does not guarantee safe behavior under unfamiliar inputs or incorrect task routing. In particular, using an unsuitable stored policy as a fallback for a novel task could produce unsafe actions on physical robots. Practical deployment would therefore require application-specific safety evaluation, validation of newly adapted skills, human oversight, and mechanisms for safe stopping or intervention. Assistive applications should additionally be evaluated in consultation with intended users and relevant domain experts.

AI USE STATEMENT

In this work, we use Qwen2.5-0.5B as the language-model component of DT-VLA's vision-language backbone. The resulting multimodal backbone processes visual observations and natural-language task instructions to provide representations for robotic action prediction. This use is an integral part of the proposed architecture, as described in the methodology section. The authors take full responsibility for the research methodology, experimental results, and the final content of this paper.

REFERENCES

Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, et al. GR00T N1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael 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*, Los Angeles, CA, USA, June 2025. doi: 10.15607/RSS.2025.XXI.010.

Liang Chen, Haozhe Zhao, Tianyu Liu, Shuai Bai, Junyang Lin, Chang Zhou, and Baobao Chang. An image is worth 1/2 tokens after layer 2: Plug-and-play inference acceleration for large vision-language models. In *European Conference on Computer Vision*, pp. 19–35. Springer, 2024.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Guangdong Disabled Persons' Federation. International Guide Dog Day: One guide dog for every 90,000 visually impaired people in China. Guangdong Disabled Persons' Federation official website (media coverage), 2023. In Chinese.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Jiaheng Hu, Jay Shim, Chen Tang, Yoonchang Sung, Bo Liu, Peter Stone, and Roberto Martín-Martín. Simple recipe works: Vision-language-action models are natural continual learners with reinforcement learning. *Reinforcement Learning Journal*, 7, 2026.

Jinwu Hu, Zitian Zhang, Guohao Chen, Xutao Wen, Chao Shuai, Wei Luo, Bin Xiao, Yuanqing Li, and Mingkui Tan. Test-time learning for large language models. In *International Conference on Machine Learning*, pp. 24823–24849. PMLR, 2025.

Wenxuan Huang, Zijie Zhai, Yunhang Shen, Shaosheng Cao, Fei Zhao, Xiangfeng Xu, Zheyu Ye, and Shaohui Lin. Dynamic-LLaVA: Efficient multimodal large language models via dynamic vision-language context sparsification. In *International Conference on Learning Representations*, 2025.

International Federation of Robotics. World robotics 2025 – industrial robots. Technical report, International Federation of Robotics (IFR), 2025a.

International Federation of Robotics. World robotics 2025 – service robots. Technical report, International Federation of Robotics (IFR), 2025b.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. OpenVLA: An open-source vision-language-action model. In *Conference on Robot Learning*, pp. 2679–2713. PMLR, 2024.

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*, 2025.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017. doi: 10.1073/pnas.1611835114.

Daehee Lee, Minjong Yoo, Woo Kyung Kim, Wonje Choi, and Honguk Woo. Incremental learning of retrievable skills for efficient continual task adaptation. In *Advances in Neural Information Processing Systems*, 2024.

Yuheng Lei, Sitong Mao, Shunbo Zhou, Hongyuan Zhang, Xuelong Li, and Ping Luo. Dynamic mixture of progressive parameter-efficient expert library for lifelong robot learning. *Transactions on Machine Learning Research*, 2026. ISSN 2835-8856.

Bin Lin, Zhenyu Tang, Yang Ye, Jinfa Huang, Junwu Zhang, Yatian Pang, Peng Jin, Munan Ning, Jiebo Luo, and Li Yuan. MoE-LLaVA: Mixture of experts for large vision-language models. *IEEE Transactions on Multimedia*, 28:4408–4419, 2026. doi: 10.1109/TMM.2026.3654458.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 44776–44791. Curran Associates, Inc., 2023.

