



Scalable Behavior Cloning with Open Data, Training, and Evaluation

Arthur Allshire^{1*}, Himanshu Gaurav Singh^{1*}, Ritvik Singh^{1*}, Adam Rashid^{2*}, Hongsuk Choi^{1*}, David McAllister^{1*}, Justin Yu^{1,4}, Yiyuan Chen⁴, Huang Huang⁴, Pieter Abbeel^{1,3}, Xi Chen³, Rocky Duan³, Phillip Isola², Jitendra Malik^{1,3}, Fred Shentu^{1,4}, Guanya Shi^{3,5}, Philipp Wu⁴, Angjoo Kanazawa^{1,3}



Policies trained on ABC-130K being rolled out. More videos at <https://abc.bot>

Abstract. We introduce ABC, a fully open-source stack for manipulation with behavior cloning. At its core is ABC-130K: the largest open-source teleoperation dataset to date, featuring 3,500 hours of data spanning over 130K episodes across 195 diverse tasks. Furthermore, we open-source our accessible hardware setup, training infrastructure, and simulation pipeline. We also release 400 hours of sim-teleop data and provide a co-training recipe that produces correlated simulation and real-world evaluation, offering a reliable proxy for ablating model-design and training decisions before costly real-world evaluation. We explore various training recipes and compare common architectural choices for Diffusion Transformers (DiT) and Vision-Language-Action (VLA) models, grounding our findings in real-world evaluations. The resulting policies successfully execute dexterous tasks such as box folding and extracting credit cards from wallets. By providing a reproducible toolkit, we aim to place researchers on an equal footing, establishing the necessary foundation to learn the ABCs of Behavior Cloning together as a community.

*Core Contributor; work performed during an internship at Amazon FAR.

¹UC Berkeley ²MIT ³Amazon FAR ⁴XDOF ⁵Carnegie Mellon University

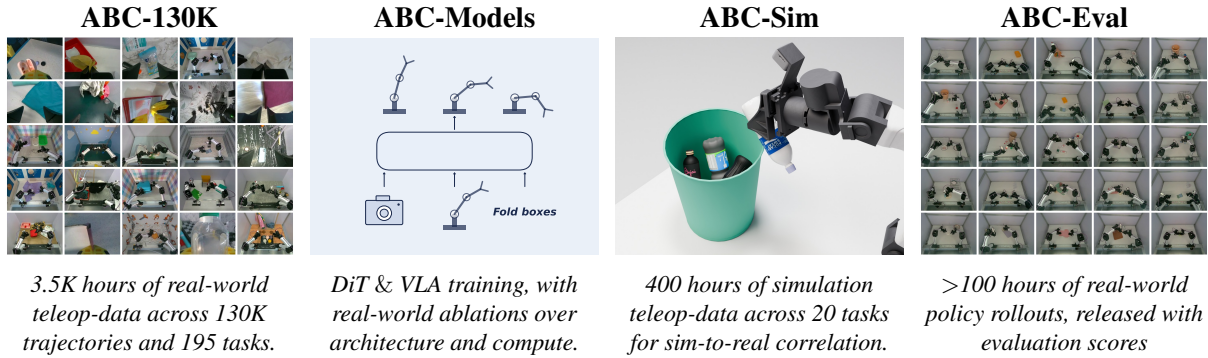


Figure 2: **An overview of the ABC stack.** ABC-130K provides large-scale real-world teleoperation data. ABC-Models instantiates DiT and VLA policies and studies architecture and compute-scaling choices through real-world ablations. ABC-Sim provides simulation environments and data for studying sim-to-real correlation. ABC-Eval provides a large-scale real-world evaluation suite with rollouts and rubrics. The full stack will be released for the community to build upon.

1 Introduction

The stack for behavior cloning (BC) is deep. In order to make progress, a practitioner must first set up robot hardware and collect demonstrations. Policy learning on this data requires both an efficient training system over heterogeneous data streams and an effective learning recipe, which the community has yet to converge on. The resulting pretrained policy may be sufficient for simple tasks, but more complex skills require further iteration, such as DAgger [1] finetuning, to achieve success. Rigorously evaluating design decisions for BC policies typically requires scaling up real-world evaluation, which is expensive and logistically demanding. As a result, progress at the frontier of robot manipulation remains opaque. State-of-the-art systems are often developed in industry labs on proprietary datasets, and their training recipes, when described, rarely provide enough detail to reproduce the results or determine which design choices drive performance.

To address this bottleneck, we present an open, end-to-end stack for BC. At the core of our release is ABC-130K, the largest open teleoperation dataset, comprising 3,500 hours of real-world interaction. It spans over 130,000 episodes and 195 distinct tasks, with a wide spectrum of manipulation primitives—including pick-and-place, handovers, tool use, and assembly—as well as highly dexterous behaviors like folding paper planes, folding boxes, and retrieving credit cards from wallets. We also release the accompanying infrastructure required to leverage this corpus, including our hardware setup, training and deployment code, and a simulation pipeline paired with 400 hours of simulation teleoperation data. We demonstrate sim-to-real evaluation correlation, enabling those without physical robots to iterate on their design choices and easily integrate custom task environments.

Using our pipeline we conduct an exploration of training recipes and architectural design choices for two classes of models: diffusion transformers [2] and vision-language-action (VLA) models [3]. We evaluate core components such as the vision encoder, conditioning schemes, and various pretraining recipes. Our findings are grounded in offline validation metrics (e.g. loss, action error, etc.), simulation performance, and real-world evaluations. The resulting baseline policies successfully execute dexterous tasks out-of-the-box, and we publicly release these model weights alongside our real-world evaluation rollouts. Figure 2 summarizes the full ABC stack, including ABC-130K, ABC-Models, ABC-Sim, and ABC-Eval. With the data and models we release, our policies are able to perform long-horizon and dexterous tasks such as inserting 6 AirPods in a row, inserting a key into a lockbox to unlock it, folding a cardboard box, and packing a student bag.

Robot learning for complex manipulation is still in its early stages, and optimal training paradigms are far from established. Existing open datasets [4, 5] have driven the field forward, but they offer insufficient scale, rely on bespoke and expensive hardware platforms, or lack the necessary training infrastructure and evaluation suites. Our main goal is to accelerate this ongoing exploration by releasing a comprehensive and extensible manipulation stack.

2 ABC-130K Dataset

Publicly available robot data for policy learning has grown substantially in recent years. DROID [4] and BridgeData-V2 [6] provide high-quality single-arm data on the Franka and WidowX robots, respectively, but on tasks that are largely single-step pick-and-place. Open X-Embodiment [5] spans many embodiments and tasks, but with wide variation in quality along with noisy language and action-space metadata makes it hard to use. AgiBot-World [7] offers 3,000 hours of bimanual data, although on a \$30,000 robot. Closest to ours is MolmoAct-2 [8], which offers 750 hours of data on the YAM platform. Because episodes are stored as encoded video across multiple camera views that range up to 469 s, naively decoding frames on the fly becomes the dominant throughput cost at this scale. We detail the data-loading challenges this introduces and our optimized dataloader design in Appendix B.1.

With **ABC-130K**, our goal is to bring the best of these works together: high-quality data for complex, diverse tasks on a bimanual, inexpensive robot at scale. Our dataset comprises **134,806 episodes** across **195 tasks**, totaling **3,553 hours** of bimanual manipulation (see Fig. 3). It spans diverse manipulation primitives such as pick-and-place, folding, handover, insertion, tool use, and assembly. We organize the 195 tasks into 7 primitive categories that capture the dominant contact modes and control strategies. Within each task, we vary the objects and initial configurations. Description of our primitive categories can be found in Appendix A. Figure 4 provides the distribution of data across tasks within each category. Figure 3 shows random samples of the top camera frames from episodes in the data.

We also provide teleoperation metadata, including anonymized teleoperator IDs and collection timestamps. In addition, a 1,552-hour subset of **ABC-130K** includes subgoal annotations: contiguous subtrajectories labeled with subtask descriptions. In Appendix H, we show how these metadata and annotations can be used for policy conditioning.



Figure 3: **Dataset overview.** Random top camera frame samples from ABC-130K. We recommend zooming in to see the diverse tasks, backgrounds, and objects present in the data.

3 ABC-Models

We study two policy families for large-scale behavior cloning: **ABC-DiT**, a diffusion-transformer policy with lightweight vision and language encoders, and **ABC-VLA**, a VLM-backed policy with a diffusion action head. These two models reflect common design choices in recent robot learning systems: diffusion policies built on pretrained visual representations [9, 10], and vision-language-action models that adapt pretrained VLM backbones for robot control [11–14]. Below, we study how base encoders, in-

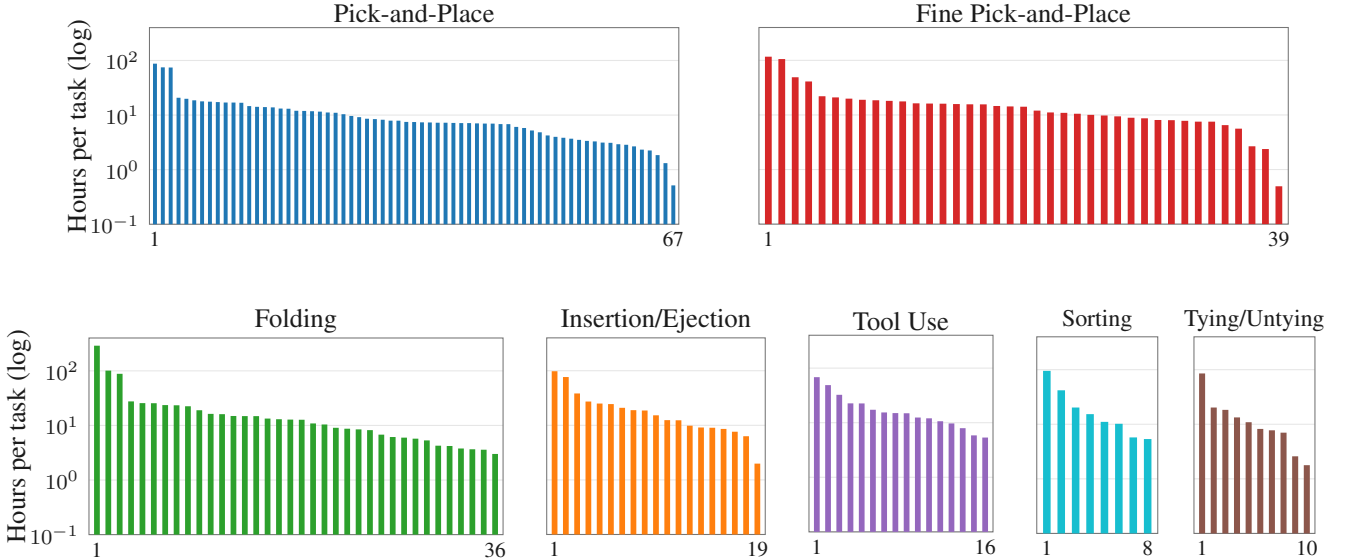


Figure 4: **Per-task hours by primitive category.** Hours-per-task on a log scale in descending order within each category. Top row: *Pick-and-Place* and *Fine Pick-and-Place* (the two largest categories in terms of number of tasks). Bottom row: the five remaining categories. Descriptions of each task category along with example images from the data can be found in Appendix A.

Task	Metric	VLA variants			DiT variants		
		Pooled adaLN	FAST + X-Attn	X-Attn	DINOv3-xattn	CLIP-adaln	CLIP-xattn
Bottles	Strict	67.3%	8.0%	0.0%	73.5%	11.8%	38.9%
	Progress	83.0%	44.0%	5.2%	93.1%	53.6%	74.1%
Dishrack	Strict	30.0%	2.8%	0.0%	23.2%	28.6%	34.7%
	Progress	75.7%	47.2%	25.0%	74.1%	72.3%	81.3%
Mugs	Strict	1.0%	0.0%	0.0%	2.0%	0.0%	0.0%
	Progress	25.5%	6.5%	5.0%	35.3%	15.9%	21.0%
Aggregate	Mean strict	32.8%	3.6%	0.0%	32.9%	13.4%	24.5%
	Mean progress	61.4%	32.6%	11.7%	67.5%	47.3%	58.8%
	Latency (ms)	17.24	19.24	19.24	37.4	27.5	37.5

Table 1: **Architecture ablations.** Real-world evaluation of VLA and DiT architecture variants after 200k training steps. Strict success reports the percentage of tasks completed; progress reports average task progress. Latency is measured in ms/chunk for 10 diffusion steps. The models are trained on an internal corpus of 7,000 hours of robot data.

terfaces between backbones and action heads, and training hyperparameters affect performance across these two families. Unless otherwise noted, the comparison models in this section are trained on an internal corpus of 7,000 hours, which we had for development before the release dataset was finalized. We take the modeling decisions from this study to train and evaluate our final ABC models on the publicly released **ABC-130K** in Section 5. All models take as input three 224×224 images from the top and two gripper cameras, along with a 14D proprioceptive joint state. Non-square images are padded using letterbox padding. The policy outputs a 30-step chunk of absolute joint-position targets, and states and actions are normalized with z-score normalization. The architecture details for both models can be found in Appendix B.2 and the training details in Appendix B.3. All models are trained on NVIDIA H200 GPUs.

