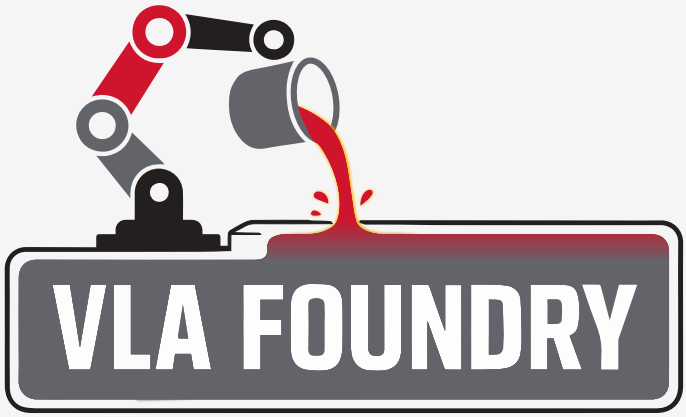
VLA Foundry: A Unified Framework for Training Vision-Language-Action Models

Jean Mercat[*†], **Sedrick Keh**[*†], **Kushal Arora**[†], **Isabella Huang**[†], **Paarth Shah**[†], **Haruki Nishimura**, **Shun Iwase**, **Katherine Liu**[†]

Toyota Research Institute
[*]Co-first authors, [†]Core contributors



We present VLA Foundry, an open-source framework that unifies LLM, VLM, and VLA training in a single codebase. Most open-source VLA efforts specialize on the action training stage, often stitching together incompatible pretraining pipelines. VLA Foundry instead provides a shared training stack with end-to-end control, from language pretraining to action-expert fine-tuning. VLA Foundry supports both from-scratch training and pretrained backbones from Hugging Face. To demonstrate the utility of our framework, we train and release two families of models: the first trained fully from scratch through our LLM→VLM→VLA pipeline and the second built on the pretrained Qwen3-VL [4] backbone. We evaluate closed-loop policy performance of both models on LBM Eval [66], an open-data, open-source simulator. We also contribute usability improvements to the simulator and the STEP [63] analysis tools for easier public use. In the nominal evaluation setting, our fully-open from-scratch model is on par with our prior closed-source work [65] and substituting in the Qwen3-VL backbone leads to a strong multi-task table top manipulation policy outperforming our baseline by a wide margin.

The VLA Foundry codebase is available at https://github.com/TRI-ML/vla_foundry and all multi-task model weights are released on https://huggingface.co/collections/TRI-ML/vla-foundry. Additional qualitative videos are available on the project website https://tri-ml.github.io/vla_foundry.

Date: April 20, 2026
Correspondence: Jean Mercat and Katherine Liu at firstname.lastname@tri.global

1 Introduction

Robotics foundation models are advancing at a rapid pace, with many systems [51, 52, 61, 32, 72, 36] demonstrating capabilities that would have seemed out of reach just a few years ago. As the frontier moves faster, the tooling required to support rigorous research must keep pace. Many high-impact questions – about data scaling, backbone pretraining, and the interplay between robotics and non-robotics data – require both scale (compute, data, etc.), as well as modular algorithmic infrastructure that allow users full control over different parts of the model and training pipeline. However, most existing codebases have either not been extensively tested at scale [21], or are largely focused on model releases [32, 50, 46] and therefore tightly coupled to specific algorithmic decisions, limiting research flexibility.

At the same time, data scarcity remains a fundamental bottleneck in robotics. Robot interaction data is severely constrained relative to data used for language and vision models, especially in diversity and in signal density per token [5]. As robot policies continue to scale, the relative importance of non-robotics data only grows [35]. Despite this data disparity, most open-source VLA frameworks focus narrowly on the action training stage, treating the upstream data recipe as fixed or out-of-scope. Such separation is problematic: data decisions made during LLM and VLM pretraining have direct consequences for downstream robotics performance. Exploring the design space requires a framework that treats the entire pipeline, from pretraining to policy learning, as a single, controllable system.

We developed **VLA Foundry** to address these challenges. VLA Foundry is a unified, open-source framework with a shared data-loading and training stack that spans LLM, VLM, and VLA training in a single codebase, giving practitioners control over the entire pipeline – from backbone pretraining to action-expert fine-tuning. Because every stage shares the same infrastructure, researchers can co-train across modalities, mix datasets, and prototype new architectures without stitching together disparate tools. The framework natively supports pretrained backbones from Hugging Face, and its modular, configuration-driven design lets users swap architectures, data pipelines, and training recipes through simple command-line or YAML changes.

VLA Foundry has the following key features:

- **Full pipeline controllability**, enabling researchers to intervene at any stage of the data and training recipe — from backbone pretraining to action expert fine-tuning — through a shared, configuration-driven interface.
- **Flexible multi-modal training** with probabilistic dataset mixing and dataloaders that support text, image-caption, and robotics data, allowing precise control over the training mixture at every stage.
- **Native Hugging Face integration** facilitates loading of pretrained vision encoder, LLM, and VLM backbones, making benchmarking new architectures straightforward within the same training and evaluation pipeline.
- **Scalable distributed training** built on FSDP2 and cloud-native tooling (AWS SageMaker, S3), supporting multi-node, multi-GPU runs with automatic gradient accumulation, mixed precision, and checkpoint synchronization.

2 Related Work

2.1 LLM/VLM Training Frameworks

Large Language Models (LLMs) form the foundation of modern multimodal systems, providing scalable sequence modeling capabilities, strong linguistic representations, and emergent reasoning abilities. Early work established the effectiveness of the scaling of transformer-based language models while subsequent efforts have largely focused on improving training efficiency, transparency, and reproducibility. Projects such as Megatron-LM [60], DeepSpeed [56], and GPT Neox [1] introduced distributed training strategies that enabled scaling to hundreds of billions of parameters. More recent and accessible open training initiatives, including OpenLM [23], Olmo [67], LLM360 [40], and its follow-up K2 model [39] emphasize full stack transparency by releasing training data, code, intermediate checkpoints, and logs. Complementary efforts such as FastLLM [58] provide practical recipes for training competitive models under more constrained compute budgets, while

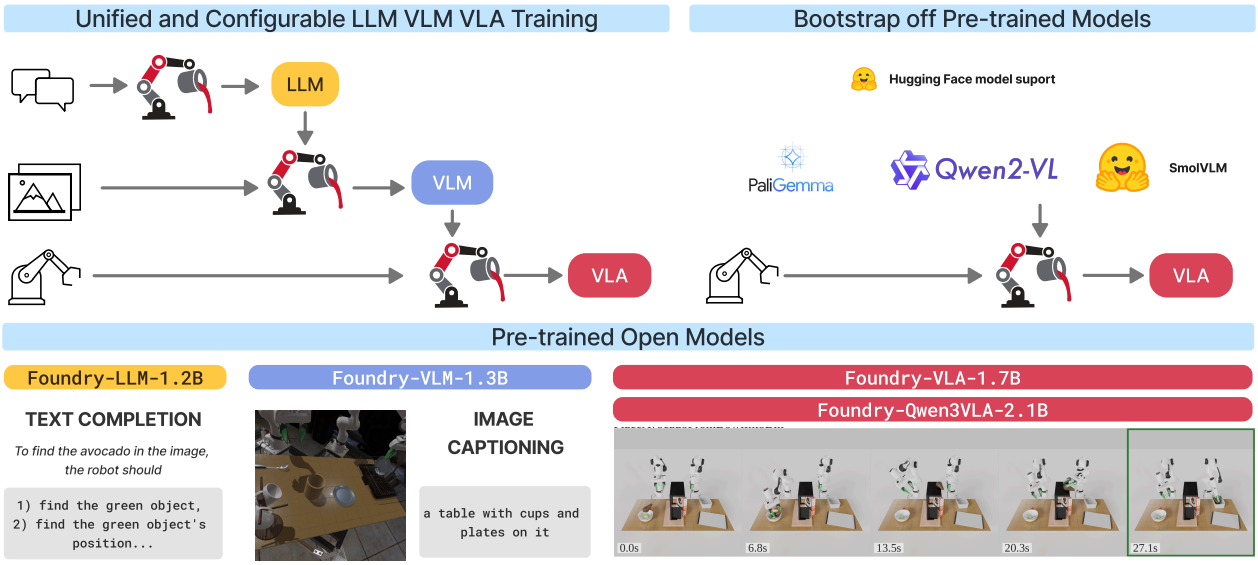


Figure 1 VLA Foundry overview. *Unified and Configurable LLM-VLM-VLA Pipeline*: VLA Foundry was designed to enable flexible model composition. For example, users can train an LLM, use the LLM to train a VLM, and use the VLM to train a VLA. *Bootstrap off Pre-trained Models*: VLA Foundry also supports loading off-the-shelf VLMs from Hugging Face. *Pre-trained Open Models*: We release LLM, VLM, and VLA models trained both from scratch and finetuned from open weights under a permissive license at https://huggingface.co/collections/TRI-ML/vla-foundry.

educational repositories such as nanoGPT [28] and LLMs from scratch [55] have further lowered the barrier to reproducing end-end-end LLM training pipelines. Dataset frameworks such as DCLM [33] and FineWeb [48] provide high quality language datasets. Together, these works highlight a shift towards reproducible and modular LLM training frameworks.

Vision-language model (VLM) frameworks must additionally address cross-modal representation learning and heterogeneous data integration. A common and prominent design pattern in VLMs is to couple a pretrained vision encoder with language model backbone, with intermediate modules responsible for aligning visual and text representations. Frameworks such as OpenFlamingo [2] operationalize this approach by providing infrastructure for interleaved image-text sequence construction, multimodal batching, and various architecture choices that enable the training of autoregressive VLMs on web-scale data. Similarly, LLaVA [38] offers a streamlined pipeline for multimodal instruction tuning, including data formatting, visual feature extraction, and supervised fine-tuning stages.