Tenglong Liu, Jianxiong Li, Yinan Zheng, Haoyi Niu, Yixing Lan, Xin Xu, and Xianyuan Zhan. Skill expansion and composition in parameter space. In *International Conference on Learning Representations*, 2025.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Ralf Römer, Yi Zhang, Yuming Li, and Angela P. Schoellig. CLARE: Continual learning for vision-language-action models via autonomous adapter routing and expansion. *IEEE Robotics and Automation Letters*, 2026. doi: 10.1109/LRA.2026.3693992.

Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, et al. SmolVLA: A vision-language-action model for affordable and efficient robotics. *arXiv preprint arXiv:2506.01844*, 2025.

Mingchen Song, Xiang Deng, Guoqiang Zhong, Qi Lv, Jia Wan, Yinchuan Li, Jianye Hao, and Weili Guan. Few-shot vision-language action-incremental policy learning. *arXiv preprint arXiv:2504.15517*, 2025.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International Conference on Machine Learning*, pp. 9229–9248. PMLR, 2020.

Gido M. van de Ven, Tinne Tuytelaars, and Andreas S. Tolias. Three types of incremental learning. *Nature Machine Intelligence*, 4:1185–1197, 2022. doi: 10.1038/s42256-022-00568-3.

Yihao Wang, Pengxiang Ding, Lingxiao Li, Can Cui, Zirui Ge, Xinyang Tong, Wenxuan Song, Han Zhao, Wei Zhao, Pengxu Hou, et al. VLA-Adapter: An effective paradigm for tiny-scale vision-language-action model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 18638–18646, 2026.

World Health Organization. Blindness and vision impairment. WHO Fact Sheet, 2026.

Prateek Yadav, Derek Tam, Leshem Choshen, Colin Raffel, and Mohit Bansal. TIES-merging: Resolving interference when merging models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 7093–7115. Curran Associates, Inc., 2023. doi: 10.52202/075280-0310.

Yang Yue, Yulin Wang, Bingyi Kang, Yizeng Han, Shenzhi Wang, Shiji Song, Jiashi Feng, and Gao Huang. DeeR-VLA: Dynamic inference of multimodal large language models for efficient robot execution. In *Advances in Neural Information Processing Systems*, volume 37, pp. 56619–56643, 2024.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning. In *International Conference on Learning Representations*, 2023.

Rongyu Zhang, Menghang Dong, Yuan Zhang, Liang Heng, Xiaowei Chi, Gaole Dai, Li Du, Yuan Du, and Shanghang Zhang. MoLe-VLA: Dynamic layer-skipping vision language action model via mixture-of-layers for efficient robot manipulation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 18764–18772, 2026.

Jiarun Zhu, Yijun Hong, Xiaoquan Sun, Zetian Xu, Qijun He, Haijier Chen, Zhiyong Wang, Mingqi Yuan, Wenjun Zeng, and Jiayu Chen. Can vision-language-action models learn from real-world data continually without forgetting? *arXiv preprint arXiv:2605.26820*, 2026a.

Shaoting Zhu, Baijun Ye, Jiaxuan Wang, Jiakang Chen, Ziwen Zhuang, Linzhan Mou, Runhan Huang, and Hang Zhao. TTT-Parkour: Rapid test-time training for perceptive robot parkour. In *Conference on Robot Learning (CoRL)*, 2026b. Accepted, to appear.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, et al. RT-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pp. 2165–2183. PMLR, 2023.

A Appendix

A.1 Additional Proofs and Auxiliary Analysis

We prove Lemma 2 and analyze low-rank approximation and scheduling. Section 4.5 gives the notation and proves the update-size, zero-forgetting, switching, and model-storage results.