All real-world evaluations in this section are on three tasks: *throwing plastic bottles into a bin*,

loading plates into a tabletop dish rack, and turning mugs right side up. Each task is evaluated for 50 trials. Our policies are executed at 30 Hz. Following [15], we execute our policies asynchronously with an action prefix. This is done to prevent mode switching, thereby ensuring smooth trajectories. As noted in Appendix H.2, depending on the task, performance can improve by modulating the length of the prefix as well as the execution horizon of the action chunk.

3.1 ABC-DiT

We study various architectural choices for diffusion transformers [2]. We specifically investigate how the DiT head communicates with various vision encoders. All our DiTs use 32 layers, 24 attention heads, and a hidden dimension of 1536, which is larger than the DiT-XL architecture. We chose this large model size to ensure that architecture choices were not bottlenecked by network capacity, allowing us to better understand what matters at scale (comparisons of various DiT sizes are in Appendix B.4). We compare the following three variants below.

CLIP-AdaLN: We begin with a baseline inspired by the LBM architecture [10]. The policy backbone is a DiT [2] that takes as input a noised action chunk and predicts the velocity of the conditional probability flow. Each image is encoded independently using a CLIP ViT-B image encoder [16], and the resulting CLS tokens are concatenated to form a compact visual representation. The task language instruction is embedded using the CLIP text encoder and projected with an MLP. This vector is concatenated with the visual representation, an MLP projection of the robot proprioception, and the diffusion timestep embedding to form a single conditioning vector. The DiT is then conditioned on this vector through a standard adaLN conditioning mechanism [2].

CLIP-Cross-Attention: In the previous variant, the AdaLN conditioning compresses each image into a single token. This compression might lose fine-grained spatial information that is important for manipulation. Alternatively, we can have the DiT attend to the sequence of visual tokens. Since directly attending to all tokens is expensive, we use a set of learned latent queries to attend over all vision tokens. For all our experiments, we use 12 latent query tokens per image as it offers a good balance of expressivity and memory overhead. In the DiT block, after each self-attention layer, we add a cross-attention layer where the query tokens are taken from the noised action tokens and the keys and values are the pooled visual tokens. The non-visual conditioning variables remain in the AdaLN pathway.

DINO-Cross-Attention: Image-text alignment is not necessarily the objective most aligned with low-level robot actions. Manipulation policies often require precise spatial cues that may not be optimally preserved by CLIP but may be captured with more pixel aligned representations like DINOv3 [17]. We use the cross-attention architecture and replace the CLIP image encoder with a DINOv3 ViT-B encoder.

All models are trained for 200k steps with a global batch size of 4608. We linearly warmup the learning rate for the first 1000 iterations to 1×10^{-4} and then keep it constant throughout training. We use the AdamW optimizer [18] with a weight decay of 0.01. We use a learning rate scale of 0.1 for the vision encoder. We use gradient clipping with a maximum gradient norm of 10. The proprioception input is dropped out with a probability of 0.1.

The results are shown in Table 1. Replacing global AdaLN-based visual conditioning with pooled cross-attention improves task progress, suggesting that preserving token-level visual information is important for action generation. Replacing CLIP with DINOv3 further improves performance, indicating that the choice of visual representation is also a key design factor for real-world manipulation. For the rest of this paper, we use DINOv3 and this pooled cross-attention conditioning scheme as our *ABC-DiT* model. This model has 2B parameters in total.

3.2 ABC-VLA

For *ABC-VLA*, we build on Gemma 3 [19], a 4B-parameter VLM with a SigLIP vision encoder and a Gemma language decoder, which we pair with a small DiT action head to generate action chunks. We consistently find that freezing the VLM backbone performs poorly, so we finetune the backbone using the action prediction objective. The connector between the VLM and action head routes vision-language features into the action expert and can provide gradients back to update the VLM. Existing VLAs adopt many different connector mechanisms, but their effects are typically entangled with differences in data

and training recipe, so their relative merits are difficult to isolate. We therefore compare three representative connectors directly on our data, fixing the backbone, action head, and training recipe so that only the connector varies.

Cross Attention: We run the VLM on the image, language, and proprioceptive context, extract final-layer token features, and project them to the DiT token dimension. These projected VLM features serve as the keys and values for a cross-attention layer in each DiT block, while the noisy action token stream inside the DiT queries them. The VLM backbone is finetuned end-to-end by the diffusion objective, so diffusion gradients flow through the DiT’s cross-attention back into the VLM.

Cross Attention + FAST: Following [20], we concatenate FAST-tokenized [21] actions in the VLM token sequence and train with next-token-prediction CE to finetune the VLM. Simultaneously, the DiT cross-attends to the context token features at the final VLM layer, which we detach to “insulate” the backbone from diffusion gradients. In our experiments, cross attention slightly outperforms $\pi_{0.5}$ -style [13] reuse of the VLM’s keys and values.

Adaptive LayerNorm (AdaLN): We adapt the LBM-2 [10] conditioning scheme, where VLM features are compressed into a fixed-length conditioning vector that modulates the DiT through adaLN [2]. Unlike the cross-attention variants, the DiT action tokens do not directly attend to VLM tokens. Instead, VLM token features from the final layer are summarized by an attention-pooling module into 8 512-dimensional tokens, which are flattened and projected to produce the DiT’s adaLN shifts, scales, and residual gates. We use attention pooling with QK-Norm [22] in place of LBM-2’s register tokens.

Variance Reduction For *AdaLN* and *Cross Attention*, the diffusion training loss is an expectation over three random variables: the data sample x , the noise ϵ , and the timestep τ (i.e., $\nabla\mathcal{L} = \mathbb{E}_{x,\epsilon,\tau}[\nabla\ell(x, \epsilon, \tau)]$). Since our DiT is small relative to the VLM (42M vs. 4B parameters), it is efficient to amortize a single VLM pass across many (ϵ, τ) draws. Specifically, in the forward pass, we replicate the VLM features and clean action k times within the batch and pair each copy with an independent (ϵ, τ) sample. In the backward pass, the gradient expansion averages at the conditioning interface, so the VLM backward cost is independent of k . This yields a strictly lower-variance gradient estimator with negligible overhead: 1.346 s/step at $k = 1$ and 1.366 s/step at $k = 8$ on H200. Figure 6 shows the improvement that the multiple draws trick provides.

Table 1 compares the three variants. Vanilla cross-attention performs worst on aggregate progress (9.2%), consistent with prior studies. FAST co-training helps (29.6%) by replacing the diffusion gradient on the VLM with a cross-entropy gradient. AdaLN beats both variants on our data (67.5%) despite exposing the VLM to denoising gradients. This suggests that diffusion gradients are not intrinsically incompatible with VLM features if given a proper interface.

3.3 Scaling compute across batch size and training steps

When training policies, there is no consensus on the optimal batch size. Models like OpenVLA [23] and LBM [10] use 2048 and 2560, respectively, whereas GR00T N1 [11] uses 16,384. In order to more precisely understand the effect of increasing the batch size, we train three different variants of *ABC-DiT* and *ABC-VLA*. We train both on batch sizes of 1536, 4608, and 9216. Figure 5 shows the plot of real world task success and progress for these variants for 100K and 200K training steps. For the relatively smaller batch sizes, the DiT outperforms the VLA. However, as we scale the batch size to 9216, the DiT does not improve as much, whereas we see the VLA has a much larger jump in performance. Furthermore, we see that the VLA, at higher batch sizes, is able to outperform the DiT in real world performance, though the DiT is more compute-efficient.

3.4 Offline Metrics for Policy Evaluation

Real-world evaluation does not scale to the number of minor architecture and hyperparameter decisions involved in developing a policy, so in practice we make many of these using offline metrics and reserve real-world evaluation for the major architectural comparisons. This raises the question of whether offline metrics are predictive of real-world performance at all.

For all the models trained in the previous sections, we compare the real-world strict success rates and progress rates against training loss, validation loss, and validation action error to assess whether offline

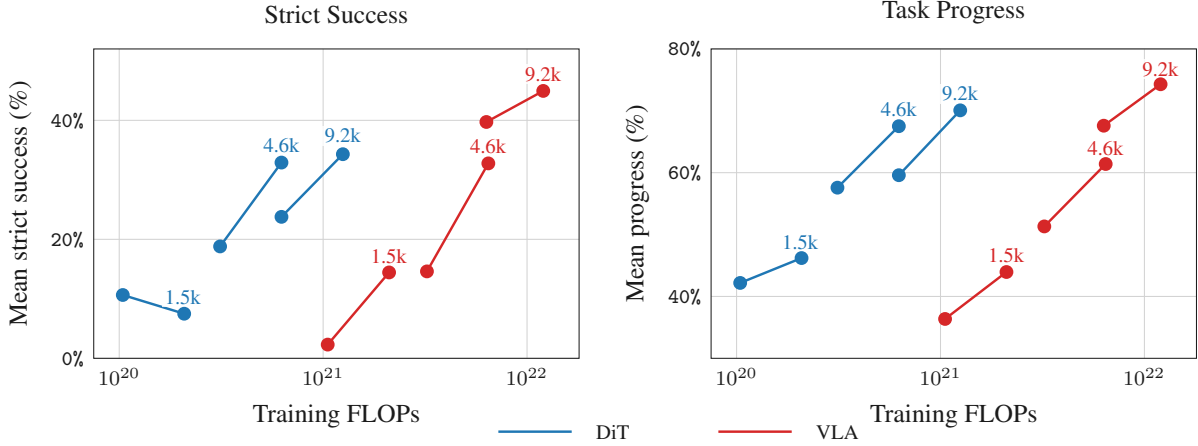


Figure 5: **More training steps and larger batch size give better performance.** Real-world task success (left) and task progress (right) with training compute for DiT and VLA policies; each connected pair uses the same effective batch size (approximately 1.5K, 4.6K, or 9K) evaluated at different checkpoints. We consistently find that the DiT is more flop-efficient.

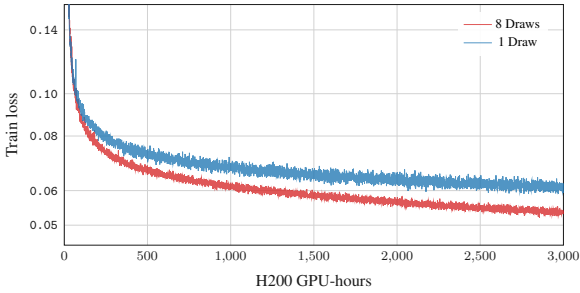


Figure 6: **Eight diffusion draws reduce VLA train loss at fixed accelerator time.** Training loss versus GPU-hours for one- and eight-draw VLA diffusion training.

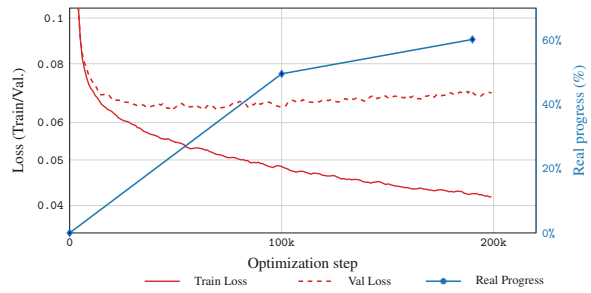


Figure 7: **VLA success rate and offline metrics over time.** VLA training and validation loss with real-world progress across checkpoints.

metrics predict real-world performance. While training and validation loss are defined as the conditional flow matching loss of the models on their respective data splits, validation action error is the L2 distance between predicted and ground-truth action chunks, computed by running 10 diffusion steps per model with no action prefix.

Training loss and validation action error correlate with real world performance. The resulting plots are shown in Figure 8. Training loss and validation action error are both negatively correlated with real-world performance, with statistically significant Pearson and Spearman correlations. Validation action error shows the strongest negative correlation. In contrast, validation loss is not significantly correlated with real-world performance. We note that validation action error is only meaningful when the number of diffusion steps is held fixed, since the error can be trivially reduced by lowering the number of diffusion steps, but this does not improve real-world performance. These correlations hold true across different model architectures and training batch sizes. Figure 7 shows an example of the phenomenon with the VLA—the training loss goes down over time as real world progress improves, meanwhile validation loss initially goes down but then increases even though performance improves over time. Offline metrics, such as validation action error and training loss, thus give us a cheap, scalar proxy for evaluating the many small design decisions in this section without having to test everything in the real world. While they cannot reliably predict task success rates or reveal failure modes, we still find them to be helpful for iterating on modeling decisions. For a higher-fidelity proxy, we turn to simulation (Section 4).