Other frameworks emphasize modularity and composability as first-class design principles. BLIP-2 [34] introduces a modular bridging component (Q-former) that decouples vision and language backbones, allowing each to be reused independently. Prismatic VLMs [27] extend this idea by explicitly structuring the framework around interchangeable components, enabling controlled experimentation over vision encoders, language models, and training mixtures. InternVL [12] further demonstrates how such modular designs can be scaled, incorporating large vision encoders and staged alignment strategies within a unified pipeline. Qwen [3, 4] offers state-of-the-art VLM capabilities in an open-source codebase. Complementing this line of work that primarily focuses on model architecture, dataset frameworks such as DataComp [22] provide standardized pipelines for construction and evaluating large-scale image-text datasets, addressing a critical bottleneck in reproducible multimodal training.

Across LLM and VLM settings, these frameworks expose several key dimensions of design including data pipelines (e.g. pre-tokenized vs. dynamic processing), model composition (e.g. monolithic vs. modular architectures), and training orchestration (e.g. distribution execution, staged vs joint optimization). As a result, existing systems span a spectrum from highly optimized distributed training backends to more modular, research oriented frameworks that facilitate experimentation with model architecture and data mixtures.

2.2 VLA Training Frameworks

In recent years, the open source vision-language-action (VLA) ecosystem has expanded rapidly, moving from a small number of isolated model releases to a broader set of training pipelines, pre-trained checkpoints and reproducible research frameworks. One of the earliest milestones of this shift was OpenVLA [30] which released a 7B-parameter model with a full PyTorch compatible codebase built off Prismatic [27]. Since then, a number of open-source alternatives have emerged. OpenPi [50] provides training and fine-tuning support for Physical Intelligence's π_0 model family, with base checkpoints pretrained on more than 10,000 hours of robot data. GR00T [46] pairs a vision-language backbone with a diffusion transformer action head in a dual-system architecture trained on real, simulated, and synthetic data. MolmoAct [32] explores a complementary direction by introducing an "Action Reasoning Model" that reasons in 3D space via depth-aware perception tokens rather than purely language-based action representations.

Beyond individual model development, several efforts have focused on standardization, infrastructure, and reproducibility of the entire VLA pipeline. LeRobot [10] adopts a community-first approach, emphasizing affordable hardware (SO-100/101 arms), integrating dataset collection, training, and deployment across affordable hardware platforms and lowering the barrier to real-world experimentation. They report results on a 450M-parameter model, SmolVLA [61], which is trained on a single GPU and remains competitive with much larger VLAs on standard benchmarks. VLAb [18] complements LeRobot as a dedicated pretraining library and SmolVLA reproduction kit. VLA-Scratch [69] provides a modular, performance-oriented training stack built on PyTorch FSDP2 with support for multiple VLM backbones (Qwen3-VL, PaliGemma, SmolVLM), heterogeneous dataset co-training, and Hydra-based configuration for rapid experimentation. StarVLA [17] further advances this direction by explicitly decoupling backbone architectures from action heads and supports both VLM backbones (e.g., Qwen-VL) and world-model backbones (e.g., Cosmos) with multiple options for action heads (autoregressive tokens, continuous decoding, and flow-matching), and integrates multiple benchmarks through a unified evaluation interface. Dexbotic [71] takes an experiment-centric approach, adopting a unified PyTorch toolbox with optimized reimplementations of various VLAs across different platforms such as the Franka and SO-101.

3 VLA Foundry

VLA Foundry is an open-source framework for training LLMs, VLMs, and VLAs within a single codebase. It is designed around end-to-end control of the embodied-model pipeline: the same training loop, data abstractions, and configuration interface extend from language pretraining to vision-language training and action learning. In this sense, VLA Foundry connects capabilities often treated separately across LLM/VLM training frameworks [60, 67, 27] and VLA frameworks [30, 50, 46, 10, 69, 17]. For robotics, this unified stack makes it practical to build and scale VLA systems while exploring new training recipes, architectures, and data mixtures. It supports both pre-training from scratch or initialization from pretrained Hugging Face backbones without requiring users to switch codebases across stages. An accompanying tutorial illustrates the full LLM→VLM→VLA training path from scratch[1].

In this section we present the key elements of the VLA Foundry framework that we believe make it a useful tool for policy pretraining research and experimentation.

[1] https://github.com/TRI-ML/vla_foundry/tutorials/training_llm_vlm_vla.ipynb

3.1 Design Principles

VLA Foundry is designed around four principles. We state them here; Section 3.2 shows how the architecture embodies each, and Appendix A gives the full reference.

1. **Modularity and Composability** – Components plug together rather than being baked into the training code. Models, data pipelines, encoders, and loss handlers are instantiated by name from a YAML-based configuration system that supports nested includes, so presets can be composed, locally overridden, and reused across experiments; swapping a vision encoder, a language backbone, or an entire model family is a single command-line change.
2. **Hackability and Interoperability** – Any component can be extended or replaced without touching the rest of the system. We avoid heavy framework wrappers (PyTorch Lightning, Hugging Face Trainer) and keep the training loop thin with parallelism primitives exposed rather than hidden, so users are not locked into a particular stack and can extend the framework with new modeling architectures or distributed-training paradigms as they emerge.
3. **Performance** – VLA Foundry targets researchers with moderate to medium-scale compute. Training throughput has been benchmarked across LLM, VLM, and VLA stages up to 128 GPUs across 16 nodes.
4. **Reproducibility** – Runs are repeatable at a given configuration. We rely on deterministic per-rank RNG seeding, dataloader state checkpointing for exact restarts, and immutable frozen dataclasses that prevent hidden configuration changes at runtime.

3.2 Framework

VLA Foundry's architecture has four layers: a YAML-based configuration system backed by frozen dataclasses, a registry that makes models and data pipelines pluggable, modality-specific preprocessing and dataloading, and a model-agnostic training loop. The remainder of this section walks through each; Appendix A.1 gives the full reference.

3.2.1 Modular Configuration System

VLA Foundry's modularity and composability is ensured by our configuration system. We base it on Draccus [42]: every parameter is declared in a dataclass and can be overridden by a YAML preset or a command-line argument, in increasing order of priority. Presets themselves are composable – a YAML file can inherit from others, so experiments are expressed by combining building blocks rather than by duplicating them. Parameters shared across modules (e.g., hidden dimensions, sequence lengths) are resolved once and propagated through the dataclass tree preventing silent cross-module mismatches. Configuration dataclasses are frozen to avoid run-time configuration changes that easily result in discrepancies between configuration files, logs, and runtime. See Appendix A.1.2 for details and a worked example.

3.2.2 Extending the Framework

VLA Foundry is designed to be hackable and extensible. Adding a model that fits an existing model type (LLM, VLM, or DP-VLA) requires only a parameter dataclass and a factory function, registered by name at import time; the model type's *batch handler* – which owns batching, loss construction, and output reduction – is shared, and a single training loop drives all model types. A new batch handler is needed only when introducing a new training paradigm. Adding a dataset follows a similar pattern. Raw data is converted to WebDataset [9] tar shards through a per-modality preprocessing stage. Preprocessing runs in parallel with Ray [44] and emits both training shards and the per-dataset statistics needed for normalization at training time. The dataloading pipeline itself is an ordered composition of small stages that users extend or reorder independently from the training loop. Dataloader can be mixed and each dataset contributes its own shards, statistics, and modalities with weighted dataset proportions.

3.2.3 Robotics Data Handling

Robotics data carries structure beyond what text and image-caption pipelines handle. Normalization is known to require careful handling in multi-dataset robotics training [65]; our `RoboticsNormalizer` supports global and per-timestep schemes, including percentile-based variants. Statistics can be merged across datasets. For percentile estimation and merging, we use t-digest [20]. Actions may be represented in absolute world-frame coordinates or relative to an anchor end effector pose, with rotations in the 6D continuous format [76] and relative poses composed in SE(3). Actions are chunked [75] in a configurable window of past and future time steps around an anchor: the future portion supervises the model, the past portion is available as input. Proprioceptive observations are causally restricted to past and current time steps. See Appendix A.2.

3.2.4 Training Performance

The training loop supports the standard levers for scaling distributed training – FSDP (with optional CPU offloading), mixed precision, gradient checkpointing, `torch.compile`, and gradient accumulation. See Appendix A.1.1. Figure 2 shows the training throughput across the different stages of our pipeline (LLM, VLM, and VLA). We used a 1.2 billion parameter language model, add a 86 million parameter ViT for the VLM and add a 325 million parameter transformer action head for the VLA. For the LLM, we used a sequence length of 2048 tokens with padding if needed. For the VLM, each image is represented with 64 tokens and the caption inputs are variable lengths but for consistency, we chose a total length of 256 tokens, truncated and padded sequences as needed. For the VLA, the model encodes 8 images from different cameras and timesteps, producing 512 tokens and a short task description. We pad VLA sequences dynamically and the average sequence length is 549 tokens. At this model scale, each GPU can hold the full model weights during training thus FSDP doesn't offer an advantage, and even shows weaker scaling for the VLM.

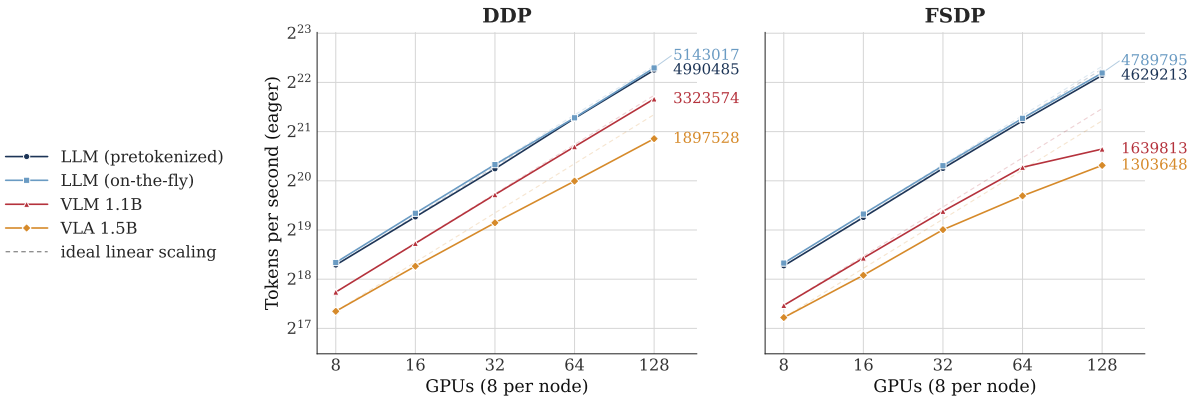


Figure 2 Throughput scaling as the number of GPU nodes is increased for the LLM, VLM, and VLA with either DDP or FSDP parallelization. Tests were done through Sagemaker on P5 nodes of 8 Nvidia H100 GPUs each. See section 4 for details about the models.