Proof of Lemma 2. Each task initializes a fresh increment φ_k, so the parameter sets are pairwise disjoint and disjoint from θ_0 by construction. The Update stage optimizes Eq. (1) over φ_k alone. The backbone θ_0 and all other increments remain frozen throughout this optimization. □

Lemma 3 (Best low-rank approximation error)**.** *Let ΔW_k^* be the ideal task-specific weight change and let $\sigma_1 \geq \sigma_2 \geq \cdots$ be its singular values. The best rank-r increment satisfies*

$$\min_{\mathrm{rank}(\Delta W_k) \leq r} \left\| \Delta W_k^* - \Delta W_k \right\|_F = \Big(\sum_{i>r} \sigma_i^2 \Big)^{1/2}. \tag{7}$$

Proof sketch. This is the Eckart–Young theorem: truncating the singular value decomposition of ΔW_k^* at rank r attains the bound, and no rank-r matrix improves on it in the Frobenius norm. □

By Lemma 3, the best rank-r approximation error is determined by the singular-value tail beyond rank r and decreases monotonically as r grows. When the required task adaptation lies mostly in a low-rank subspace, this tail is small and a modest r suffices for the matrix approximation. Appendix A.4 evaluates the effect of increment rank on task performance.

Remark 1 (Scheduling cost). Resolving one identity-free request requires one query-embedding computation followed by K cosine scores against d_e-dimensional entries. The scoring step takes $\mathcal{O}(Kd_e)$ time. The embedding index stores one vector per task and requires $\mathcal{O}(Kd_e)$ memory.

A.2 Implementation Details

All experiments use Python and PyTorch on a workstation with one NVIDIA GeForce RTX 5090 GPU (32 GB), a 16-core CPU, and 96 GB of RAM, under Ubuntu 22.04. The task stream instantiates a lifelong-learning setting over the LIBERO tasks (Liu et al., 2023), and every accuracy reading is a success rate over 20 evaluation episodes. The raw LIBERO release provides 50 demonstrations per task; the processed form filters out unsuccessful demonstrations and no-op segments, leaving the 1,693 demonstrations we train on, 273,465 image frames and about 10 GB of data in total.

Table 1 lists the eight held-out tasks of Section 5.1.1 in stream order; the remaining 32 tasks form the held-in set used for training the general backbone.

Table 1: Eight held-out tasks in stream order; parentheses show indices within each suite.

	Suite (index)	Instruction
$\mathcal{T}_1$	Spatial (5)	pick up the black bowl on the ramekin and place it on the plate
$\mathcal{T}_2$	Spatial (8)	pick up the black bowl next to the plate and place it on the plate
$\mathcal{T}_3$	Object (3)	pick up the bbq sauce and place it in the basket
$\mathcal{T}_4$	Object (9)	pick up the orange juice and place it in the basket
$\mathcal{T}_5$	Goal (4)	put the bowl on top of the cabinet
$\mathcal{T}_6$	Goal (0)	open the middle drawer of the cabinet
$\mathcal{T}_7$	Long (5)	pick up the book and place it in the back compartment of the caddy
$\mathcal{T}_8$	Long (0)	put both the alphabet soup and the tomato sauce in the basket

The sample budget of Section 5.1.2 is matched as follows: VLA-Adapter takes 30,000 steps at batch size 4 with its own learning rate, against our 15,000 steps at batch size 8, so both see 120k samples. All methods that use low-rank increments use the official PEFT implementation. The per-task artifact whose size the memory measurement of Section 5.1.4 reports is one increment file for DT-VLA, one full model for per-task fine-tuning, and the corresponding per-task state for the other baselines; the shared LoRA baseline keeps one increment for all tasks, a single file in total. The deployment axis uses one end-to-end run per scale, with 240 and 1,680 requests; Figure 2(b) plots the longer run, while the main text's 240-request figures come from the dedicated shorter run.

A.3 The Flat Panels of the Factor–Measurement Matrix

Reading the three measurements of Section 5.1.4 on the two factors of Section 5.1.3 gives six panels. Two of them carry the results and are shown in the main text; the remaining four are flat by construction, and we report them here so that the matrix is complete. The reasons are as follows: per-task storage and per-switch stall do not depend on how many tasks are learned afterwards (Figures 5 and 4); and serving updates no parameters and fixes the serving state before the first request, so neither accuracy nor per-task memory moves with the number of requests served (Figures 6 and 7). In every memory panel the shared LoRA line is one file amortized over the eight tasks it serves.