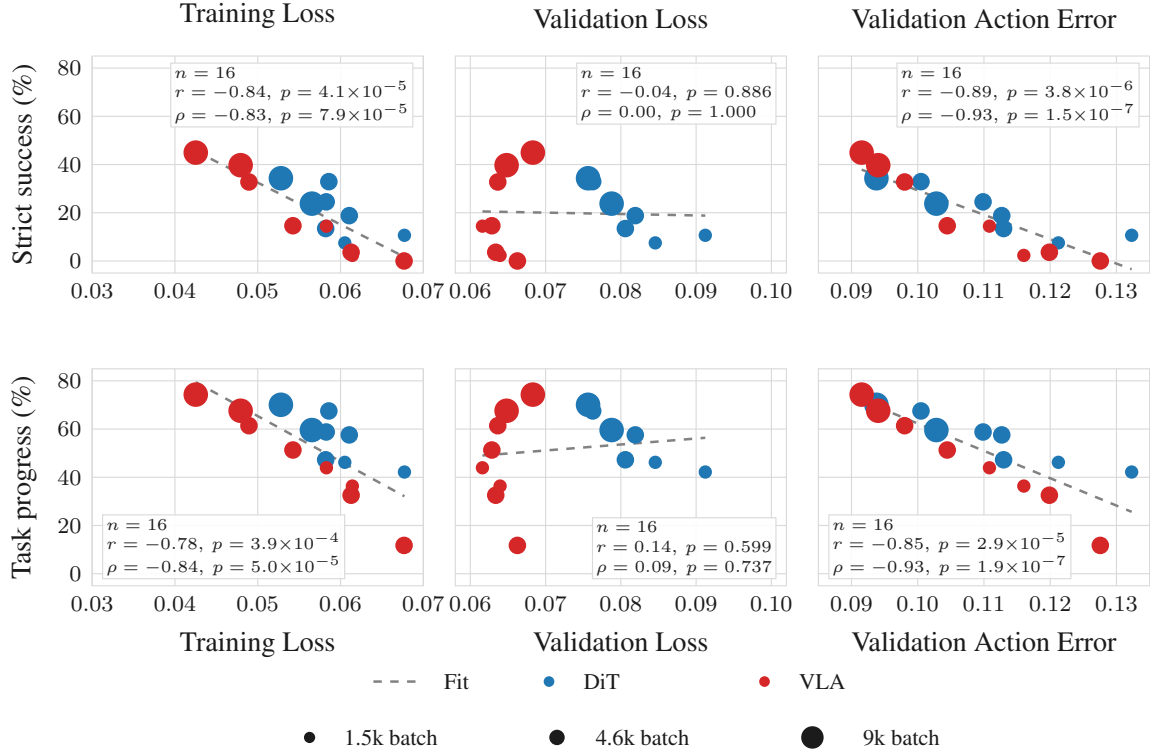


Figure 8: **Offline metrics versus real-world policy performance.** We analyze correlation between checkpoint training diagnostics and real-world success. Training loss and validation action error are both negatively correlated with real-world performance. Each point is one checkpoint, with real-world success averaged across the three evaluation tasks.

4 ABC-Sim

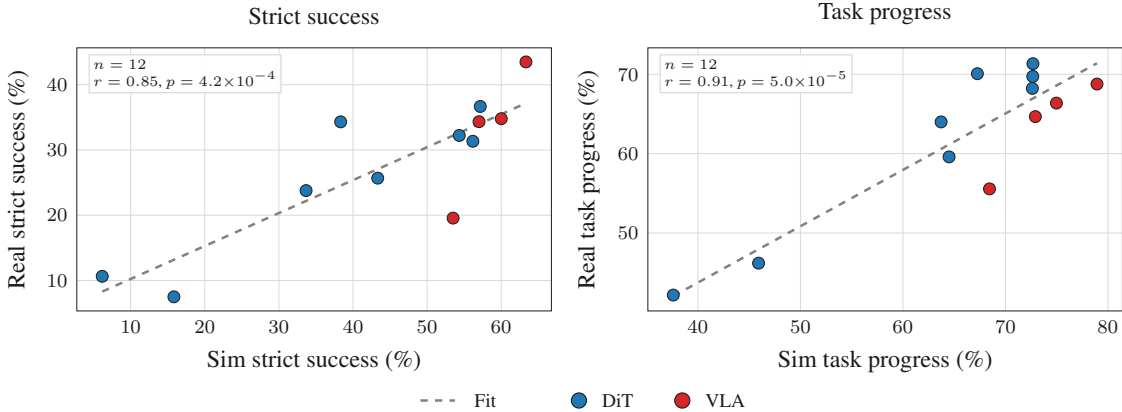


Figure 9: **Sim-to-real progress correlation.** Task-level simulation progress correlation with real-world progress across DiT checkpoints.

Offline metrics allow us to assess many decisions cheaply but cannot report task success or reveal failure modes. Simulation closes this gap as it yields actual success rates and watchable rollouts, and lets researchers without hardware iterate. Our release includes **ABC-Sim**—a suite of simulation environments and accompanying teleoperation data.

We re-create the real robot setup in MuJoCo [24]. The images are rendered inside the simulation, leading to low-fidelity captures. We also expose a pipeline for re-rendering these scenes in Blender to

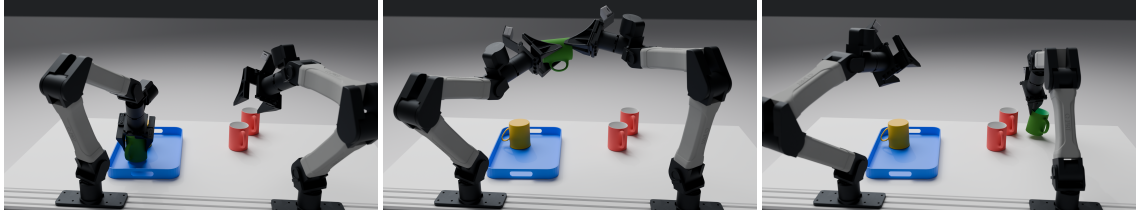


Figure 10: **ABC-Sim tasks.** Policy rollouts for our simulated task built in MuJoCo for sim-to-real evaluation. The multi-step *turning mugs right-side up* task, shown across three stages (grasp, flip, place). Rollouts on more simulated tasks can be found in the Appendix. By default we render the rollouts in MuJoCo but we also provide a Blender pipeline for generating high-quality renders.

create higher-fidelity images with path tracing. This allows users to both generate and train on higher-quality simulation renderings. We release simulation environments for the following ten tasks (Fig. 10):

1. throwing plastic bottles into a bin
2. sweeping paper scraps off the table
3. turning mugs right-side up
4. loading plates into tabletop dishrack
5. hanging a mug on a mug rack
6. setting up chess pieces on a board
7. spelling “abc” with blocks
8. placing markers in a drawer
9. pouring beads
10. in-hand object handover

The simulation runs at 240Hz and visual observations are sent to a VR headset at 60Hz. The current framework supports both Meta Quest and Apple Vision Pro headsets. The teleoperator uses a pair of GELLO [25] leader arms in order to teleoperate the YAM arms in simulation to complete the various tasks. VR-based teleoperation for extended periods of time can lead to motion sickness in operators. To mitigate this, we enable pass-through rendering so that the operators view the real world behind the simulated robot setup. We release over 400 hours of VR teleoperation data in simulation on all of these tasks, as well as the VR teleoperation stack to collect teleop data similar to [26–28].

To assess whether simulation performance is indicative of real-world performance, we conduct evaluations for sim-to-real correlations. We train *ABC-DiT* and *ABC-VLA* on ABC-130K and select four checkpoints per model. We also select four different *ABC-DiT* checkpoints trained on a mixture of our internal 7K hour dataset and the simulation data for a total of 12 checkpoints. We evaluate each of them across the same three tasks used in Section 3: *throwing plastic bottles into a bin*, *loading plates into tabletop dishrack*, and *turning mugs right-side up*. Each task was evaluated for 50 trials. For both simulation and real results, we average the success rates and progress across the tasks for each checkpoint.

Simulation evaluations correlate strongly with real world performance. The results of the evaluation are shown in Figure 9. For the strict success metric, we observe a Pearson correlation of $r = 0.85$ ($p = 4.2 \times 10^{-4}$, $n = 12$) and for task progress we observe $r = 0.91$ ($p = 5.0 \times 10^{-5}$, $n = 12$). This indicates that simulation is a meaningful predictor of real performance. We hope that given this correlation, our sim stack can provide a way for researchers to debug design decisions in their policies prior to conducting expensive real world evaluations, or for those without hardware to contribute to the research frontier.

5 Real World Capabilities

In this section, we demonstrate the capabilities of our trained policies on the publicly released ABC-130K set. We begin by showing the performance of the pretrained base policies in Section 5.1. Section 5.2 shows the importance of pretraining when finetuning on dexterous tasks. Lastly, for Section 5.3, we study the case of box folding, a task for which naive finetuning is insufficient for real world performance, and how DAGger can be used to specifically collect recovery behaviors. Appendix H studies how conditioning can steer policy behavior at inference time—selecting an operator’s style, trading off motion smoothness against visual responsiveness, and resolving temporal ambiguity within a task. Two of these are enabled directly by metadata we release with ABC-130K—per-episode operator identities

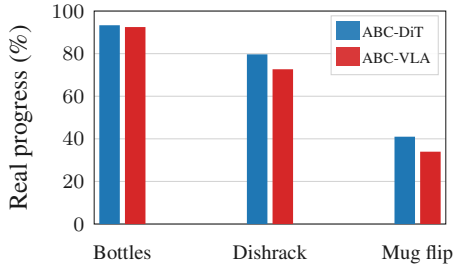


Figure 11: **ABC-DiT/ABC-VLA real-world performance.** Task progress across the three real-world tasks.

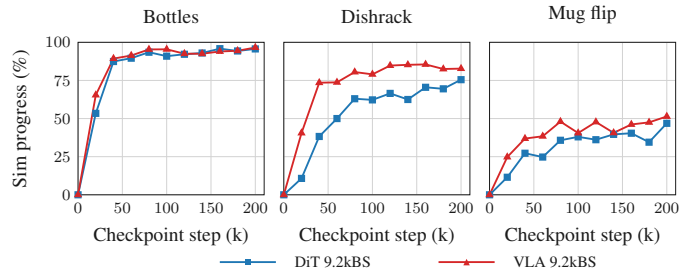


Figure 12: **Pretraining simulation progress.** Mean task progress across three tasks in simulation at multiple checkpoints during pretraining of *ABC-DiT* and *ABC-VLA* on ABC-130K.

and subtask annotations—illustrating how richer dataset labels, not just more trajectories, translate into finer-grained control at test-time.

5.1 Pretraining Capabilities

We train the *ABC-DiT* and *ABC-VLA* models, which were, respectively, the best DiT and VLA architectures found in Section 3 on the ABC-130K for 200,000 steps with the same training settings as in Section 3. In Figure 12, we plot the progress of three tasks across various points during training. We can see that for both models, as training iterations continue, simulation performance tends to increase. We also evaluate each task 50 times in the real world in Figure 11.

5.2 Downstream single-task Finetuning

We test the effectiveness of our pretrained policies as a useful initialization for downstream finetuning. We choose four high-precision, dexterous tasks for this comparison: i) extracting a credit card from a wallet, ii) sorting LEGO bricks, iii) inserting a pen cap, and iv) unscrewing a bottle cap. For each task we train three single-task policies that differ only in their initialization: (i) trained from scratch on the target task; (ii) finetuned from a checkpoint pretrained on the public ABC-130K (3,500 hours); and (iii) finetuned from a checkpoint pretrained on a larger internal corpus of 7,000 hours, included to test whether the returns from pretraining continue to scale. Figure 13 reports these results and Appendix G details our evaluation rubrics. We use *ABC-DiT* for the LEGO sorting, pen-cap insertion, and bottle-cap unscrewing, and *ABC-VLA* for the credit-card extraction task. We found that *ABC-VLA* was markedly better as a base model compared to *ABC-DiT* for this task. Pretraining quality has a substantial effect on downstream performance. Across all tasks, finetuning from a checkpoint pretrained on more data yields a stronger single-task policy. More details about the task, rollouts, evaluation rubric, and finetuning recipe can be found in Appendix E.