3.3 Evaluation

VLA Foundry supports evaluation on `lbm_eval_oss`, the open-source release of the LBM simulation benchmark [65]. The `lbm_eval_oss` framework is a challenging benchmark that uses the high fidelity Drake physics engine [68] to model the robots and scene dynamics. It defines 49 tasks to measure the performance of table top bimanual manipulation policies. Users can compare their own trained policies against the released checkpoints under a shared protocol. We ship the simulator as a Docker image, sidestepping platform-specific build and dependency issues across user environments. A simple dashboard lets users manage evaluation experiments, view rollout videos, and plot results as they accumulate.

Additionally, we provide rigorous statistical analysis via STEP [63] to compare success rates of multiple policies. Following [65, 35, 31], the dashboard has violin plots for Bayesian estimates of individual success rates, with Compact Letter Display (CLD) [53] attached for comparison. Policies not sharing any CLD alphabet

are significantly different at 5% family-wise error rate (FWER). Notably, our statistical framework lets the user base decisions on intermediate comparisons, as results are gathered. The user can decide to stop an evaluation early to save time, or to collect more rollouts than initially planned to seek higher statistical power. Such a practice would constitute harmful "p-hacking" [64] for standard statistical tests such as Barnard's test [6]. More details can be found in prior work [63, 45, 35]; we include our general design principles and suggested best practices as documentation in the dashboard. In particular, when concatenating results over multiple tasks for aggregate comparison, we balance the per-task sample size for each policy to ensure that the aggregate represents an unbiased estimate of the policy's equally-weighted multi-task performance. For instance, if some Model A has $[50, 49, 50, 50]$ rollouts across 4 tasks, where the second task is missing one rollout, the results are truncated to $[49, 49, 49, 49]$ before aggregation. Therefore, Model A's aggregated performance is computed by 196 rollouts instead of 199 before it is fed to STEP for comparison. We note that this unbiased aggregation was not strictly enforced in the prior work [65]. Our results as well as those from [65] are included in the dashboard so that users can compare their own experiments with the reported numbers from the released checkpoints.

4 FOUNDRY-VLA-1.7B and FOUNDRY-QWEN3VLA-2.1B

Having described the framework itself, we now turn to two applications. We release two VLA models types alongside this report. Each showcase different capabilities of the VLA Foundry pipeline:

- **FOUNDRY-VLA-1.7B** – trained fully from scratch along the LLM→VLM→VLA pipeline, demonstrating end-to-end controllability over the training recipe.
- **FOUNDRY-QWEN3VLA-2.1B** – trained on top of a pretrained Qwen3-VL 2B backbone, showing that the same codebase efficiently supports the traditional VLM→VLA recipe and that a stronger/larger VLM backbone translates into a more capable VLA.

Both models share the same action expert architecture (Section 4.1). Section 4.1 walks through the from-scratch pipeline, Section 4.2 describes the Qwen3-based model, and Section 4.3 reports simulation results, including ablations over multi-task vs. single-task training as well as sim-only and real-only subsets.

4.1 FOUNDRY-VLA-1.7B: Training From Scratch

FOUNDRY-VLA-1.7B is our end-to-end demonstration of VLA Foundry's full-pipeline controllability. We first train a language model (LLM), extend it to a vision-language model (VLM), and finally adapt it into a vision-language-action (VLA) model (Figure 1). We release the intermediate FOUNDRY-LLM-1.2B and FOUNDRY-VLM-1.3B checkpoints in addition to FOUNDRY-VLA-1.7B so that the community can reproduce or modify any stage of the pipeline[2].

LLM training We used a standard transformer architecture [23] to define a 1.2 billion parameter model with a hidden dimension of 2048, 24 layers, and 16 heads. Note that, following the convention [26], we discount the additional 200 million parameters of the embedding layers.

The model was trained on 500 million samples (or 1 trillion tokens) from the openly available DCLM [33] dataset with a sequence length of 2048. Text was tokenized with the processor `HuggingFaceTB/SmolVLM2-256M-Video-Instruct`, which has a vocabulary size of 49,280. We used a warmup-stable-decay learning rate schedule [25]. The model and its full set of configuration parameters is available on HuggingFace[2]. Table 1 shows results of this model on standard benchmarks before the learning rate decay phase and after the full training. Note the lack of instruction tuning and the size of the model keep it close to random chance on difficult benchmarks such as MMLU; however, we see good results well above random chance on easier benchmarks.

VLM training We add a 86 million parameter randomly initialized vision transformer (ViT) [19], with a similar architecture as CLIP [54], to encode (224×224) input images. A pixel-shuffle [41], operation is

[2] https://huggingface.co/collections/TRI-ML/vla-foundry

Table 1 LLM evaluation results on multiple-choice reasoning benchmarks. HS = HellaSwag, WG = WinoGrande, OBQA = OpenBookQA. See descriptions, references, and terms of use in Appendix C.2.

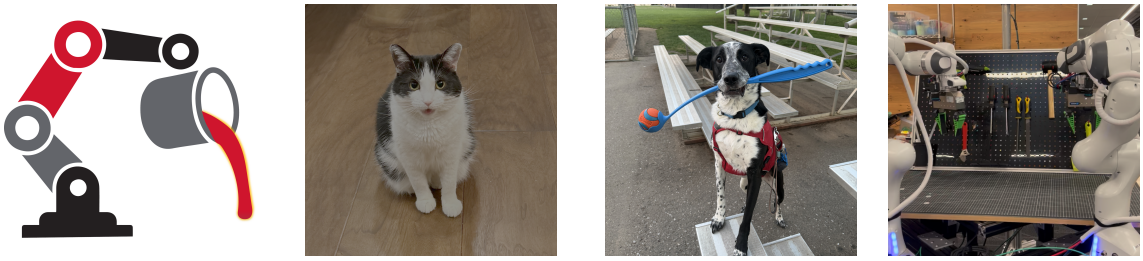
Model	HS	MMLU	ARC-e	ARC-c	PIQA	WG	OBQA	BoolQ
FOUNDRY-LLM-1.2B (800B tokens)	64.3	26.0	70.3	37.0	75.8	60.9	40.0	63.2
FOUNDRY-LLM-1.2B (1T tokens)	66.7	26.6	71.7	39.3	77.5	62.6	40.8	65.4

used as pooling to reduce the sequence length of the image. We assemble the ViT and pooling with the previously pre-trained 1.2B LLM at 800B tokens of training – before the learning rate cooldown, following recommendations from [29]. The VLM is trained with 200M samples of the openly available DataCompDR-1B [22][3]. Our results are reported in table 2 as evidence of end-to-end training functionality rather than as a claim of optimal performance. We also include qualitative examples in figure 3.

Although in this instance we use a randomly initialized ViT and the in-house LLM, both could instead be replaced by off-the-shelf pre-trained components such as SigLIP [74] or DINO [47, 62] which would likely lead to improved model performance. Alternatively, the VLM itself can take advantage of pre-trained backbones such as PaliGemma2 [7] or Qwen3-VL [3]; this is precisely the route we take for FOUNDRY-QWEN3VLA-2.1B in Section 4.2. Here we show that VLA Foundry supports all stages of training and can produce a functional VLM backbone, giving full control to users to experiment with known training data and procedures, modify architectures, and train or fine-tune any part of the model.

Table 2 COCO_VAL captioning evaluation. BLEU-n: Measures n-gram overlap between the generated caption and the references, ROUGE-L: Measures longest common subsequences, CIDEr: Measures weighted n-gram similarity (with n=1-4) so distinctive, informative phrases count more than common ones.

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE_L	CIDEr
FOUNDRY-VLM-1.3B 165M	57.25	37.12	23.23	14.44	37.13	50.17
FOUNDRY-VLM-1.3B 200M	58.64	38.62	24.49	15.57	38.17	55.14



a red and black robot arm with a red handle

a cat sitting on the floor looking at the camera

a dog with a leash on a bench

a robot is working on a project in a workshop

Figure 3 VLM 1.1B caption-only model predictions (greedy decoding). The model uses normalized, 224×224 input images to generate the captions. Images were sampled from the authors' phones (+ logo) to avoid any contamination.

VLA training We define the VLA architecture on top of the previous VLM (Figure 4). To extend the VLM architecture to to predict robot actions, we begin by adding a new *observation* token to the LLM vocabulary. The VLM input sequence is composed of images and a text describing a task as well as the new observation token, in that order. The embedded sequence that is fed to the LLM part of the VLM is composed of the concatenated embedded image patches from multiple images, embedded text tokens, and the embedded observation token. The hidden features of the last N (in our experiments, 4) layers of the VLM matching the observation token are used to condition a flow transformer that denoises an action sequence. This action head is a 325 million parameter transformer with the same architecture as the LLM (except a vocabulary size of 0). Its input sequence is composed of the concatenated hidden features from the VLM, optionally, the

[3]Image links from this dataset are known to break, which limits exact reproducibility.

proprioception encoded with a linear layer and the noised action sequence also encoded by a linear layer, in that order. The output action sequence is trained with the flow-matching objective [37]. We denote this model FOUNDRY-VLA-1.7B.