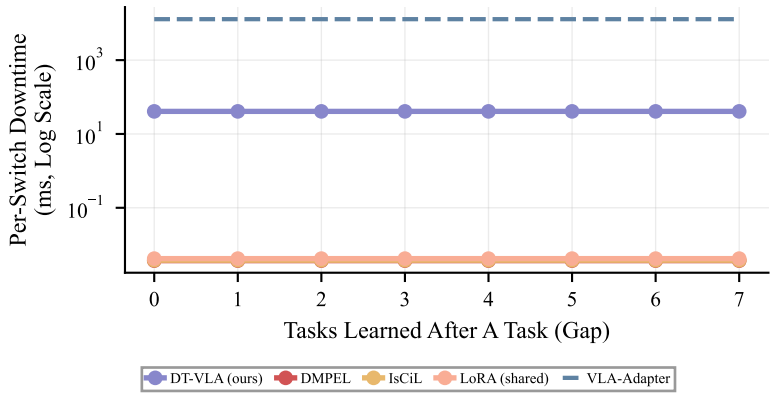


Figure 4: Per-switch downtime against the number of subsequently learned tasks.

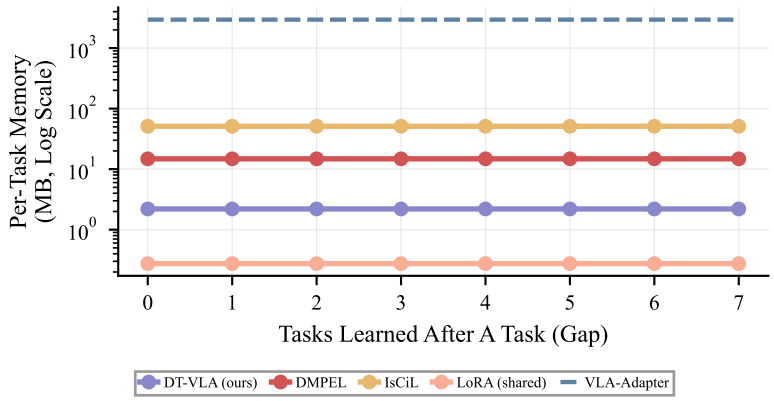


Figure 5: Per-task memory against the number of subsequently learned tasks.

A.4 The Increment Rank

Besides the two factors of Section 5.1.3 we vary a third, the rank r of each task-local increment, which sets each task's capacity; the main text fixes it at the default $r = 16$ of Section 5.1 and this appendix reads the three measurements of Section 5.1.4 along it. Accuracy climbs from $r = 8$ and then plateaus (Figure 8): halving the default capacity to $r = 8$ (≈ 1.1 MB per task) lowers the end-of-stream accuracy from 86.9 to 76.25, and quadrupling it to $r = 64$ (≈ 8.7 MB per task) yields 88.75. We keep $r = 16$: it sits on the accuracy plateau at the smallest per-task artifact, and the gain at $r = 64$ does not justify four times the memory per task. Past the plateau more capacity therefore buys memory rather than accuracy, while forgetting stays at zero at every rank, because the increments remain disjoint whatever their capacity.

The two ends of the curve differ in what they pay. At $r = 8$ the loss relative to the default falls on the harder half of the tasks, and one spatial task also gives up ground, so the smaller increment no longer covers every capability type reliably. At $r = 64$ the larger increment lifts the hardest

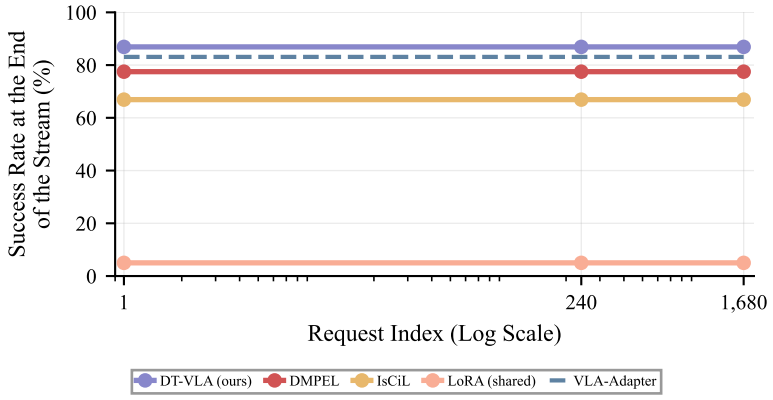


Figure 6: Success rate at the end of the stream against the number of task-switch requests.