5.3 DAgger

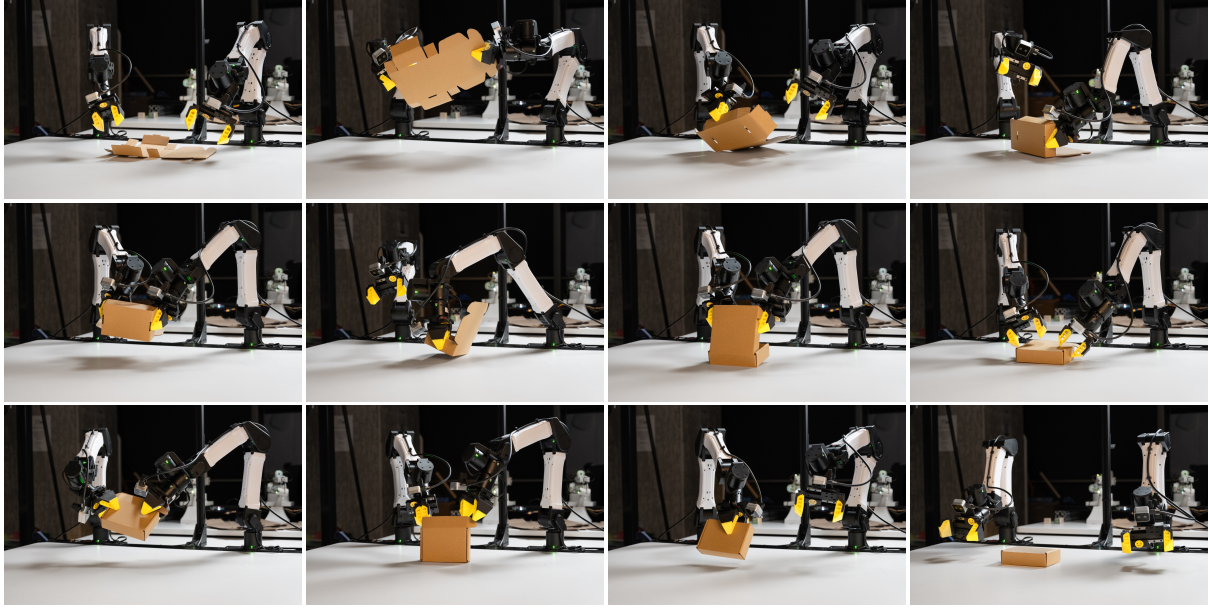


Figure 15: **The robot autonomously folds a cardboard box and closes the lid.** DAgger intervention data is crucial to get good performance on this long horizon, precise manipulation task.

To demonstrate the ability of our stack to learn very dexterous tasks, we perform box folding with our policies. We start with the pretrained ABC-DiT model which cannot achieve any real world success on this dexterous task. Normally, different teleoperators collect demonstrations in different ways making the data difficult to fit. Due to this, we collect an extra 10 hours of “collapsed data” which we define as data collected with a relatively stricter standard operating procedure (SOP). We fine-tune the ABC-DiT model on this dataset for 75k steps with a batch size of 720. After training, the model performance is still quite low at only 24%. While collecting more data under this SOP would likely improve performance, doing so for every dexterous task is prohibitively expensive. One key insight is that the fine-tuned policies exhibit a good understanding of the task, but struggle only to make the intermediate adjustments and corrections necessary to complete the task. Instead of collecting data for the entire rollout, we can simply collect recovery behaviors to make these fine-grained adjustments.

This motivates collecting DAgger intervention data. During collection, we roll out the policy, and intervene when it enters a state where it is struggling and we teleoperate the robot to complete the task. We record data for the whole rollout, not just the human intervention. Each round of DAgger constitutes roughly 1-1.5 hours of data. During the first round, the ratio of intervention data is around 30%, which

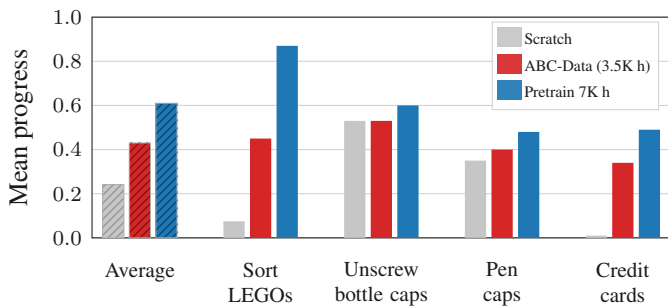


Figure 13: **Effect of pretraining on downstream finetuning.** Single-task finetuning performance improves with more diverse pretraining across tasks.

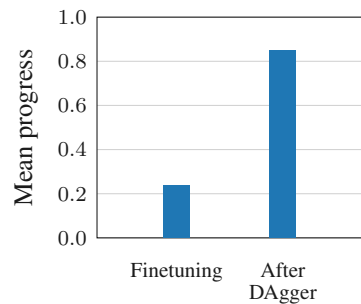


Figure 14: **Box folding: effect of DAgger.** Average task progress for a finetuned policy, before and after DAgger.

drops down to 15% during the second round. While the fine-tuning data was collected inside a cage, we opt to collect the DAgger interventions with no cage.

After each round of DAgger, we take the base model from the previous round and continue training for 75k steps. We maintain an 80:10:10 ratio where 80% of the data is from the previous round, 10% is from interventions in the current round, and the other 10% is from the rest of the episode in the current round. This implicitly overweights interventions while still keeping the rollouts from the whole episode. We find that this ratio is better than training on either whole episodes or interventions alone. This is partly because training on intervention data alone allows the policy to exploit spurious correlations between the source distribution (collected in cages) and the intervention data (collected without), which ultimately worsens the policy. Figure 15 shows qualitative results on this task; see the website for videos showcasing the dexterity, and Appendix F for details of our novel DAgger collection infrastructure.

6 Related Work

6.1 Large-scale open datasets for robotics

We target the lack of large-scale, dexterous data for low-cost bimanual systems. Early efforts towards data scaling primarily focused on single-arm manipulation. BridgeData-V2 [6] released ~100 hours of robot data across 24 environments on a low-cost WidowX arm. DROID [4] provides 350 hours of curated teleoperation data on a single Franka arm across more than 500 scenes, while RH20T [29] contributes 110k contact-rich trajectories across 41 skills and 5 single-arm platforms. Open X-Embodiment [5] aggregates trajectories from multiple embodiments and institutions into a unified source. However, it is a heterogeneous collection with widely varying per-dataset quality, action spaces, and control frequencies.

More recently, the community has shifted focus toward bimanual and dexterous manipulation. The ALOHA ecosystem [30–32] has popularized bimanual teleoperation for contact-rich tasks, though the hardware remains moderately expensive (~\$20,000) and the released datasets are smaller in scale compared to our collection. Taking a complementary approach to scaling, GR00T N1 [11] combines 88 hours of real teleoperation data with 780k synthetic trajectories generated via simulation and video models. The MolmoAct2-BimanualYAM Dataset [8] is a bimanual dataset containing 28 unique tasks and 720 hours in total.

In contrast, our dataset contains over 3,500 hours of high-quality, real robot data on a single low-cost bimanual embodiment (our \$8,000 YAM station), spanning more than 100 tasks with diverse skills and background scenes. The closest dataset to our scale is AgiBot-World [7], which features approximately 3000 hours of real-world bimanual data across roughly 200 tasks. However, AgiBot-World was collected on a substantially more expensive platform (the \$30,000 AgiBot G1). Additionally, while AgiBot-World emphasizes long-horizon macro-manipulation, we also provide data for various fine, dexterous manipulation tasks. We feature tasks such as extracting cards from wallets, folding paper planes, and unlocking doors using a key.

6.2 Large-scale Behavior Cloning

Policy architecture and optimization details are critical for training robot policies at scale. While the broad strokes of these designs are publicly available—such as the effectiveness of diffusion objectives for action modeling [9] and the benefits of reusing visual features from pre-trained models—the exact implementations vary widely. For instance, the Large Behavior Model (LBM) [10] initializes its image backbone with CLIP [33] and uses adaptive layer normalization (adaLN) [2] to condition its diffusion model. Conversely, other works leverage Vision-Language Model (VLM) backbones: GR00T-N1 [11] cross-attends to VLM outputs, $\pi_{0.5}$ [13] cross-attends layer-wise to the KV cache of its VLM. Despite these high-level insights, finer details of these training recipes are rarely reported, and the underlying training code is largely kept closed-source [10, 12–14, 34]. Because these large-scale efforts are trained and evaluated on proprietary datasets with diverse and often unknown compute budgets, the broader community cannot evaluate these design choices without reproducing deep implementation stacks from scratch. Our dataset, along with the accompanying infrastructure for large-scale training and evaluation, provides an open-source foundation to answer research questions in manipulation.

7 Infrastructure

Beyond the usual data and modeling details, the low-level systems implementations are equally as important. To this end, we also release the surrounding infrastructure for this project. Appendix B.1 contains the details for efficient data loading from multiple video streams to improve training time. Appendix C details our lightweight and extensible real robot code framework. Appendix D describes the inference optimizations for both ABC-DiT and ABC-VLA. Appendix F contains the implementation details for our novel DAGger collection setup with passive leader arms.

8 Requests for Research

We present ABC-130K, ABC-Models, and ABC-Sim with the aim of seeding an open-source, reproducible ecosystem for robot learning research. A lot of interesting open questions remain to be answered for large-scale real world robot learning, including but not limited to:

- **History conditioning:** policies that condition on a sequence of past observations rather than the current frame alone.
- **Scaling laws for BC:** how performance scales with model and data size for imitation learning.
- **RL finetuning:** improve ABC base models with reinforcement learning beyond the base model.

We hope ABC provides a foundation for answering these questions and many others.

References

- [1] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning, 2011. URL <https://arxiv.org/abs/1011.0686>.
- [2] William Peebles and Saining Xie. Scalable diffusion models with transformers, 2023. URL <https://arxiv.org/abs/2212.09748>.
- [3] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, Pete Florence, Chuyuan Fu, Montse Gonzalez Arenas, Keerthana Gopalakrishnan, Kehang Han, Karol Hausman, Alexander Herzog, Jasmine Hsu, Brian Ichter, Alex Irpan, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Lisa Lee, Tsang-Wei Edward Lee, Sergey Levine, Yao Lu, Henryk Michalewski, Igor Mordatch, Karl Pertsch, Kanishka Rao, Krista Reymann, Michael Ryoo, Grecia Salazar, Pannag Sanketi, Pierre Sermanet, Jaspier Singh, Anikait Singh, Radu Soricut, Huong Tran, Vincent Vanhoucke, Quan Vuong, Ayzaan Wahid, Stefan Welker, Paul Wohlhart, Jialin Wu, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. Rt-2: Vision-language-action models transfer web knowledge to robotic control, 2023. URL <https://arxiv.org/abs/2307.15818>.
- [4] Alexander Khazatsky, Karl Pertsch, Suraj Nair, Ashwin Balakrishna, Sudeep Dasari, Siddharth Karamcheti, Soroush Nasiriany, Mohan Kumar Srirama, Lawrence Yunliang Chen, Kirsty Ellis, Peter David Fagan, Joey Hejna, Masha Itkina, Marion Lepert, Yecheng Jason Ma, Patrick Tree Miller, Jimmy Wu, Suneel Belkhale, Shivin Dass, Huy Ha, Arhan Jain, Abraham Lee, Youngwoon Lee, Marius Memmel, Sungjae Park, Ilija Radosavovic, Kaiyuan Wang, Albert Zhan, Kevin Black, Cheng Chi, Kyle Beltran Hatch, Shan Lin, Jingpei Lu, Jean Mercat, Abdul Rehman, Pannag R Sanketi, Archit Sharma, Cody Simpson, Quan Vuong, Homer Rich Walke, Blake Wulfe, Ted Xiao, Jonathan Heewon Yang, Arefeh Yavary, Tony Z. Zhao, Christopher Agia, Rohan Bajjal, Mateo Guaman Castro, Daphne Chen, Qiuyu Chen, Trinity Chung, Jaimyn Drake, Ethan Paul Foster, Jensen Gao, Vitor Guizilini, David Antonio Herrera, Minho Heo, Kyle Hsu, Jiaheng Hu, Muhammad Zubair Irshad, Donovan Jackson, Charlotte Le, Yunshuang Li, Kevin Lin, Roy Lin, Zehan Ma, Abhiram Maddukuri, Suvir Mirchandani, Daniel Morton, Tony Nguyen, Abigail O’Neill, Rosario Scalise, Derick Seale, Victor Son, Stephen Tian, Emi Tran, Andrew E. Wang, Yilin Wu, Annie Xie, Jingyun Yang, Patrick Yin, Yunchu Zhang, Osbert Bastani, Glen Berseth, Jeannette Bohg, Ken Goldberg, Abhinav Gupta, Abhishek Gupta, Dinesh Jayaraman, Joseph J Lim, Jitendra Malik, Roberto Martín-Martín, Subramanian Ramamoorthy, Dorsa Sadigh, Shuran Song, Jiajun Wu, Michael C. Yip, Yuke Zhu, Thomas Kollar, Sergey Levine, and Chelsea Finn. Droid: A large-scale in-the-wild robot manipulation dataset, 2025. URL <https://arxiv.org/abs/2403.12945>.
- [5] Open X-Embodiment Collaboration. Open X-embodiment: Robotic learning datasets and RT-X models. In IEEE International Conference on Robotics and Automation (ICRA), 2024.
- [6] Homer Walke, Kevin Black, Abraham Lee, Moo Jin Kim, Max Du, Chongyi Zheng, Tony Zhao, Philippe Hansen-Estruch, Quan Vuong, Andre He, Vivek Myers, Kuan Fang, Chelsea Finn, and Sergey Levine. BridgeData V2: A dataset for robot learning at scale. In Conference on Robot Learning (CoRL), 2023.
- [7] Qingwen Bu, Jisong Cai, Li Chen, Xiuqi Cui, Yan Ding, Siyuan Feng, Xindong He, Xu Huang, et al. Agibot world colosseum: A large-scale manipulation platform for scalable and intelligent embodied systems. In 2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2025.
- [8] Haoquan Fang, Jiafei Duan, Donovan Clay, Sam Wang, Shuo Liu, Weikai Huang, Xiang Fan, Weichuan Tsai, Shirui Chen, Yi Ru Wang, Shanli Xing, Jaemin Cho, Jae Sung Park, Ainaz Eftekhari,