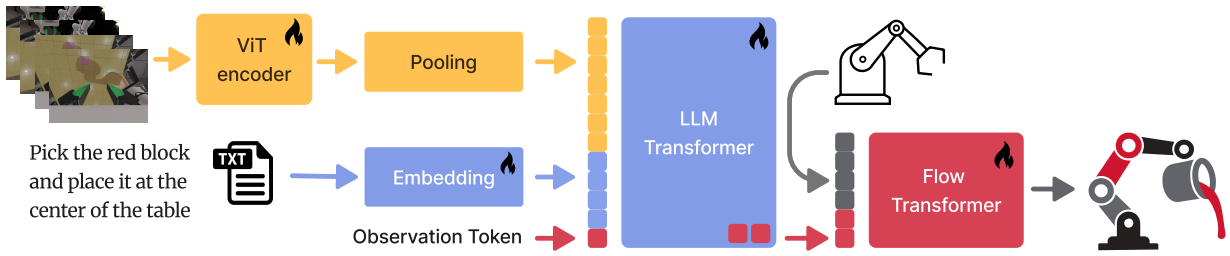


Figure 4 FOUNDRY-VLA-1.7B architecture. Four images over two timesteps each are fed into the same ViT image encoder. For each of the 8 images, the result is pooled with "pixel-shuffle" [41] (see appendix C.4) and projected into the embedding space of the LLM. An additional observation token is appended to the sequence. The LLM embedding of the last layer matching this token is passed to a flow transformer with a noised action sequence. The flow transformer outputs the predicted denoising direction.

We train FOUNDRY-VLA-1.7B models on a data mixtures consisting of both simulated and real teleoperated demonstrations from stationary bimanual manipulation stations described in our previous work LBM [65]. The data mix features 42 tasks in simulation and 361 tasks in the real world; 39 tasks are replicated in both real and simulation with copies of the stations and manipulands. Unlike our previous work we do not train on open-sourced data such as OXE [16] or data collected with a universal manipulation device (UMI) [13]. Further details regarding the dataset, including number of episodes per benchmark task and differences from the dataset of LBM, can be found in Section C.6. Unless otherwise noted, FOUNDRY-VLA-1.7B and FOUNDRY-QWEN3VLA-2.1B are trained on a multi-task mixture of both real and simulation data[4]. We additionally train multi-task variants of FOUNDRY-VLA-1.7B on simulation-only and real-only subsets, yielding FOUNDRY-VLA-1.7B-SIM and FOUNDRY-VLA-1.7B-REAL respectively; these are used for the ablations in Section 4.3.

4.2 FOUNDRY-QWEN3VLA-2.1B: Leveraging a Strong VLM Backbone

A key design principle of VLA Foundry is that architectural components can be swapped with minimal effort. To exercise this, we also train a VLA with the pretrained Qwen3-VL 2B model [4] as backbone. We reuse the same architecture as FOUNDRY-VLA-1.7B for the action flow transformer and train on the full real and simulated data mixture. We denote this model FOUNDRY-QWEN3VLA-2.1B.

The performance of FOUNDRY-QWEN3VLA-2.1B demonstrates that a stronger and larger VLM backbone yields stronger VLA performance: FOUNDRY-QWEN3VLA-2.1B improves over FOUNDRY-VLA-1.7B on the shared simulation benchmark and outperforms our prior closed-source multi-task LBM policy in a statistically significant manner by more than 20 percentage points (Figure 5). Moreover, we show that the traditional VLM→VLA recipe can be reproduced efficiently inside VLA Foundry, on the same training loop, dataloader, and preprocessing pipeline used for the from-scratch run, so practitioners do not need a separate training stack to adopt off-the-shelf backbones.

4.3 Simulation Evaluation Results

In line with LBM [65], we evaluate our models on a set of 16 simulation tasks (see Figure 6) seen at training time, as well as 3 simulation tasks held out from training[5], and compare performance with the statistical analysis tools introduced in Section 3.3. The tasks in the benchmark vary in complexity and manipulation modes: PutKiwiInCenterOfTable is a simple pick-and-place task, PutRedBellPepperInBin requires one arm

[4]Download instructions for the processed LBM simulation data can be found in the codebase.

[5]We do not evaluate on the distribution shift variant of the benchmark or additional long horizon simulation tasks; we leave this for future work.

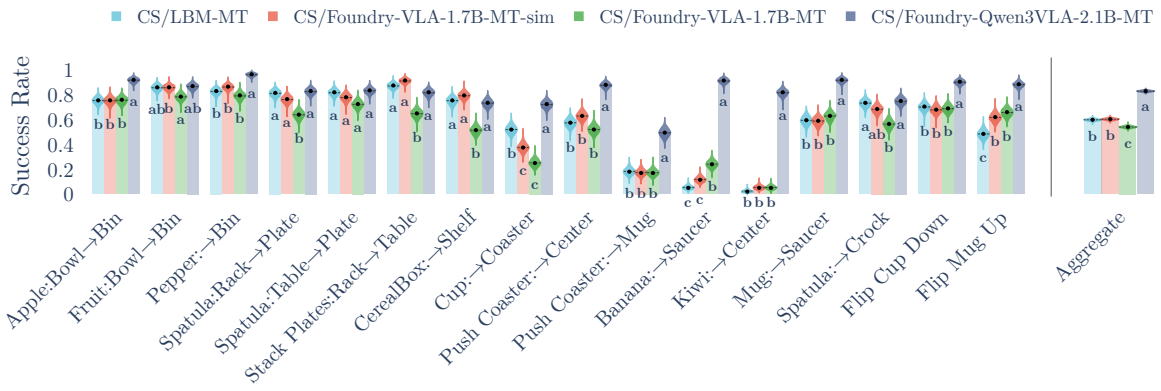


Figure 5 We compare our multi-task models—FOUNDRY-VLA-1.7B-SIM, FOUNDRY-VLA-1.7B-FULL, and FOUNDRY-QWEN3VLA-2.1B—against the LBM-MT [65] multi-task model on a set of seen tasks in `lbm_eval_cs`. In aggregate, LBM-MT and FOUNDRY-VLA-1.7B-SIM are on par, while FOUNDRY-QWEN3VLA-2.1B far outperforms the rest. Note that here only FOUNDRY-VLA-1.7B-FULL and FOUNDRY-QWEN3VLA-2.1B share the same exact robot training data; for more details refer to Section 4.3.1.

to place the bell pepper onto the shelf and the other arm to retrieve the item and place it in the bin, TurnCupUpsideDown requires only one arm but uses a wider range of motion especially in end effector rotations, and PushCoasterToMug requires non-prehensile manipulation. We evaluate on both the closed-source benchmark `lbm_eval_cs` from which results were reported in [65] and the later open-sourced version `lbm_eval_oss` [66]. Due to updates between the two versions, policy performance can vary substantially, as `lbm_eval_oss` can be considered a distribution-shifted version of the former; a comparison between model performance on the closed-source evaluation and the open-sourced evaluation on selected checkpoints is provided in Figure 11. For brevity, we use the following notation:

- **CS**: closed-source; the simulation environment used in [65]; it is largely the same used for data collection
- **OSS**: open-source software; the simulation environment that is openly accessible from [66]
- **ST**: single-task; the model is trained and evaluated on the same task
- **MT**: multi-task; the model is trained on multiple tasks (can be simulation, real, or both)
- **FT**: multi-task finetuned: a multi-task checkpoint that is finetuned on a specific evaluation task

For both ST and FT, each task is evaluated with a specific set of model weights while MT models are evaluated on all the tasks with the same weights. All experiments are done with an evaluation budget of 200 rollout episodes[6]. Note that some simulation seeds can also result in immediate, default successes; the raw data to produce the violin plots is included in the codebase.

In the following sections, we first compare models trained in VLA Foundry to LBM on the closed source simulator. We then compare ST, MT, and FT training results for FOUNDRY-VLA-1.7B and FOUNDRY-QWEN3VLA-2.1B on seen and unseen tasks. For details of the violin plots and the CLD letters, refer to Section 3.3. Additional results can be found in Section D.

4.3.1 Comparison with LBM

First, we compare our results with LBM, a multi-task model from previous work [65]. LBM is a 566 million parameter model that is composed of a pre-trained CLIP model for text and image embedding and a diffusion transformer head; in contrast to FOUNDRY-QWEN3VLA-2.1B and FOUNDRY-VLA-1.7B, the LBM action head architecture utilizes cross-attention for the diffusion conditioning. Additionally, the

[6]The results in this report are collated from an initial run and a followup run to patch missing trials; the evaluation results were then combined by keeping the most recent simulated episode. Therefore, exactly 200 rollout episodes were collected for each model.