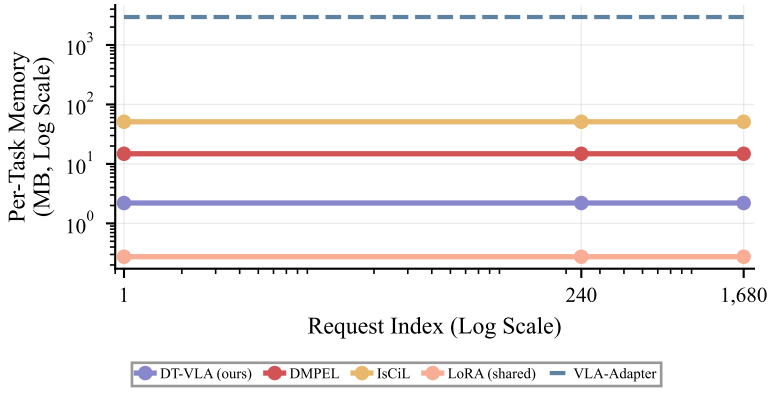


Figure 7: Per-task memory against the number of task-switch requests.

two-object long-horizon task $\mathcal{T}_8$ from 25 to 50, confirming that this task is capacity-bound at the default rank; the gain does not carry to the stream average.

The shared increment ends the stream at 2.5, 5.0, and 6.25 for $r = 8$, 16, and 64, respectively, showing consistent collapse across the tested ranks. DMPEL moves from 75.6 to 78.1 and IsCiL from 66.9 to 71.9 between their smallest and largest adapter ranks, consistent with the rank-independent machinery that dominates their storage, and the full-model recipe has no rank knob, entering as a constant reference at 83.1 that DT-VLA passes from $r = 16$ on.

The other two measurements follow the rank only through the size of the increment file that is stored and moved. Switching time grows mildly, from 35.9 to 70.2 milliseconds between $r = 8$ and $r = 64$, against the 12.9 seconds the full-model recipe reloads at any rank (Figure 9), and per-task storage grows in proportion to the rank r (Figure 10).

A.5 Additional Experimental Detail

This appendix carries the secondary readings of Section 5.2: the per-task detail behind the shared-increment collapse, the per-arm and per-variant numbers behind the ablations, and the routing, novelty-gate, and margin readings of the temporal module. Every claim in the main text stands on the numbers reported there; what follows adds detail rather than evidence.

How the shared increment collapses. The shared LoRA baseline of Section 5.2.1 ends the stream with six of its first seven tasks at 0 and the seventh at 25, the eighth being the one it has just learned. Its two averages therefore have different denominators: over all eight tasks it falls from 83.1 to 5.0, while the 89.3-point drop is the mean over the seven tasks that later learning overwrites.

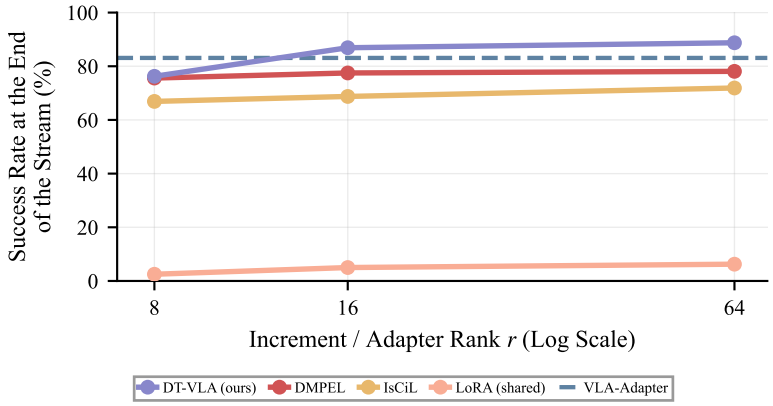


Figure 8: Success rate at the end of the stream against the increment rank.