- Peter Sushko, Karen Farley, Angad Wadhwa, Cole Harrison, Winson Han, Ying-Chun Lee, Eli VanderBilt, Rose Hendrix, Suveen Ellawela, Lucas Ngoo, Joyce Chai, Zhongzheng Ren, Ali Farhadi, Dieter Fox, and Ranjay Krishna. Molmoact2: Action reasoning models for real-world deployment, 2026. URL <https://arxiv.org/abs/2605.02881>.
- [9] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 44(10-11):1684–1704, 2025.
- [10] Jose Barreiros, Andrew Beaulieu, Aditya Bhat, Rick Cory, Eric Cousineau, Hongkai Dai, Ching-Hsin Fang, Kunimatsu Hashimoto, Muhammad Zubair Irshad, Masha Itkina, et al. A careful examination of large behavior models for multitask dexterous manipulation. *arXiv preprint arXiv:2507.05331*, 2025.
- [11] Johan Bjorck, Abhinav Prasad, Aleksei Grigoriev, Feiyu Xia, Peng Ding, Zhengyi Luo, et al. GROOT N1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.
- [12] Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael 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, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0 : A vision-language-action flow model for general robot control, 2026. URL <https://arxiv.org/abs/2410.24164>.
- [13] Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael 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, 2025. URL <https://arxiv.org/abs/2504.16054>.
- [14] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint arXiv:2212.06817*, 2022.
- [15] Kevin Black, Allen Z. Ren, Michael Equi, and Sergey Levine. Training-time action conditioning for efficient real-time chunking, 2025. URL <https://arxiv.org/abs/2512.05964>.
- [16] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021. URL <https://arxiv.org/abs/2103.00020>.
- [17] Oriane Siméoni, Huy V. Vo, Maximilian Seitzer, Federico Baldassarre, Maxime Oquab, Cijo Jose, Vasil Khalidov, Marc Szafraniec, Seungeun Yi, Michaël Ramamonjisoa, Francisco Massa, Daniel Haziza, Luca Wehrstedt, Jianyuan Wang, Timothée Darcet, Théo Moutakanni, Leonel Sentana, Claire Roberts, Andrea Vedaldi, Jamie Tolan, John Brandt, Camille Couprie, Julien Mairal, Hervé Jégou, Patrick Labatut, and Piotr Bojanowski. Dinov3, 2025. URL <https://arxiv.org/abs/2508.10104>.
- [18] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL <https://arxiv.org/abs/1711.05101>.

- [19] Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, et al. Gemma 3 technical report. [arXiv preprint arXiv:2503.19786](https://arxiv.org/abs/2503.19786), 4, 2025.
- [20] Danny Driess, Jost Tobias Springenberg, Brian Ichter, Lili Yu, Adrian Li-Bell, Karl Pertsch, Allen Z. Ren, Homer Walke, Quan Vuong, Lucy Xiaoyang Shi, and Sergey Levine. Knowledge insulating vision-language-action models: Train fast, run fast, generalize better, 2025. URL <https://arxiv.org/abs/2505.23705>.
- [21] Karl Pertsch, Kyle Stachowicz, Brian Ichter, Danny Driess, Suraj Nair, Quan Vuong, Oier Mees, Chelsea Finn, and Sergey Levine. Fast: Efficient action tokenization for vision-language-action models, 2025. URL <https://arxiv.org/abs/2501.09747>.
- [22] Alex Henry, Prudhvi Raj Dachapally, Shubham Pawar, and Yuxuan Chen. Query-key normalization for transformers, 2020. URL <https://arxiv.org/abs/2010.04245>.
- [23] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag 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, 2024. URL <https://arxiv.org/abs/2406.09246>.
- [24] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.
- [25] Philipp Wu, Yide Shentu, Zhongke Yi, Xingyu Lin, and Pieter Abbeel. Gello: A general, low-cost, and intuitive teleoperation framework for robot manipulators, 2024. URL <https://arxiv.org/abs/2309.13037>.
- [26] Younghyo Park, Jagdeep Singh Bhatia, Lars Lien Ankile, and Pulkit Agrawal. Dexhub and dart: Towards internet scale robot data collection. *ArXiv*, abs/2411.02214, 2024. URL <https://api.semanticscholar.org/CorpusID:273821640>.
- [27] Yajvan Ravan, Adam Rashid, Alan Yu, Kai McClennen, Gio Huh, Kevin Yang, Zhutian Yang, Qinxi Yu, Xiaolong Wang, Phillip Isola, and Ge Yang. Lucid-xr: An extended-reality data engine for robotic manipulation, 2026. URL <https://arxiv.org/abs/2605.00244>.
- [28] Xinkai Jiang, Qihao Yuan, Enes Ulas Dincer, Hongyi Zhou, Ge Li, Xueyin Li, Julius Haag, Nicolas Schreiber, Kailai Li, Gerhard Neumann, and Rudolf Lioutikov. Iris: An immersive robot interaction system, 2025. URL <https://arxiv.org/abs/2502.03297>.
- [29] Hao-Shu Fang, Hongjie Fang, Zhenyu Tang, Jirong Liu, Junbo Wang, Haoyi Zhu, and Cewu Lu. RH20T: A comprehensive robotic dataset for learning diverse skills in one-shot. *IEEE Robotics and Automation Letters*, 2024.
- [30] Tony Z. Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware, 2023. URL <https://arxiv.org/abs/2304.13705>.
- [31] Zipeng Fu, Tony Z. Zhao, and Chelsea Finn. Mobile aloha: Learning bimanual mobile manipulation with low-cost whole-body teleoperation, 2024. URL <https://arxiv.org/abs/2401.02117>.
- [32] Tony Z. Zhao, Jonathan Tompson, Danny Driess, Pete Florence, Kamyar Ghasemipour, Chelsea Finn, and Ayzaan Wahid. Aloha unleashed: A simple recipe for robot dexterity, 2024. URL <https://arxiv.org/abs/2410.13126>.

- [33] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In International conference on machine learning, pages 8748–8763. PmLR, 2021.
- [34] Chi-Lam Cheang, Guangzeng Chen, Ya Jing, Tao Kong, Hang Li, Yifeng Li, Yuxiao Liu, Hongtao Wu, Jiafeng Xu, Yichu Yang, et al. Gr-2: A generative video-language-action model with web-scale knowledge for robot manipulation. arXiv preprint arXiv:2410.06158, 2024.
- [35] I2RT Robotics. i2rt: Python API for I2RT Robots. <https://github.com/i2rt-robotics/i2rt>, 2025. Accessed: 2026-04-20.
- [36] The ZeroMQ authors. ZeroMQ: An open-source universal messaging library. <https://zeromq.org>, 2007. Accessed: 2026-05-31.
- [37] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. Science Robotics, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>.
- [38] Tony Z Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. arXiv preprint arXiv:2304.13705, 2023.
- [39] Jason Jingzhou Liu, Yulong Li, Kenneth Shaw, Tony Tao, Ruslan Salakhutdinov, and Deepak Pathak. Factr: Force-attending curriculum training for contact-rich policy learning, 2025. URL <https://arxiv.org/abs/2502.17432>.
- [40] Steven Oh, Jason Jingzhou Liu, Tony Tao, Philip Han, Kenneth Shaw, Satoshi Funabashi, Ruslan Salakhutdinov, and Deepak Pathak. Factr 2: Learning force sensing and force-aware policies for any robot arm. 2026.
- [41] Kevin Zakka. Mink: Python inverse kinematics based on MuJoCo, February 2026. URL <https://github.com/kevinzakka/mink>.
- [42] Qianzhong Chen, Justin Yu, Mac Schwager, Pieter Abbeel, Yide Shentu, and Philipp Wu. SARM: Stage-aware reward modeling for long horizon robot manipulation. In International Conference on Learning Representations (ICLR), 2026.

Appendices

A	ABC-130K Task Taxonomy	18
B	Training & Model Details	19
B.1	Data Loading	19
B.2	Architecture Details	21
B.3	Training Details	22
B.4	Architecture Sizes and FLOPS	22
C	Real Robot Setup	22
C.1	Hardware	22
C.2	Real Robot Code	23
D	Inference Optimization	23
E	Single-task Finetuning	25
E.1	Task description and evaluation	25
E.2	Finetuning recipe	25
F	Dagger Infrastructure	25
G	Evaluation Details	26
H	Policy Conditioning	27
H.1	Operator-ID conditioning	27
H.2	Action prefix conditioning	28
H.3	Subtask conditioning	28

A ABC-130K Task Taxonomy

We organize the 195 tasks in ABC-130K into the following 7 primitive categories with a goal to group the tasks with similar contact modes and precision requirements together.

1. **Pick-and-Place (67 tasks, 793 h)**. Simple transfers: place cup by coaster, place snacks into paper bag and multi-step variants involving packing, organizing, or bussing: pack a student bag, place the mixed dishes into the plastic bin. Tasks range from single-object grasps to sequences requiring 5–10 objects placed in a target arrangement.
2. **Fine Pick-and-Place (39 tasks, 736 h)**. Millimeter-level precision tasks: for example, inserting credit cards into holders, building block towers, and pinning/removing brooches, phone case installation, and inserting photos into frames.
3. **Folding (36 tasks, 883 h)**. Deformable cloth manipulation spanning garment types: t-shirts, long-sleeve shirts, shorts, trousers, skirts, tank tops, towels, as well as non-garment deformables such as paper boxes and paper planes.
4. **Insertion/Ejection (19 tasks, 441 h)**. Peg-in-hole-style skills generalized to everyday objects: plugs into sockets, pens into caps, batteries into remote controls, earbuds into charging cases, and keys onto keyrings and caps onto bottles.
5. **Tool Use (16 tasks, 321 h)**. Tasks requiring an intermediate tool or mechanism: zipping jackets, sealing zip-top bags, locking and unlocking with keys, and sweeping with a dustpan.
6. **Sorting (8 tasks, 205 h)**. Multi-object classification tasks: sorting LEGOs by color, pills into containers, screws and nuts by type, and chemistry lab equipment.



Figure 16: **Representative top-camera frames for each task category.** **Pick-and-Place** (row 1, all 5): *Packing luggage, Place fruits in bag, Organize toys on shelf, Place snacks in bag, Place glasses in tray.* **Fine Pick-and-Place** (row 2, all 5): *Build block tower, Load dishrack, Place coffee filter, Insert credit cards, Set up chess pieces.* **Folding** (row 3, cells 1–4): *Fold paper box, Fold skirt pile, Fold t-shirt pile, Roll up utensils.* **Insertion/Ejection** (row 3 cell 5, row 4 cell 1): *Attach microphone, Cap pens.* **Tool Use** (row 4, cells 2–3): *Lock with key, Seal zip-top bag.* **Sorting** (row 4 cell 4): *Organize lab equipment.* **Untying/Tying** (row 4 cell 5): *Tie flower bouquet.*

- 7. Untying/Tying (10 tasks, 175 h).** Knot-tying on plastic bags, shoe-lacing, tying bouquets of flowers, and cable/headphone wrapping and untangling—long, thin deformables requiring bimanual coordination and contact-rich manipulation.

Examples of top-camera frames from each task category are provided in Figure 16. Task names in the dataset are often minor grammatical variants of each other. For example, *sorting legos into containers* and *sort the legos into the containers* are two distinct task descriptions in the dataset that have the same qualitative behavior. We adjust for this duplication in the our provided statistics by counting such variants as one task.

B Training & Model Details

This appendix provides the implementation details for reproducing our results. We first describe `abcd1`, our custom dataloader, together with the MP4-plus-binary episode format used to load thousands of hours of video efficiently in distributed training. We then provide the training and architecture details for ABC-DiT and ABC-VLA. Finally, we report the model sizes and FLOP counts. All models are trained on NVIDIA H200 GPUs.

B.1 Data Loading

We develop a custom dataloader, `abcd1` (A Behavior Cloning Dataloader) to facilitate fast loading of training data in our pipeline. Many data loading pipelines do not scale to loading thousands of hours of data in a distributed training setup; hence part of making ABC-130K useful is releasing a dataloader along with our dataset that can facilitate distributed dataloading at such a scale.