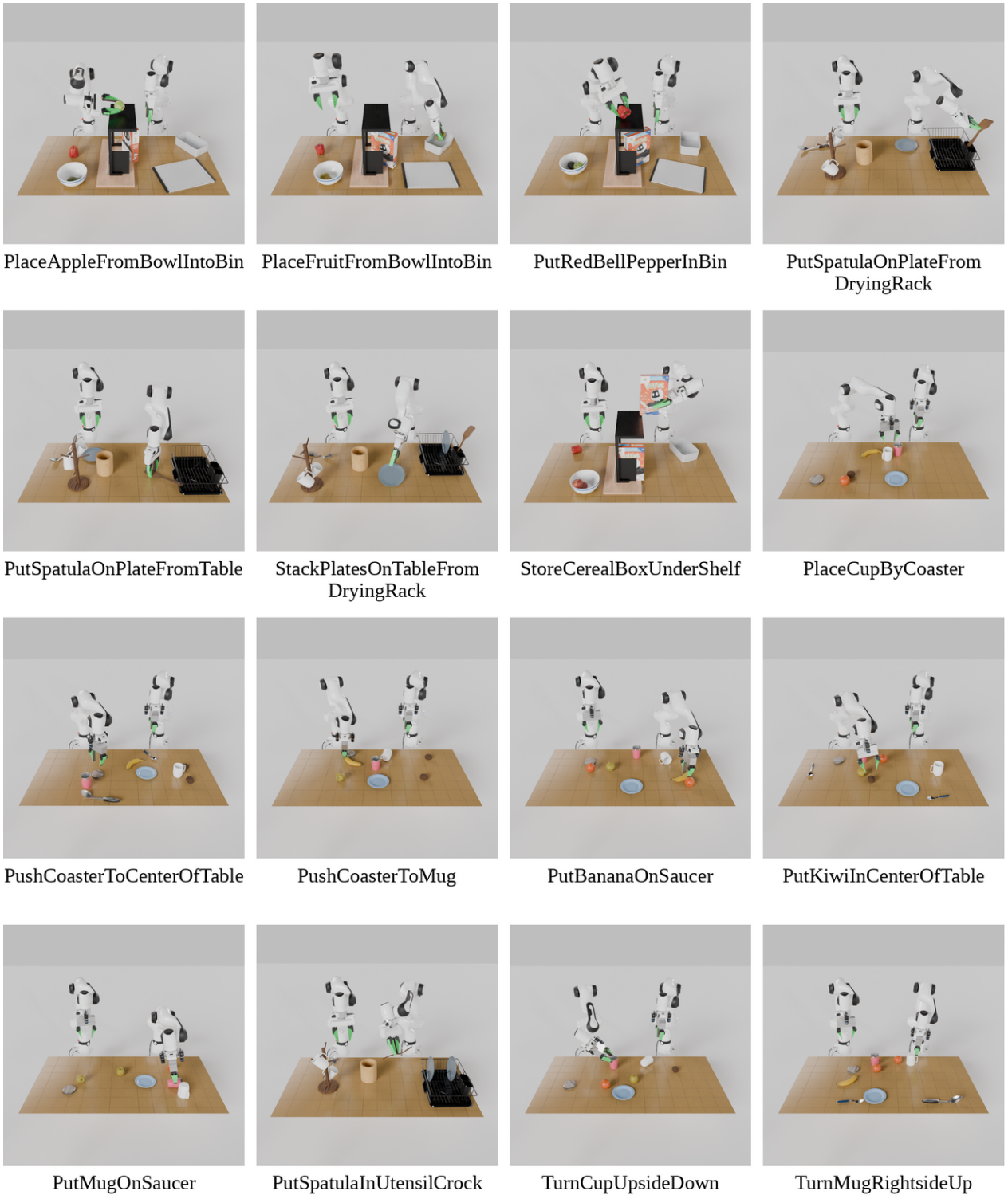


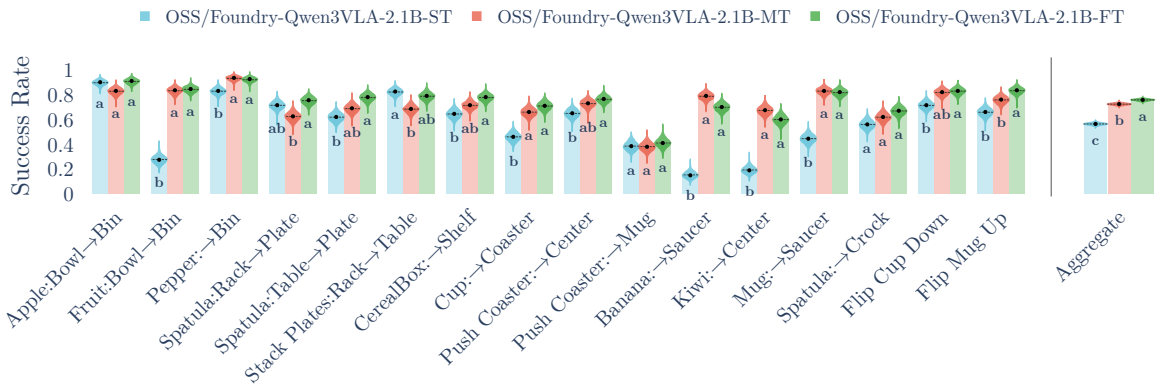
Figure 6 Overview of seen simulation evaluation tasks. The `lbm_eval_oss` task suite spans tasks that require different qualities of manipulation capabilities: from pick-and-place to non-prehensile manipulation to bimanual coordination. Here, we show a single still from about the midpoint of a successful rollout from FOUNDRY-QWEN3VLA-2.1B. Video versions of these images can be found at https://tri-ml.github.io/vla_foundry. We note that the images here build on top of [49], where we re-light and re-render the Meshcat scenes at the desired frame rate from rollouts using Blender's Cycles after filtering out station geometry such as the external camera mounts, and table base for visual clarity; a representative example of sensor measurements actually used for model inference can be seen in Figure 16. For a comparable figure of failed rollouts, refer to Figure 14.

LBM model uses all camera images, zero padding when cameras are not present in certain data, whereas FOUNDRY-QWEN3VLA-2.1B and FOUNDRY-VLA-1.7B use only the two wrist camera and two external camera images shared between the different simulation stations.

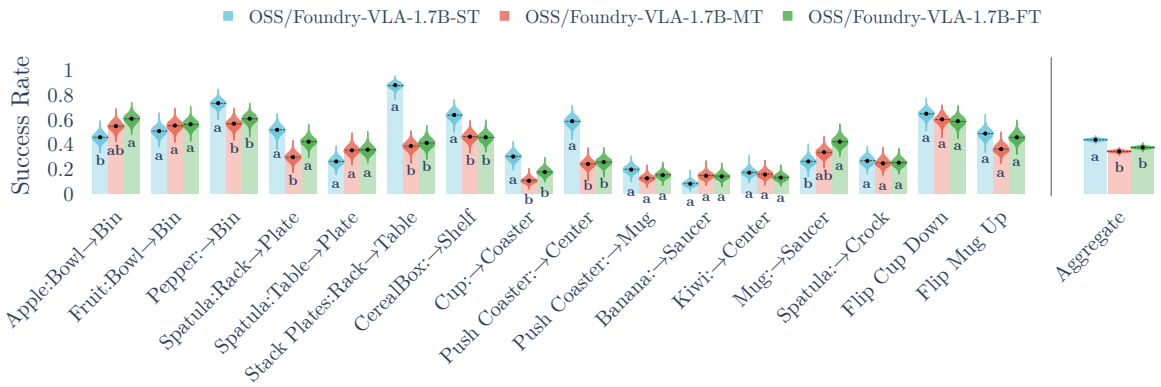
Figure 5 compares FOUNDRY-QWEN3VLA-2.1B, FOUNDRY-VLA-1.7B, and FOUNDRY-VLA-1.7B-SIM multi-task against LBM multi-task on `lbm_eval_cs`. In aggregate, FOUNDRY-QWEN3VLA-2.1B outperforms multi-task LBM in terms of task success by a wide margin, while LBM and FOUNDRY-VLA-1.7B-SIM are statistically on par with each other. FOUNDRY-VLA-1.7B is the worst of the four models considered. Section 4.3.3 includes further evaluation and discussion on the effect of data recipes in the context of FOUNDRY-VLA-1.7B.

4.3.2 Training Stage Comparisons

Figure 7 (a) shows the results of FOUNDRY-QWEN3VLA-2.1B at different training stages: direct single-task training, multi-task training, and multi-task finetuned on each task. After multi-task training on the simulation and real data, the FOUNDRY-QWEN3VLA-2.1B model shows better performance than the single task training regime; finetuning the multi-task model on single seen tasks further improves performance in aggregate. Figure 7 (b) shows the same results from FOUNDRY-VLA-1.7B. We see that while for some tasks such as `Apple:Bowl` → `Bin` the finetuned model outperforms the single task model, the opposite is true for other tasks such as `Stack Plates:Rack` → `Table`; in aggregate, the multi-task training and finetuning are statistically worse than the single task model.



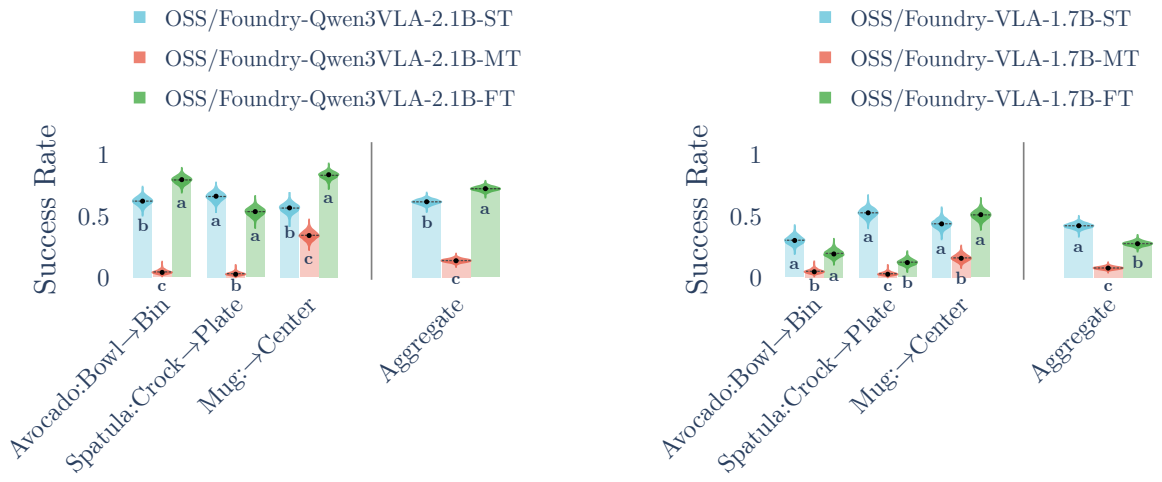
(a) FOUNDRY-QWEN3VLA-2.1B model family



(b) FOUNDRY-VLA-1.7B model family

Figure 7 Simulation results on `lbm_eval_oss` (seen tasks). Aggregate performance increases from ST to MT to FT for the FOUNDRY-QWEN3VLA-2.1B family; FOUNDRY-VLA-1.7B performance is more mixed; the MT and FT variants are statistically worse than the ST.

Figure 8 shows the same models but evaluated on the 3 held-out tasks that are not part of the multi-task dataset. In both multi-task models, we observe some small amount of zero-shot generalization. However, while finetuning the multi-task FOUNDRY-QWEN3VLA-2.1B model results in better performance than the single task variant in aggregate, the same is not true for FOUNDRY-VLA-1.7B. These results are consistent with the hypothesis that stronger backbones can result in improved policy outcomes.