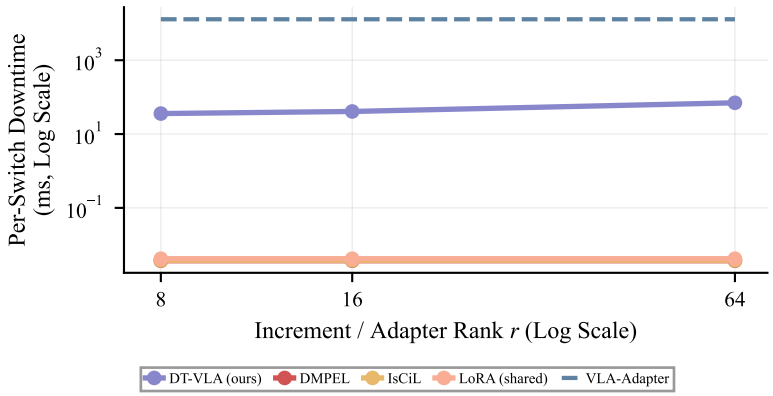


Figure 9: Per-switch downtime against the increment rank.

Reading the accuracy panels. In Figure 2(a) each point averages all tasks observed at that gap, so small non-monotonic wiggles reflect the changing task mix rather than forgetting. In Figure 2(b), each scale uses a fixed seed and adjacent requests always target different tasks.

The frozen-backbone arms. The diagnostic fixes the adaptation recipe and varies only the frozen backbone. With a backbone specialized to a *different* task family (mismatched), the same local adaptation reaches 100 on the goal task and 57.5 on the long-horizon tasks. With this paper's general backbone, it reaches 85 and 60. With a backbone specialized to the *same* family (matched), it reaches 95 and 95. The same weights serve every family without prior knowledge of an incoming task's family, and remain available for tasks outside these families.

Increment coverage. The all-linear-layer placement variant spreads a lower rank ($r = 8$) across all linear layers of the backbone instead of concentrating $r = 16$ on the attention projections. It reaches 81.9 at the end of the stream, still with zero drop, at 8.9 MB per task. The concentrated default is stronger here, almost entirely due to the hardest two-object long-horizon task $\mathcal{T}_8$ (10 against 25).

The temporal module. The log of Section 5.2.3 is measured on two functions. *Routing.* Index entries and queries use separate inputs. Routing resolves all 40 in-library requests correctly and stays 100% correct with any one entry removed and across 72 unseen paraphrases; two tasks have instructions differing only in a prepositional phrase, and routing separates them with the largest margins in the set. *Novelty gate.* Detection reaches an AUROC of 0.9992; its protocol and its two operating points follow below. Appendix A.7 reports one end-to-end run with no identity given.

Scheduler protocol. A task's index entry averages, over 16 demonstration frames, the backbone's last-layer hidden states pooled over the non-visual part of the sequence, that is, everything after the vision patches; a query is embedded the same way from the first frame of a deployment episode, so

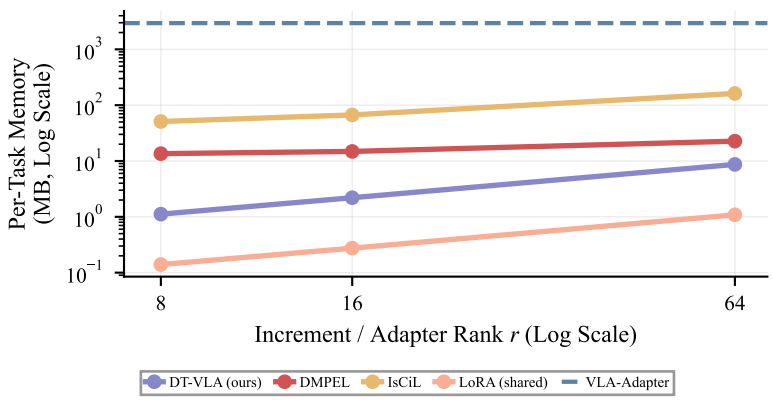


Figure 10: Per-task memory against the increment rank.

the index and the queries never come from the same inputs. Routing is measured on five requests per stored task, 40 in total. Novelty detection uses eight leave-one-out rounds: each task's increment and log entry are removed in turn, and its five requests must be flagged as new. It also tests three requests from each of eight held-in tasks that never had log entries and are thus new.