The training data format consists of two files per episode: 1) an MP4 containing the concatenated

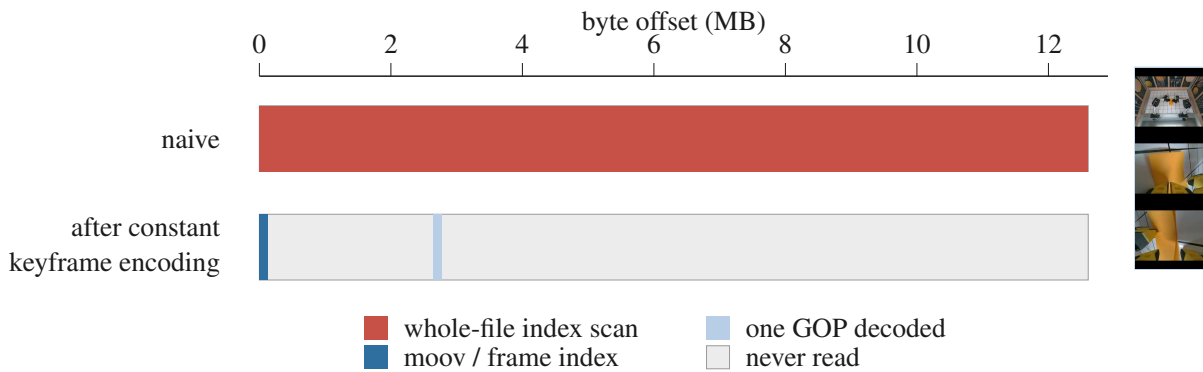


Figure 17: **Efficient frame access for fast data loading.** Encoding keyframes more frequently and in a manner that allow for an analytically reconstructed frame index make random frame access nearly free. To read one frame from a video, `torchcodec`'s default scans the entire file to build its frame index (top). Correctly encoding the file allows us to compute the index analytically, meaning we only need to read the file header plus frames since the closest keyframe, leaving the rest of the file untouched, reducing disk pressure. The decoded frame is three vertically stacked camera views.

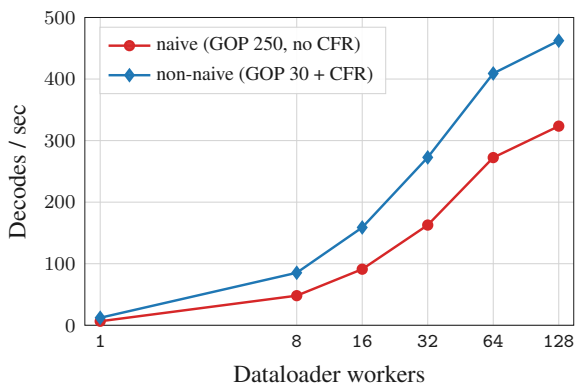


Figure 18: **Random-frame decode throughput.** Decodes per second versus dataloader workers on a local filesystem. Fixing the encoding options and dataloader args improves throughput.

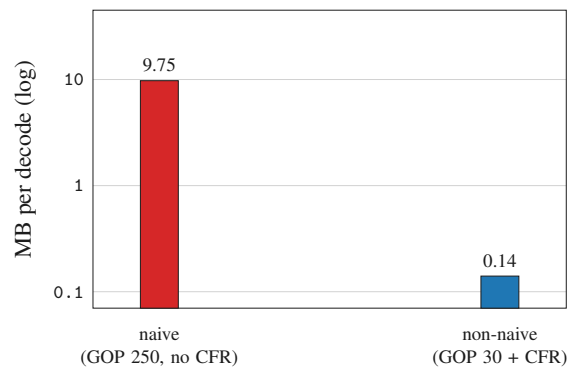


Figure 19: **Per-decode read volume.** Mean MB requested from the filesystem per decode (log scale). CFR frame mapping cuts per-decode reads by $\sim 70\times$ ($9.75\text{MB} \rightarrow 0.14\text{MB}$) on average, reducing wall-time on bandwidth-constrained dataloading.

camera views, 2) and a binary file containing the states and actions. Our data loading philosophy is to keep our data in an extremely simple format while still facilitating extremely fast on the fly loading of data. From this perspective, MP4 provides great advantages as it is readable by all manner of video players, IDEs, etc., for convenient viewing without special software, while still being a highly compressed format and while providing many format options to facilitate fast dataloading. Our system facilitates either loading quickly from a locally mounted file system or streaming data from cloud object storage such as s3, simply by changing the backend. In the case of local data loading, files are sharded by episode and downloaded onto individual nodes. The videos are encoded using the H.264 format. We use several format options to make decoding easy:

- `+faststart` moves the MP4 `moov` atom to the front of the file meaning that the whole file doesn't need to be read before decoding can begin.
- We disable MP4 B-frames so that frames only depend on the most recent keyframe, and not on frames after them.

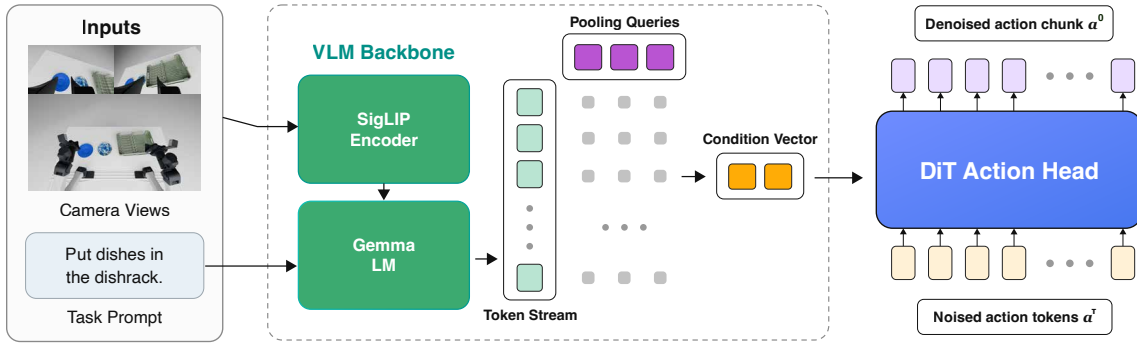


Figure 20: **ABC-VLA pooled adaLN architecture.** ABC-VLA uses a Gemma 3 VLM backbone, attention-pools the final VLM hidden states into a compact feature vector, and projects the result to an adaLN conditioning vector for the lightweight DiT action head.

- Crucially, we encode the video with constant GOP (Group of Pictures, i.e. a keyframe + following frames compressed relative to it). This means that the keyframes in the video are in deterministic positions.

In practice, we encode videos with $\text{GOP}=30$, corresponding to one keyframe per second for 30-FPS video. A fixed GOP and constant ticks per frame allow us to reconstruct the file index. The reconstructed index is passed to `torchcodec`'s `VideoDecoder` object in order to facilitate fast loading (see Figure 17). We note that all of these speedups are achieved by using existing APIs in pure Python, keeping our system readable and maintainable. For remote data loading, we make a custom Python file-like object which facilitates reading bytes from the given remote storage source (e.g. `s3` or `GCS`), and is passed to `torchcodec`.

To handle multiple camera streams, we simply stack them into a single MP4 file. This cuts down the number of requests to have multiple camera streams by a factor of 3 (assuming 3 cameras, more in the general case). This is important especially when streaming from a remote data source. We also support loading multiple consecutive frames; this can be done at minimal bandwidth / CPU decoding cost as the keyframe for subsequent frames is more likely than not already loaded.

While providing speed at the low level, our dataloading infrastructure facilitates the ability for practitioners to freely mix datasets from different sources and weight them independently. This is an essential feature for many experiments – for example for experimenting with different slices of pretraining mixes at the pretraining scale, or for experiments with intervention data at the single task scale.

B.2 Architecture Details

ABC-DiT, the DINOv3-xattn version in Table 1, uses DINOv3 [17] ViT-B. We use 12 query tokens per image and share the weights of the DINO encoder across all three images. We use a modified DiT architecture with 32 layers, a hidden size of 1536, 24 heads, and an MLP ratio of 4.

ABC-VLA, the Pooled adaLN version in Table 1, uses Gemma 3 [19] 4B backbone. The images are fed into the context through the SigLIP encoder. The robot proprioception is fed through a two-layer MLP with hidden size 256 to produce a 2560-dimensional vector that is fed into the token stream, followed by the task prompt and a `<action_start>` token. We use 8 learnable query tokens to attention-pool the tokens of the last layer of the Gemma VLM. This produces 8 512-dimensional vectors that are flattened. The proprioception is also linearly projected to a 512-dimensional embedding. The 4096-dimensional vector corresponding to the VLM features is concatenated with this proprioception embedding as well as a 512-dimensional embedding of the diffusion timestep. This combined 5120-dimensional vector is then fed through a small MLP to be projected down to a 512-dimensional vector and is used for adaLN modulation in the diffusion head. The diffusion head is relatively small with 8 layers, hidden size 512, 8 heads, and an MLP ratio of 4.

Model	Parameters		Training TFLOPs / sample		
	Backbone	Action head	Backbone	Action head	Total
ABC-DiT	85.7M	1.93B	0.329	0.349	0.678
ABC-VLA	4.3B	44.7M	6.957	0.063	7.020

Table 2: **Parameter and compute split for ABC-DiT and ABC-VLA.** TFLOPs are training Tera-FLOPs per sample, using a $3\times$ forward-pass estimate for forward plus backward and counting one multiply-add as two FLOPs. ABC-VLA uses eight independent diffusion noise/timestep draws per sample. ABC-DiT uses a single diffusion draw per sample because extra draws would repeatedly evaluate its large DiT action head; in ABC-VLA, by contrast, the expensive VLM pass is shared across draws and only the small action head is repeated.

B.3 Training Details

For ABC-DiT, we linearly warmup the learning rate for the first 1000 iterations to 1×10^{-4} and then keep it constant throughout training. We use the AdamW optimizer [18] with a weight decay of 0.01. We use a learning rate scale of 0.1 for the vision encoder. We use gradient clipping with a maximum gradient norm of 10. We train on 12 H200 nodes with a batch size of 96 per GPU bringing the global batch size to 9216. We dropout the input proprioception with a probability of 0.1.

For ABC-VLA, we linearly warm up the learning rate for the first 1000 steps to 1×10^{-4} and keep it constant throughout training. We use the AdamW optimizer [18] with weight decay 0.01. We use gradient clipping with max gradient norm 10 and maximum gradient value 100. The model is trained with bf16 mixed precision. We train on 8 H200 nodes with batch size 24 per GPU and gradient accumulation 6, bringing the global batch size to 9216. We dropout the input proprioception with probability 0.1.

B.4 Architecture Sizes and FLOPS

Our choices of model size are relatively nonstandard: for ABC-DiT, we use a large action head while keeping the DINOv3 visual backbone comparatively small. This is because the visual backbone is parameter-efficient but compute-intensive: it attends over image patches for three camera views, so its FLOP cost is large relative to its parameter count. This makes it comparatively cheap to add parameters in the action head. Empirically, Figure 21 shows that scaling the size of the DiT head is efficient in reducing training loss in terms of both training steps and FLOPs.

Conversely, ABC-VLA places most of its parameters and FLOPs in the Gemma-SigLIP backbone, while its diffusion action head is small. Scaling up the diffusion head in this case would appear to be less effective than simply doing the multiple diffusion draws trick covered in the main paper (i.e. taking multiple diffusion noise + timestep samples and averaging the gradients into the backbone) as the bottleneck is the variance of the diffusion gradients into the VLM rather than the absolute number of parameters. Table 2 summarizes this split.

C Real Robot Setup

Reproducible robot learning requires a platform that is both capable and easy to replicate. We build ABC around a low-cost bimanual YAM workstation and a lightweight robot codebase for data collection, evaluation, and deployment.

C.1 Hardware

All data collection and evaluation are conducted on a bimanual platform comprising two *I2RT Yam* arms [35]: openly available, low-cost 6-DoF manipulators. The two arms are mounted parallel to each other on a table and enclosed on three sides by white walls; the enclosure narrows background diversity in the training distribution, but (i) reduces variance in evaluation from incidental visual disturbances and (ii) isolates fine-manipulation learning from the confound of background generalization. We note, however, that despite using only data collected in this caged setup, many of our policies transfer outside the cage and can be deployed in some in-the-wild settings. We use three Intel RealSense D405 cameras,

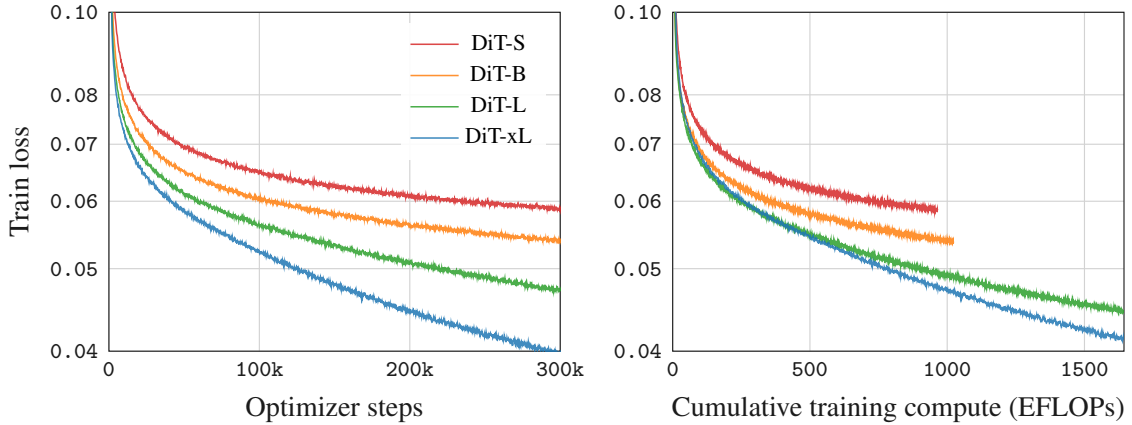


Figure 21: **ABC-DiT model-size scaling.** Train loss versus optimizer steps (left) and versus cumulative training compute (right) for four ABC-DiT sizes trained with identical hyperparameters and global batch size 9,216. At a fixed compute or step budget the larger DiTs reach lower train loss. The total number of parameters in S/B/L/xL are 153M, 290M, 746M, and 1.93B respectively.