(a) FOUNDRY-QWEN3VLA-2.1B model family

(b) FOUNDRY-VLA-1.7B model family

Figure 8 Simulation results on `lbm_eval_oss` (unseen tasks). Both FOUNDRY-VLA-1.7B and FOUNDRY-QWEN3VLA-2.1B demonstrate non-zero success rates 0-shot from real training to simulated evaluation.

4.3.3 Data Subset Comparisons

To isolate the contribution of each data source, we additionally compare the results of training FOUNDRY-VLA-1.7B on three subsets of data: simulation only, real robot only, and both combined. Simulation results of multi-task models trained on each of the 3 subsets are given in Figure 9. All three models were trained for the same amount of compute but different amounts of data, i.e., the simulation only model was trained on more epochs of the same data. As expected the real-only training shows almost 0% success rate because the simulation environment is out of its training distribution. Similar to Figure 5, the simulation only variant FOUNDRY-VLA-1.7B-SIM performs the best in aggregate. The number of episodes used to finetune the seen tasks can be found in Table 6. Potential hypotheseses for the slightly worse performance compared to FOUNDRY-VLA-1.7B-SIM include model undertraining or the representational power of the model being split between real and simulated tasks; we leave further investigation to future work.

5 Conclusions

Limitations This initial release reflects deliberate choices in scope rather than framework constraints. Our reported evaluation is restricted to closed-loop LBM simulation on a narrow slice of embodiments, and we do not yet include real-hardware numbers; VLA Foundry's shared evaluation and dataloader abstractions are designed so that additional simulation suites (e.g., LIBERO, SimplerEnv, RoboCasa), new embodiments, and on-robot evaluation can be added without touching the core training stack. All experiments in this report use a flow-matching action head; while additional heads such as a diffusion policy are already implemented in the codebase, the action head is a modular component and integrating further variants – for example, autoregressive discrete action tokenizations – requires only a new head module rather than changes to the training loop or data pipeline. Finally, although VLA Foundry exposes the full LLM–VLM–VLA pipeline with probabilistic multi-modal mixing, we do not yet characterize optimal data recipes across stages, nor do we address safety, alignment, or failure-mode detection for embodied agents. We view these as open research

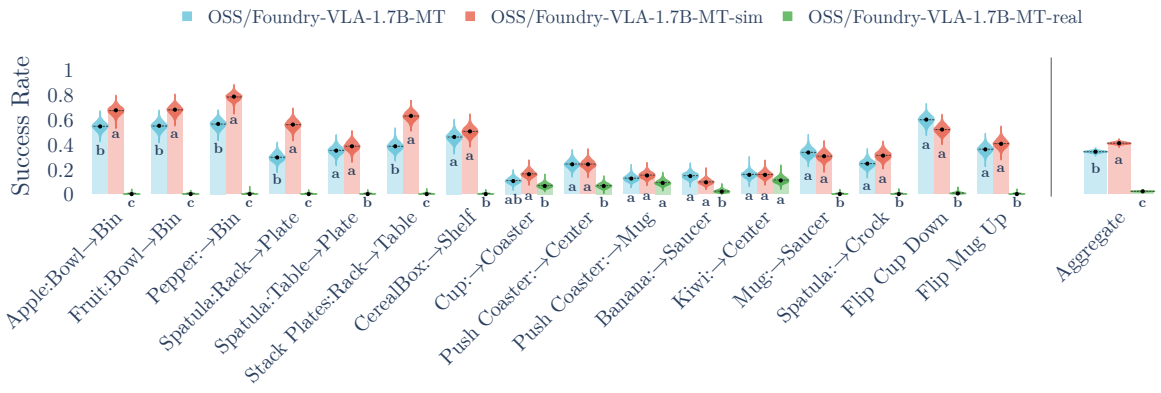


Figure 9 Simulation results of our three multitask FOUNDRY-VLA-1.7B variants: trained on simulated data FOUNDRY-VLA-1.7B-SIM, real data FOUNDRY-VLA-1.7B-REAL, and both combined FOUNDRY-VLA-1.7B-FULL.

directions that VLA Foundry is well-positioned to enable, and we invite the community to build on it to explore them.

Conclusion In this technical report, we introduced **VLA Foundry**, an open-source framework that unifies LLM, VLM, and VLA training within a single codebase. The framework provides end-to-end control over the embodied-model pipeline – from language pretraining through action learning – with shared abstractions for data, configuration, training, and evaluation. Alongside the framework, we released two model families: FOUNDRY-VLA-1.7B, trained fully from scratch through the LLM→VLM→VLA pipeline, and FOUNDRY-QWEN3VLA-2.1B, built on a pretrained Qwen3-VL backbone with the same action head and training recipe. On closed-loop LBM evaluation, FOUNDRY-VLA-1.7B-SIM is statistically on par with our prior closed-source LBM performance over our 16-tasks benchmark. FOUNDRY-QWEN3VLA-2.1B outperforms both models with a wide margin of 23 percentage points on average. We demonstrated that VLA Foundry can be used to build VLAs both from-scratch and starting with a pretrained-backbone model. Together with the released checkpoints, the statistical comparison dashboard, and integration of `lbm_eval_oss`, VLA Foundry's unified LLM–VLM–VLA stack enables the community to explore the design space that connects these stages – training recipes, multi-modal data mixing, fusion architectures – within a single codebase. We hope these tools will serve the community and that the community will contribute to their improvements.

5.1 Acknowledgements

VLA Foundry would not be possible without the support of multiple teams and individuals at TRI.

Max Bajracharya managed the VLA team and provided guidance throughout the project.

Mark Zolotas and Tim Chu provided feedback in various stages of the project and contributed quality-of-life improvements to the general infrastructure. We also thank Aykut Onol, Mengchao Zhang, Mark Zolotas, Naveen Kuppuswamy, and Sunny Sun for testing early versions of VLA Foundry on new embodiments. Ian McMahon and Jeremy Nimmer provided support for simulation evaluation. Andrew Beaulieu provided feedback to VLA Foundry and helped coordinate efforts with the TRI team in Cambridge.

Rishi Shah implemented small bugfixes and quality-of-life improvements, and helped test out VLA Foundry on various sim and hardware environments.

Richard Cheng, Chen Zou, Shanmuga Harikumar, Daiki Mori, Yukinori Kurahashi, and Takahiro Yamazaki provided support for testing VLA foundry on new simulation and mobile hardware environments.

Chen Xu and Swati Gupta helped in early versions of our diffusion implementation. Pooja Kabra, Nagarjun Vinukonda, and David Berkowitz provided additional engineering support. We also thank Rhythm Syed, Jose Barreiros, Krishnan Srinivasan, and Blake Wulfe for support in various stages of the project.

Satya Kotari provided compute infrastructure and AWS support.

Nicholas Pfaff provided advice and code for rendering simulation rollouts.

Finally, we thank our robot teachers – Emma Dixon, Christopher Rodriguez, Derrick Seale, and Rudy Bravo for helping validate VLA Foundry on hardware. We also thank Patrick Miller and Masha Itkina for coordinating our data collection efforts.

5.2 Disclaimers

Parts of the initial draft of the repo were taken from OpenLM [23]. Parts of the ViT implementation were taken from nanoVLM [70].
The VLA Foundry codebase contains some code generated by LLMs.

References

[1] Alex Andonian et al. *GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch*. Version 2.0.0. Sept. 2023. DOI: 10.5281/zenodo.5879544. URL: https://www.github.com/eleutherai/gpt-neox.

[2] Anas Awadalla et al. "OpenFlamingo: An Open-Source Framework for Training Large Autoregressive Vision-Language Models". In: *arXiv preprint arXiv:2308.01390* (2023).

[3] Jinze Bai et al. "Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond". In: *arXiv preprint arXiv:2308.12966* (2023).

[4] Shuai Bai et al. "Qwen3-vl technical report". In: *arXiv preprint arXiv:2511.21631* (2025).

[5] Rohit Bandaru. "Foundation Models for Robotics: Vision-Language-Action (VLA)". In: (Sept. 2025). URL: https://rohitbandaru.github.io/blog/Foundation-Models-for-Robotics-VLA/.

[6] G. A. Barnard. "Significance Tests for 2×2 Tables". In: *Biometrika* 34.1-2 (Jan. 1947), pp. 123–138. ISSN: 0006-3444. DOI: 10.1093/biomet/34.1-2.123. (Visited on 01/20/2025).

[7] Lucas Beyer et al. *PaliGemma: A versatile 3B VLM for transfer*. en. July 2024. URL: https://arxiv.org/abs/2407.07726v1 (visited on 09/05/2024).

[8] Yonatan Bisk et al. "PIQA: Reasoning about Physical Commonsense in Natural Language". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 7432–7439. URL: https://aaai.org/ojs/index.php/AAAI/article/view/6239.

[9] Thomas Breuel. *WebDataset: A High-Performance Python-Based I/O System for Large Deep Learning Problems*. 2020. URL: https://github.com/webdataset/webdataset.

[10] Remi Cadene et al. *LeRobot: State-of-the-art Machine Learning for Real-World Robotics in Pytorch*. https://github.com/huggingface/lerobot. 2024.

[11] Xinlei Chen et al. "Microsoft coco captions: Data collection and evaluation server". In: *arXiv preprint arXiv:1504.00325* (2015).

[12] Zhe Chen et al. "Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 24185–24198.

[13] Cheng Chi et al. "Universal Manipulation Interface: In-The-Wild Robot Teaching Without In-The-Wild Robots". In: *Proceedings of Robotics: Science and Systems (RSS)*. 2024.

[14] Christopher Clark et al. "BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 2924–2936. DOI: 10.18653/v1/N19-1300. URL: https://aclanthology.org/N19-1300.

[15] Peter Clark et al. "Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge". In: *ArXiv preprint* abs/1803.05457 (2018). URL: https://arxiv.org/abs/1803.05457.

[16] Open X-Embodiment Collaboration et al. *Open X-Embodiment: Robotic Learning Datasets and RT-X Models*. https://arxiv.org/abs/2310.08864. 2023.