Scheduler similarity margins. Over the 40 in-library requests, a query's similarity to its own task's entry averages 0.774, against 0.454 to the entries of the other tasks. Across the 72 paraphrased requests the mean margin to the best wrong entry is 0.138, and the original-instruction control, evaluated on the same frames, also routes requests correctly.

Novelty-gate operating points. Detection reaches an AUROC of 0.9992, with the mean top similarity at 0.755 for in-library requests against 0.579 and 0.592 for the two kinds of new requests (held-out tasks whose entry was removed, and held-in tasks that never had one). The (u, τ_0, w) of the threshold read-out of Section 4.4 are calibrated on a held-in development set, independent of the held-out stream. The logistic keeps the read-out bounded, so the threshold stays within a calibrated range as the library grows. Two operating points are calibrated, the first maximizing F1 on the development set and the second precision-first; on the held-out stream they read $\tau = 0.7186$ (precision 0.821, recall 1.000, F1 0.901) and $\tau = 0.6598$ (precision 1.000, recall 0.969, F1 0.984). The end-to-end run of Appendix A.7 uses a threshold pre-registered between these two points.

Which increment serves which task. Figure 3(c) attaches every increment to every task, 10 episodes per cell. The only nonzero off-diagonal cell is a residual 30 between the two object-suite tasks ($\mathcal{T}_4$ served by $\mathcal{T}_3$'s increment), so each increment stores behavior specific to its own task.

A.6 Case Study Detail

The scenario values of Section 5.2.4 combine the measured per-task primitives, one 2.95 GB model against one 2.2 MB increment, with the stated bandwidth or capacity through $t = \text{size}/\text{bandwidth}$.

Three of them are used above. Pushing ten skills to fifty robots over a shared 100 Mbps uplink takes about 34 hours for full models against 1.5 minutes for increments. A skill library of $N{=}100$ tasks occupies about 295 GB against 3.2 GB, one resident backbone plus 100 increments. On an 8 GiB edge device, 2 full models fit in memory against roughly 2,500 increment-served tasks.

A.7 Running the Whole Loop Without a Task Identity

We detail the continuous three-stage run without supplied task identities (Section 5.2.4).

The run withholds one goal task, so the library holds seven of the eight, and fixes the novelty threshold at $\tau = 0.7114$ throughout, selected on a held-in development set and pre-registered beforehand. The 21 requests for stored tasks are each served by their best-matched increment and close at 81.0. When the withheld task arrives the gate flags all three of its requests as new $(3/3)$ and triggers the Update stage, which trains one increment for 15,000 steps, about one hour (Appendix A.8); while

adaptation runs, the nearest stored increment serves as a fallback. After adaptation, the withheld task routes $20/20$ and closes at 100, against 90 under the standard identity-given protocol.

A.8 Per-Task Adaptation Time

Every increment uses the same fixed training budget of $15{,}000$ steps; Table 2 reports the measured wall-clock on the workstation of Section 5.1. Each cycle covers one task's training plus one 20-episode evaluation, measured as the interval between successive increment checkpoints of the main stream; the first cycle contains the one-time backbone loading instead of a preceding evaluation. End to end, the whole eight-task stream, including every evaluation, completes in about 9.3 hours.

Table 2: Per-task adaptation cycles (minutes): $15{,}000$ training steps plus one 20-episode evaluation. *The first cycle includes backbone loading instead of a preceding evaluation.

	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$	$\mathcal{T}_5$	$\mathcal{T}_6$	$\mathcal{T}_7$	$\mathcal{T}_8$	Mean
Cycle (min)	67.9^*	69.9	69.7	67.9	68.1	68.2	69.2	67.6	68.6