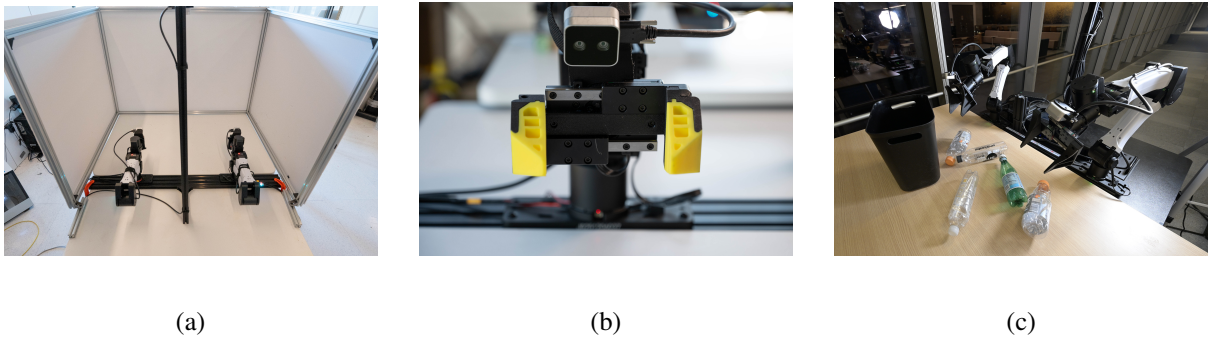


Figure 22: **Hardware setup.** (a) Bimanual YAM workstation with two 6-DoF arms, three RealSense D405 cameras, and white enclosure walls. (b) The FlexPoint gripper used in a subset of our dataset. (c) Policies using our dataset can be deployed in in-the-wild settings.

streaming at 30Hz, with one being mounted to give a third-person view above the workstation while the other two are mounted as wrist cameras.

C.2 Real Robot Code

We implement a custom framework to communicate with our robot, based on the ZMQ backend [36], which will also be released. Similar to ROS [37], it uses a decentralized node system that relies on PUB/SUB communications. It is meant to be a very lightweight library that is easily extensible to new hardware. Each module of the infrastructure (e.g. GELLO leader arm, YAM follower arm, Realsense cameras, etc.) has its own dedicated node that ticks at a predefined rate and can subscribe and publish to different topics. In order to maintain the tick rate, each node uses Python’s `time.sleep` to coarsely wait for the desired period, and for the final 300 microseconds it employs a busy-spin loop to improve timing precision.

D Inference Optimization

We implement various optimizations to reduce the inference latency of both ABC-DiT and ABC-VLA. For the purposes of this section, all timing numbers are measured on an NVIDIA GeForce RTX 5090. The main techniques include conducting inference at bf16, caching the visual embeddings/VLM features to avoid recomputation every diffusion step, and leveraging `torch.compile`. While the first two are fairly ubiquitous, `torch.compile` performance can vary depending on how it is used. When wrapping the model in a compile block, we pass in `fullgraph=True`. This causes compilation to fail on graph

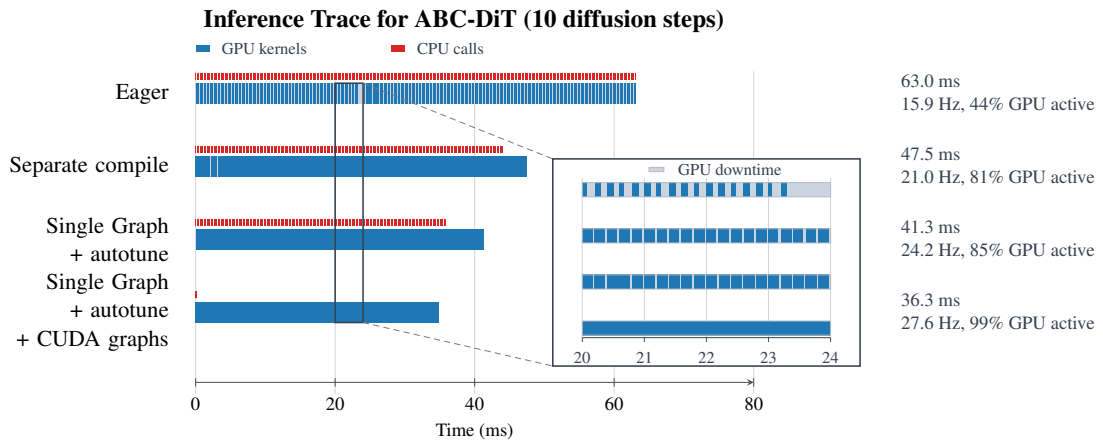


Figure 23: **Inference optimizations for ABC-DiT.** Profiling traces for ABC-DiT inference with 10 diffusion denoising steps. All timings are measured on an NVIDIA GeForce RTX 5090.

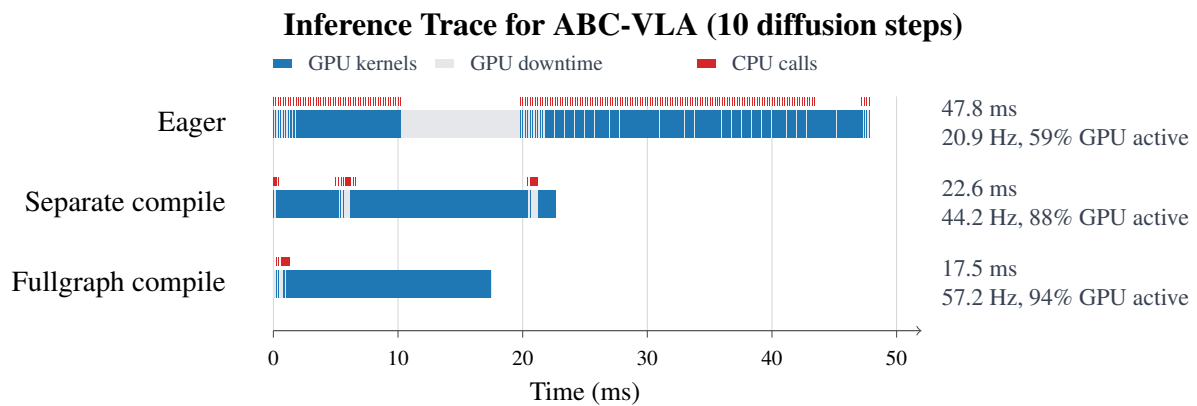


Figure 24: **Inference optimizations for ABC-VLA.** Profiling traces for ABC-VLA inference with 10 diffusion denoising steps. All timings are measured on an NVIDIA GeForce RTX 5090.

breaks, including host synchronization points, which lets us identify and modify code paths where there are CPU-side dependencies that limit compiler optimization. We use this to iteratively go through our model and remove any synchronization points until the entire inference pass compiles as a single graph. We show how the various optimizations reduce latency for ABC-DiT in Figure 23 and for ABC-VLA in Figure 24.

For ABC-DiT, our baseline (eager mode) uses bf16 and cached visual features, but no PyTorch compilation. Looking at the trace, we can see there are many CPU-side kernel launch calls throughout the inference duration of the model, as well as the fact that there are many small GPU kernels – each one carrying its own launch overhead. The first optimization involves just compiling the DINO model and the DiT head separately. Compilation reduces the latency by fusing multiple kernels together. Kernels being fused together mean that there are fewer that need to be individually launched hence lower CPU overhead. The next optimization on top of that is to compile both the DINO encoder and DiT head together instead of creating two separate graphs, and using kernel autotuning. This lets the compiler benchmark various kernel implementations and launch configurations for the actual tensor shapes on the target GPU. The last optimization does all of this with CUDA graph capture. This is a technique whereby memory locations are kept static, and during the initial graph capture phase, PyTorch records all of the kernels launched by the model. Subsequent operations replay those kernels rather than having the CPU launch each kernel separately. This dramatically reduces overhead which can be seen by how the number of CPU-side kernel launch calls diminishes. In the end, these optimizations improve latency from 63 ms down to 36.3 ms.

For ABC-VLA, the eager baseline also uses bf16 with cached embeddings but exhibits a major gap due to GPU downtime. The separate compile involves compiling the SigLIP, VLM, and DiT head separately with max autotuning and CUDA graph capture. Here, the main overhead is seen between these three main modules. When the full inference path is compiled together, much of the overhead is then eliminated. Most of the remaining CPU-side overhead occurs at the beginning likely due to input staging and graph setup. These optimizations combined bring down the latency from 47.8 ms down to 17.5 ms. One perhaps counterintuitive aspect of the inference results is that ABC-VLA is faster than ABC-DiT despite having more than twice as many total parameters. This is because the repeated component during diffusion is much smaller for ABC-VLA given the AdaLN design choice. In ABC-DiT, the diffusion head has roughly 1.93B parameters and must be evaluated at every diffusion step. In ABC-VLA, the VLM is large, but it is only run once to produce cached features; the diffusion head itself has only 45M parameters and is then run for the 10 diffusion steps. As a result, the cost of ABC-VLA inference is dominated by a single VLM forward pass plus lightweight diffusion-head evaluations, which is faster than running a 1.5B-parameter transformer 10 times in ABC-DiT.

E Single-task Finetuning

E.1 Task description and evaluation

We provide here detailed task description, evaluation protocol, and finetuning recipe for the downstream experiments in Section 5.2. We provide the task names and description for the tasks we used for finetuning.

1. **Take credit cards out of the card holder:** The task includes opening up a wallet, scrolling across it to find a filled slot, taking out the card from one, placing it in the container and continuing. This task requires high precision and fine motor skills.
2. **Sort legos into the container:** The task includes picking up lego pieces, breaking them up into atomic pieces if they are stuck together, and then placing them into the correctly colored container. This task requires strong visual representations for planning.
3. **Insert pens into pen caps:** The task is to pick up the pen and the cap with different arms and then insert the pen into the cap with the correct orientation and then press hard.
4. **Unscrew bottle caps:** The task is to pick up a bottle and open up its cap using the other arm.

We provide a time budget of 2 minutes for each task, except for credit card removal, for which we used a budget of 3 minutes. Table 3 provides the evaluation rubric we used for these tasks.

E.2 Finetuning recipe

ABC-DiT is cheaper to train and therefore is our base model for single-task finetuning except for the credit-card removal task. We find that performance for ABC-VLA finetuning was markedly better for this task compared to that of ABC-DiT. We finetune both model families with AdamW using a learning rate of $1e-5$ for both the vision backbone and the action expert, with weight decay $1e-2$. We use a batch size of 1,440 for ABC-DiT and 192 for ABC-VLA using 1 H200 node for training. We keep all other training hyperparameters the same as in pretraining. We train for 200,000 gradient steps and visually inspect checkpoints every 50,000-steps, selecting the best checkpoint for evaluation.

F DAgger Infrastructure

Most robot teleoperation data, including the data in this paper, is collected using leader arms [25, 38]. While active leader arms exist [39, 40], they tend to be more complex to set up, requiring system identification and advanced control algorithms to make them smooth. Meanwhile, alternative methods like VR require additional hardware setups separate from normal teleoperation infra. We release a new method to collect intervention data using no additional hardware for Passive Leader Intervention. The way this works is that when an intervention happens, we run forward kinematics on the pose of the leader and follower arms, and continuously on the leader arms during the course of the intervention. We then take

the delta in SE3 pose between the intervention pose and the current pose of the leader arms. We then compute the pose made up of the pose of the follower end effector + the leader SE3 delta for the follower arms, and then use IK to get follower arm joints. We use mink for IK [41]. The infrastructure to do such interventions will be released along with the rest of the code for this project.

We implement a state machine for data collection to facilitate collection of data. When an episode starts, the policy is executing. The operator can press a button (either on the leader arms if they have such a button, or a cheap commercially available foot pedal) to transition to the teleoperation mode. At any time the operator can press another button to transition back to the policy.

When transitioning from an intervention back to the policy, we condition the policy on the last few actions executed during the intervention as part of the RTC prefix. This not only makes the transition smoother, but makes it easier for the operator to select a mode of the diffusion policy if it is not obvious from the Markovian observations of the policy what the action should be (e.g. whether to retry, say, folding a flap or go to the next action during box folding).

G Evaluation Details

When we report the progress of policies on various tasks, we use the rubric in Table 3. Each score is normalized by the maximum score in order to give a measure of progress.

Table 3: **Evaluation task rubrics.** *Max Score* is the maximum achievable score for the task; reported progress is the achieved score normalized by this maximum. *Timeout* is the maximum trial duration.