[17] StarVLA Community. "StarVLA: A Lego-like Codebase for Vision-Language-Action Model Developing". In: *arXiv preprint arXiv:2604.05014* (2026).

[18] Mustafa Shukor Dana Aubakirova, Jade Cholgari, and Leandro von Werra. *VLAb: Your Laboratory for Pretraining VLAs*. https://github.com/huggingface/vlab. 2025.

[19] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV]. URL: https://arxiv.org/abs/2010.11929.

[20] Ted Dunning and Otmar Ertl. "Computing Extremely Accurate Quantiles Using t-Digests". In: *arXiv preprint arXiv:1902.04023* (2019). URL: https://arxiv.org/abs/1902.04023.

[21] EGalahad. *VLA-Scratch: A Modular, Performant, Efficient Stack For Vision-Language-Action Models*. https://github.com/EGalahad/vla-scratch. GitHub repository. 2025.

[22] Samir Yitzhak Gadre et al. "Datacomp: In search of the next generation of multimodal datasets". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 27092–27112.

[23] Suchin Gururangan et al. *open_ lm: a minimal but performative language modeling (LM) repository*. GitHub repository. 2023. URL: https://github.com/mlfoundations/open_lm/.

[24] Dan Hendrycks et al. "Measuring Massive Multitask Language Understanding". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=d7KBjmI3GmQ.

[25] Shengding Hu et al. "Minicpm: Unveiling the potential of small language models with scalable training strategies". In: *arXiv preprint arXiv:2404.06395* (2024).

[26] Jared Kaplan et al. "Scaling laws for neural language models". In: *arXiv preprint arXiv:2001.08361* (2020).

[27] Siddharth Karamcheti et al. "Prismatic VLMs: Investigating the Design Space of Visually-Conditioned Language Models". In: *International Conference on Machine Learning (ICML)*. 2024.

[28] Andrej Karpathy. *nanoGPT: The Simplest, Fastest Repository for Training/Finetuning Medium-Sized GPTs*. 2022. URL: https://github.com/karpathy/nanoGPT.

[29] Sedrick Keh et al. "Should VLMs be Pre-trained with Image Data?" In: *arXiv preprint arXiv:2503.07603* (2025).

[30] Moo Jin Kim et al. "Openvla: An open-source vision-language-action model". In: *arXiv preprint arXiv:2406.09246* (2024).

[31] Pepijn Kooijmans et al. *Unfolding Robotics: The Open-Source Recipe for Teaching a Robot to Fold Your Clothes*. Accessed: 2026-04-17. 2026. URL: https://huggingface.co/spaces/lerobot/robot-folding.

[32] Jason Lee et al. *MolmoAct: Action Reasoning Models that can Reason in Space*. 2025. arXiv: 2508.07917 [cs.RO]. URL: https://arxiv.org/abs/2508.07917.

[33] Jeffrey Li et al. "Datacomp-lm: In search of the next generation of training sets for language models". In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 14200–14282.

[34] Junnan Li et al. "BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models". In: *Proceedings of the 40th International Conference on Machine Learning (ICML)*. 2023. URL: https://arxiv.org/abs/2301.12597.

[35] Fanqi Lin et al. "A Systematic Study of Data Modalities and Strategies for Co-training Large Behavior Models for Robot Manipulation". In: *arXiv preprint arXiv:2602.01067* (2026).

[36] Xuewu Lin et al. "HoloBrain-0 Technical Report". In: *arXiv preprint arXiv:2602.12062* (2026). URL: https://arxiv.org/abs/2602.12062.

[37] Yaron Lipman et al. "Flow matching for generative modeling". In: *arXiv preprint arXiv:2210.02747* (2022).

[38] Haotian Liu et al. "Visual instruction tuning". In: *Advances in neural information processing systems* 36 (2023), pp. 34892–34916.

[39] Zhengzhong Liu et al. "LLM360 K2: Building a 65B 360-Open-Source Large Language Model from Scratch". In: *arXiv preprint arXiv:2501.07124* (2025). DOI: 10.48550/arXiv.2501.07124. URL: https://arxiv.org/abs/2501.07124.

[40] Zhengzhong Liu et al. "LLM360: Towards Fully Transparent Open-Source LLMs". In: *arXiv preprint arXiv:2312.06550* (2023). DOI: 10.48550/arXiv.2312.06550. URL: https://arxiv.org/abs/2312.06550.

[41] Andrés Marafioti et al. "Smolvlm: Redefining small and efficient multimodal models". In: *arXiv preprint arXiv:2504.05299* (2025).

[42] marin-community. *Draccus: Configuration with Dataclasses+YAML+Argparse*. https://github.com/marin-community/draccus. 2026.

[43] Todor Mihaylov et al. "Can a Suit of Armor Conduct Electricity? A New Dataset for Open Book Question Answering". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, 2018, pp. 2381–2391. DOI: 10.18653/v1/D18-1260. URL: https://aclanthology.org/D18-1260.

[44] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 561–577. URL: https://www.usenix.org/conference/osdi18/presentation/moritz.

[45] Haruki Nishimura and Masha Itkina. *Statistical Thinking for Robot Policy Evaluation: From Rigorous A/B Testing to Effective Visualization*. Medium. Accessed: 2026-04-17. 2026. URL: https://medium.com/toyotaresearch/statistical-thinking-for-robot-policy-evaluation-from-rigorous-a-b-testing-to-effective-0ae886fbd68d.

[46] NVIDIA et al. "GR00T N1: An Open Foundation Model for Generalist Humanoid Robots". In: *ArXiv Preprint*. Mar. 2025. arXiv: 2503.14734.

[47] Maxime Oquab et al. "Dinov2: Learning robust visual features without supervision". In: *arXiv preprint arXiv:2304.07193* (2023).

[48] Guilherme Penedo et al. "The fineweb datasets: Decanting the web for the finest text data at scale". In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 30811–30849.

[49] Nicholas Pfaff and Peter Werner. *Drake Blender Tools: Importing Drake Simulations into Blender*. https://github.com/nepfaff/drake-blender-tools. 2025.

[50] Physical Intelligence. *openpi: Open-Source Models and Packages for Robotics*. https://github.com/Physical-Intelligence/openpi. GitHub repository. Apache-2.0 License. 2025.

[51] Physical Intelligence et al. $\pi_{0.5}$*: a Vision-Language-Action Model with Open-World Generalization*. 2025. arXiv: 2504.16054 [cs.RO]. URL: https://arxiv.org/abs/2504.16054.

[52] Physical Intelligence et al. $\pi^*_{0.6}$*: a VLA That Learns From Experience*. 2025. arXiv: 2511.14759 [cs.LG]. URL: https://arxiv.org/abs/2511.14759.

[53] Hans-Peter Piepho. "An algorithm for a letter-based representation of all-pairwise comparisons". In: *Journal of Computational and Graphical Statistics* 13.2 (2004), pp. 456–466.

[54] Alec Radford et al. "Learning transferable visual models from natural language supervision". In: *International conference on machine learning*. PmLR. 2021, pp. 8748–8763.

[55] Sebastian Raschka. *Build A Large Language Model (From Scratch)*. Manning, 2024. ISBN: 978-1633437166. URL: https://www.manning.com/books/build-a-large-language-model-from-scratch.

[56] Jeff Rasley et al. "DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2020, pp. 3505–3506. DOI: 10.1145/3394486.3406703. URL: https://github.com/deepspeedai/DeepSpeed.

[57] Keisuke Sakaguchi et al. "WinoGrande: An Adversarial Winograd Schema Challenge at Scale". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 8732–8740. URL: https://aaai.org/ojs/index.php/AAAI/article/view/6399.

[58] ServiceNow Research. *Fast-LLM: Accelerating Your LLM Training to Full Speed*. 2024. URL: https://github.com/ServiceNow/Fast-LLM.

[59] Wenzhe Shi et al. "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 1874–1883.

[60] Mohammad Shoeybi et al. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism". In: *arXiv preprint arXiv:1909.08053* (2019).

[61] Mustafa Shukor et al. "SmolVLA: A vision-language-action model for affordable and efficient robotics". In: *arXiv preprint* (2025). arXiv: 2506.01844 [cs.RO].

[62] Oriane Siméoni et al. "Dinov3". In: *arXiv preprint arXiv:2508.10104* (2025).

[63] David Snyder et al. "Is Your Imitation Learning Policy Better Than Mine? Policy Comparison with Near-Optimal Stopping". In: *Proceedings of the Robotics: Science and Systems Conference (RSS) XXI*. 2025.

[64] Angelika M Stefan and Felix D Schönbrodt. "Big little lies: A compendium and simulation of p-hacking strategies". In: *Royal Society Open Science* 10.2 (2023).

[65] TRI LBM Team et al. "A Careful Examination of Large Behavior Models for Multitask Dexterous Manipulation". In: (2025). arXiv: 2507.05331 [cs.RO]. URL: https://arxiv.org/abs/2507.05331.

[66] TRI LBM Team et al. *LBM Eval: A Simulation Benchmark for Large Behavior Model Policies*. https://github.com/ToyotaResearchInstitute/lbm_eval. Toyota Research Institute. Version 1.1.0. 2025.

[67] Team OLMo et al. *2 OLMo 2 Furious*. 2024. arXiv: 2501.00656 [cs.CL]. URL: https://arxiv.org/abs/2501.00656.

[68] Russ Tedrake and the Drake Development Team. *Drake: Model-based design and verification for robotics*. 2019. URL: https://drake.mit.edu.

[69] Haoyang Weng et al. *VLA-Scratch: Modular, Performant and Efficient Stack*. https://github.com/EGalahad/vla-scratch. GitHub repository. 2026.

[70] Luis Wiedmann and Juyoung Suk. *nanoVLM: The simplest repository to train your VLM in pure PyTorch*. https://github.com/huggingface/nanoVLM. 2024.

[71] Bin Xie et al. "Dexbotic: Open-source vision-language-action toolbox". In: *arXiv preprint arXiv:2510.23511* (2025).

[72] Seonghyeon Ye et al. *World Action Models are Zero-shot Policies*. 2026. arXiv: 2602.15922 [cs.RO]. URL: https://arxiv.org/abs/2602.15922.