Task	Setting	Max Score	Timeout	Scoring Rubric
throw plastic bottles in bin	6 bottles, 1 bin	6	120 s	+1 for each bottle inside the bin
insert pens into the pen caps	1 pen, 1 cap	3	120 s	+1 for picking up both pen and cap; +2 for inserting the pen into the cap
load plates into tabletop dish rack	2 plates, 1 dish rack	6	180 s	Per plate: +1 for picking up the plate; +1 for placing it inside the dish rack; +1 for placing it correctly
sorting legos into containers	2 green/yellow bricks, 2 bins (with bricks in them)	4	120 s	+1 for each properly deposited lego piece
unscrew bottle caps	1 bottle	3	120 s	+1 for lifting the bottle, +2 for unscrewing the cap
take credit cards out of the card holder	1 wallet filled with credit cards and 1 container	5	180 s	+1 for opening wallet, +1 for removing card, +1 for placing in container
folding paper box	1 paper box	5	180 s	+1 for picking up box, +1 for folding each of the sides, +1 for closing the box lid, +1 for folding in the flaps
folding tshirt pile and stacking	1 t-shirt	5	390 s	+1 pick & place; +1 flatten; +1 for ≥ 1 proper fold; +1 full fold (roughly square); +1 placed in corner. Fold quality rated low/mid/high; on repeated flatten–fold cycles, score only the best fold (no double counting)

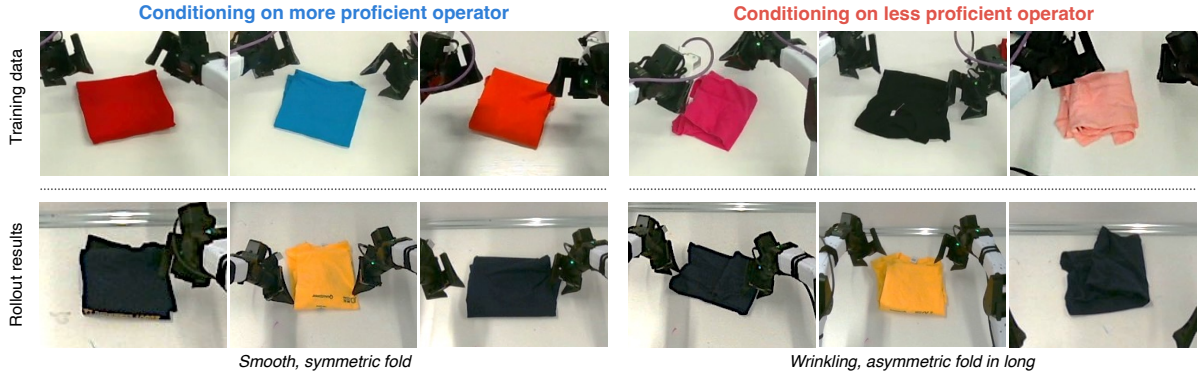


Figure 25: **Impact of Operator conditioning.** *Top row:* folds from training demonstrations contributed by Op-0 (more proficient operator) left and Op-1 (less proficient operator) right. *Bottom row:* representative folds from same trained policy under op-0 (left) and op-1 (right) conditioning. Although the policy is the same, varying only the operator prompt produces qualitatively distinct fold outcomes matching the training distribution of the conditioning operator. Quantitative results across all rollouts are reported in Table 4.

H Policy Conditioning

We study three mechanisms for shaping the behavior of our policies at inference time. Operator-ID conditioning (Section H.1) addresses stylistic multimodality across teleoperators, allowing the policy to be steered towards the chosen operator’s style at deployment. Action prefix conditioning (Section H.2) examines how the length of the executed-action prefix used by Real-Time Chunking trades off motion smoothness against visual responsiveness. Subtask conditioning (Section H.3) looks at resolving temporal ambiguity within a task via language prompting. Across all three, the common theme is shifting the policy’s effective conditioning balance toward the signals most relevant to the current decision.

All experiments in this section are evaluated on the t-shirt folding task. We chose it because it highlights all three conditioning mechanisms we study: substantial variation in operator quality (Section H.1); a multi-stage structure with visually similar states in trajectories, so that the correct next action is often ambiguous from a single observation (Section H.3); and a large pool of demonstrations (~ 268.64 hours total). Every t-shirt-folding episode is subtask-annotated, compared to 44% of ABC-130K overall, which makes it particularly suited to the subtask-conditioning study.

For these experiments we use the CLIP cross-attention variant of ABC-DiT rather than the DINOv3 backbone used elsewhere: once finetuned, the CLIP cross-attention policy was the most reliable on this task, so that the conditioning effects we report are not confounded by a weak policy. The base model is pretrained on the internal 7,000-hour corpus, then finetuned on a filtered subset of the released t-shirt-folding data: for each operator we keep only the later-collected half of their episodes, which tend to reflect more skillful execution as operators grow more practiced over the course of collection. We finetune for 100k steps and evaluate on two unseen shirts (small/blue and large/yellow) with five randomly crumpled initial configurations per shirt, each initialized at approximately the same workspace location ($n = 10$ per condition, maximum trial duration 390 s). Each rollout is scored on a 5-point rubric (Section G); fold quality is rated low / medium / high. All evaluation data, including the rollouts’ actions, states, and image observations, are released for reader inspection.

H.1 Operator-ID conditioning

The natural baseline for operator-ID conditioning is to filter the training data to the demonstrations of a single high-quality operator. We evaluate this baseline by finetuning the ABC-DiT model on the subset of the t-shirt folding corpus contributed by the highest-volume operator (op 0, 19.5 hours across 1,183 episodes). This filtered checkpoint achieves a mean rubric score of 3.3 with only 2 of 10 completions and a mean trial duration of 369 s (Table 4); it performs worse on every metric than the unconditioned all-operator baseline (3.8, 4/10, 302 s). We trained the filtered model to 30k steps rather than 100k, after

Table 4: **Operator-ID conditioning affects t-shirt folding behavior at inference time.** T-shirt folding evaluation across the operator-filtered baseline and operator-ID conditioning inference modes. Rubric scores are out of 5; mean trial duration is reported in minutes:seconds with timeouts counted as the maximum trial duration (390 s); quality counts (H/M/L) are among completed trials. All conditioning rows use the same trained checkpoint, varying only the inference-time prompt.

Training data	Inference prompt	Mean score	Completions	Mean time	H / M / L
<i>Baselines</i>					
All-operator corpus	task only	3.8	4 / 10	302 s	2 / 2 / 0
Op-0 filtered corpus	task only	3.3	2 / 10	369 s	1 / 0 / 1
<i>Operator-ID conditioning (same checkpoint)</i>					
All-operator corpus	task + op 0	4.6	8 / 10	237 s	4 / 3 / 1
All-operator corpus	task only (marginalize)	4.4	6 / 10	247 s	1 / 4 / 1
All-operator corpus	task + op 1	4.0	5 / 10	277 s	1 / 1 / 3

observing overfitting when trained longer.

Operator-ID conditioning allows test-time conditioning on desired behavior without data filtering. We finetune the ABC-DiT policy on the all-operator corpus with operator-ID text appended to the task prompt (see Figure 25 for representative training demonstrations from op 0 and op 1). We evaluate three inference conditions on the same checkpoint:

1. Conditioning on op 0, the highest-volume operator, whose training demonstrations are characterized by short, deliberate execution (mean episode duration 59 s)
2. Conditioning on the task prompt alone, which marginalizes over operators by reverting to the training-time dropout target
3. Conditioning on op 1, a long-duration operator with 226 episodes of training data whose demonstrations average 205 s per episode and exhibit lower fold quality

Marginalized inference improves over the unconditioned baseline, indicating that operator-ID conditioning at training time is strictly beneficial even when the operator channel is not used at inference. Conditioning on op 0 yields the highest-quality folds with the highest completion rate, while conditioning on op 1 both yields longer duration and lower-quality folds. We also note that op-0-conditioned rollouts frequently re-execute earlier subtasks, e.g., re-flattening a partially folded garment that fails some implicit quality threshold, consistent with stricter stage-transition criteria in that operator’s training demonstrations; op-1-conditioned rollouts advance through transitions more readily and reproduce that operator’s lower fold quality (Figure 25).

H.2 Action prefix conditioning

All of our policies are trained with training-time Real-Time Chunking [15], in which each generated action chunk is conditioned on a prefix of recently-executed actions. The action prefix improves inter-chunk motion smoothness by anchoring the next chunk to the trajectory currently being executed.

We observe that while this improves the smoothness of policies, it can also hurt their performance, and policies can tend to overfit to the prefix and partially ignore visual observations in favor of continuing the existing trajectory. Quantitative evidence of this is provided in Table 5, and a qualitative explanation of the phenomenon is illustrated in Figure 26. We find that the simplest method to combat this is reducing the prefix length at the cost of some smoothness; more thorough characterization into the issue and investigation of other methods to mitigate these effects are left to future work.

H.3 Subtask conditioning

Multi-stage manipulation tasks introduce a second form of multi-modality, orthogonal to the operator-stylistic variation in Section H.1. Depending on which stage of a task the policy is in, the same task

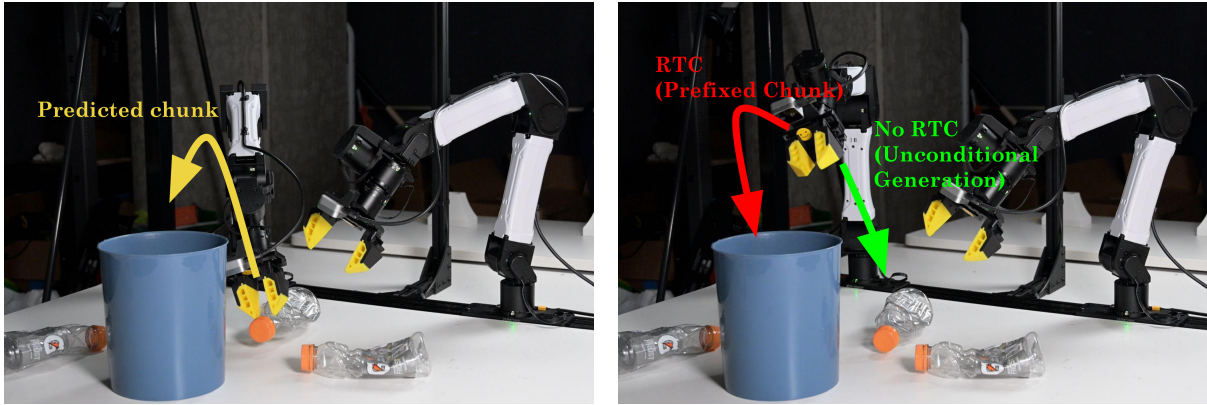


Figure 26: **Prefix conditioning can suppress visual responsiveness.** Prefixing chunk generation on recent actions can bias the generated motion toward the recently executed trajectory, while the unconditioned generation remains more responsive to the current visual observation. **Left:** In the first chunk, the robot fails to pick the bottle. **Right:** In the next chunk, we compare RTC prefix-conditioned and unconditional generations. In this example, the model conditioned on the prefix continues the trajectory into the bin, ignoring the visual observation that it has missed the grasp, while the unconditional generation, while less smooth, correctly attends to the visual observations.

Table 5: **Shorter action prefixes improve t-shirt folding performance.** Effect of inference-time action prefix length on t-shirt folding under op-0 conditioning, using the same trained checkpoint as the op-0 row in Table 4. Same evaluation protocol; only the prefix length differs. Mean trial duration is reported in seconds with timeouts counted as the maximum (390 s); quality counts (H/M/L) are among completed trials.

Action prefix	Mean score	Completions	Mean time	H / M / L
Prefix = 4	3.9	5 / 10	309 s	3 / 2 / 0
Prefix = 1	4.6	8 / 10	237 s	4 / 3 / 1

prompt corresponds to distinct action distributions. Task-prompt conditioning alone often cannot disambiguate these stages from a single visual observation without history, because intermediate and terminal states can look similar, while the correct next action depends on which stage the task has actually reached. For example, a crumpled tshirt and a completed fold could appear similar depending on the lighting, cloth texture, fold quality, etc., so a policy conditioned on the task prompt alone may regrasp an already-folded shirt, flatten it, and begin folding again—and sometimes begins re-flattening partway through a fold. Subtask conditioning addresses this by exposing the current subtask through language, using the same independent-dropout scheme as operator-ID conditioning; at inference the subtask can be set by a high-level planner, classifier, or human operator, letting the policy commit to the stage-appropriate action distribution rather than averaging over stage-conditional modes.

To evaluate subtask prompting, we train a small subtask classifier—whose architecture is inspired by the stage-aware reward model of SARM [42]—on ABC-130K, and use it to set the subtask prompt for our ABC-DiT policy at inference time. We initialize each trial with a half-folded shirt to probe the failure mode described above. Unlike the earlier rollouts, which ended once the shirt was placed in the corner, we hold the arms in the reset pose for 10 s after placement to observe the policy’s subsequent behavior. Without subtask prompting, the policy exhibits the re-flattening failure in 5 of 10 trials; in 4 of these cases, it regrabs an already-folded or already-placed t-shirt. With subtask prompting, the policy does not exhibit this behavior, although it fails to complete the task within the time limit in one trial due to a missed grasp and subsequent recovery errors.