[73] Rowan Zellers et al. "HellaSwag: Can a Machine Really Finish Your Sentence?" In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, 2019, pp. 4791–4800. DOI: 10.18653/v1/P19-1472. URL: https://aclanthology.org/P19-1472.

[74] Xiaohua Zhai et al. "Sigmoid loss for language image pre-training". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2023, pp. 11975–11986.

[75] Tony Z. Zhao et al. "Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware". In: *Proceedings of Robotics: Science and Systems*. Daegu, Republic of Korea, July 2023. DOI: 10.15607/RSS.2023.XIX.016.

[76] Yi Zhou et al. "On the Continuity of Rotation Representations in Neural Networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019, pp. 5745–5753.

Appendix

A VLA Foundry – Detailed Reference

This appendix expands on the description of VLA Foundry in Section 3. Section A.1 covers the core framework internals (configuration system, registry, dataloading, dataset mixing, and preprocessing) with representative code and configuration snippets alongside each description, and Section A.2 covers robotics-specific utilities (normalization, action representations, sampling windows, and proprioception).

A.1 Framework Internals

A.1.1 Model Training Loop

Training in VLA Foundry is done in a single training loop that is shared across every model stage in the pipeline: LLM pretraining, VLM pretraining, and VLA fine-tuning. VLA Foundry takes a data-centered approach. It expresses training budgets as a number of samples instead of a number of steps, so that runs at different batch sizes or GPU counts remain directly comparable.

The training loop is deliberately model-agnostic. Unpacking a batch into inputs and targets, masking the loss, and reducing model outputs to a scalar is attached to each model class rather than baked into the loop, so the same training pathway drives an LLM learning from raw text, a VLM learning from image–caption pairs, and a VLA learning to denoise actions. The loop composes cleanly with the distributed training primitives users expect: FSDP with optional CPU offloading, mixed-precision execution, gradient accumulation, gradient checkpointing, `torch.compile`, and exponential moving average of weights.

A.1.2 Config System and Argument Parsing

We use `draccus` [42] to parse arguments. At the most basic level, arguments are supplied as command-line flags. Optionally, to avoid manually typing many flags, users can supply `–config_path` and point to a YAML file with nested parameters, and YAML files themselves can be nested with the `include` keyword.

Configurations follow a three-level precedence: command-line arguments override YAML preset files, which override dataclass defaults. Parameters are organized hierarchically and any field can be overridden at arbitrary nesting depth. To give a concrete example, consider the command `python main.py –config_path config.yaml –model.hidden_dim=1024`. Here, the order of priority would be (1) the CLI flag `–model.hidden_dim=1024`, (2) the `hidden_dim` in `config.yaml`, (3) [if it exists] the `hidden_dim` in any nested `include` file, (4) the default values defined within the code.

A `–resolve_configs` flag prints the complete merged configuration and generates a YAML, letting users verify exactly what will run. A minimal example of a nested YAML configuration, invoked with `python main.py –config_path config.yaml`, is shown below.

```yaml
# config.yaml
model:
  include: vla_foundry/config_presets/models/transformer_11m.yaml
  hidden_dim: 2048  # overrides the preset value
  vit:
    include: vla_foundry/config_presets/models/vit_paligemma.yaml
hparams:
  lr: 1e-4
  global_batch_size: 256
  per_gpu_batch_size: 8
  precision: amp_bfloat16
```

A.1.3 Dataset and Model Registry

The `–model` and `–dataset` arguments have a special keyword called `type`. The `–model.type` and `–data.type` arguments select which parameter class and model or data pipeline to instantiate at runtime. Each model is registered via a `@register_model` decorator on its factory function, and the same pattern applies to datasets.

This means that adding a new model to VLA Foundry requires only two things: a frozen dataclass defining its hyperparameters, and a factory function decorated with `@register_model`. No central configuration file needs to be modified. At runtime, `create_model()` looks up the registry by `model.type` and dispatches to the correct factory. Each model selects a `BatchHandler` – typically one of the shared handlers defined per modality or model family (LLM, VLM, and DP-VLA) – which encapsulates batching, loss construction, and output reduction, and keeps the main training loop model-agnostic. Entirely new training paradigms can register an additional handler via `@register_batch_handler`. Registering a new model looks like:

```
# This can be accessed with `--model.type = diffusion_policy`
class DiffusionPolicy(self, [...]):
    # Implementation here

@register_model("diffusion_policy")
def create_diffusion_policy(model_params: ModelParams, load_pretrained: bool = True):
    vision_language_backbone = get_vision_language_backbone(
        model_params.vision_language_backbone, load_pretrained
    )
    transformer = create_model(model_params.transformer, load_pretrained)
    noise_scheduler = create_noise_scheduler(model_params)
    return DiffusionPolicy(model_params, vision_language_backbone, transformer, noise_scheduler)
```

A.1.4 Dataloading

We use WebDatasets for dataloading and store the data in tar shards. Within each tar file, each sample is distinguished by its unique prefix. The structure of the directory is shown below. This structure is designed to be extensible, where new fields can be added easily if necessary. For instance, if we want to include depth images, we can do this by adding `unique_name_1_depth1.jpg`. The flexibility of our data format also allows us to extend to other modalities such as video.

```
dataset_name/
├── manifest.jsonl
├── shard_00000000.tar
│   ├── unique_name_1_camera1.jpg
│   ├── unique_name_1_camera2.jpg
│   ├── unique_name_1_meta.json
│   ├── unique_name_1_actions.npz
│   ├── unique_name_2_camera1.jpg
│   └── ...
├── shard_00000001.tar
├── shard_00000002.tar
└── ...
```

The data processing pipeline steps are defined separately for each modality. Currently, the data modalities supported are text, caption (text+image), and robotics (text+image+action). The steps of the WebDataset pipeline are defined sequentially, and notably support both WebDataset built-in functions (e.g., `wds.split_by_node`) and user-defined functions that can be composed freely. This is especially useful for the robotics processing detailed in Section A.2. An example pipeline for an image-caption dataset follows; the same composition pattern is used for text and robotics pipelines, with different per-modality steps.

```
pipeline = [
    wds.SimpleShardList(datastring),
    deterministic_shuffle(
        bufsize=self.data_params.shuffle_buffer_size,
        initial=self.data_params.shuffle_initial,
        seed=self.data_params.seed,
        epoch=checkpoint_num,
    ),
```

```
    wds.split_by_node,
    wds.split_by_worker,
    wds.tarfile_to_samples(handler=log_and_continue),
    wds.decode("pilrgb", handler=log_and_continue),
    wds.select(filter_no_caption_or_no_image),
    wds.map(
        lambda sample: self.augmentations.apply_transforms(sample),
        handler=log_and_continue,
    ),
    wds.rename(image="jpg;png;jpeg;webp", text="txt"),
    wds.map(lambda sample: {**sample, "text": "<image> " + sample["text"]}),
    wds.batched(self.batch_size, partial=False),
    wds.map(
        lambda sample: self.processor(
            images=sample["image"],
            text=sample["text"],
            return_tensors="pt",
            padding="max_length",
            padding_side="right",
            max_length=self.data_params.seq_len + 1,
        ),
        handler=log_and_continue,
    ),
    wds.map(
        lambda sample: {
            "input_ids": sample["input_ids"],
            "attention_mask": sample["attention_mask"],
            "pixel_values": sample["pixel_values"],
        }
    ),
]
return pipeline
```

A.1.5 Dataset Mixing

VLA Foundry natively supports dataset mixing with command-line arguments. By default, the dataset-related arguments are lists, which means that supporting multiple datasets is as simple as adding elements to the list. Of special note is the `--data.dataset_weighting` parameter, which handles the batch balancing ratios; a 1:2:1 weighting corresponds to 25%/50%/25% of each batch drawn from the respective datasets. An example YAML snippet that mixes three robotics datasets with a 1:2:1 weighting is shown below.

```
dataset_manifest:
  - tasks_processed/BimanualPlaceAppleFromBowlIntoBin/shards/manifest.jsonl
  - tasks_processed/BimanualPlaceFruitFromBowlIntoBin/shards/manifest.jsonl
  - tasks_processed/BimanualPutRedBellPepperInBin/shards/manifest.jsonl
dataset_statistics:
  - tasks_processed/BimanualPlaceAppleFromBowlIntoBin/shards/stats.json
  - tasks_processed/BimanualPlaceFruitFromBowlIntoBin/shards/stats.json
  - tasks_processed/BimanualPutRedBellPepperInBin/shards/stats.json
dataset_modality:
  - robotics
  - robotics
  - robotics
dataset_weighting:
  - 1.0
```

```
  - 2.0
  - 1.0
```

A.1.6 Preprocessing and Manifests

VLA Foundry has custom scripts to convert raw datasets to the WebDataset tar shards described above. As noted in Section A.1.4, we currently support text, image-caption, and robotics datasets. Text preprocessing reads parquet files (typically stored on S3) and emits one JSON sample per row. Image-caption preprocessing takes a URL list and downloads image–text pairs via `img2dataset`. Utilities are also shipped for fetching upstream data from Hugging Face Hub and from HTTP directory listings into the intermediate storage consumed by these scripts. Robotics raw data can come from any source (simulation logs, real-robot recordings, etc.), as long as a converter knows how to read it and produce a standardized output; for this release, we provide converters from LeRobot and from the Spartan format used in `lbm_eval`. These robotics converters all share the same entry point and follow the same logic, so adding a new one amounts to creating a new class that inherits from `BaseRoboticsConverter` and filling in the necessary methods such as `discover_cameras`.

We use `ray` [44] to parallelize data preprocessing. Under the `ray` framework, there is a head node that orchestrates the jobs and several worker nodes that each run a small independent job. For robotics datasets, this is done in multiple stages. First, it creates a `frames` folder in the output directory, where each sample is its own unique tar file. Next, it creates an `episodes` folder, where the sample tar files are grouped together by episode. Finally, it creates a `shards` folder, where sample tar files are grouped together randomly. This `shards` folder is what is ultimately used for training. Within the `shards` folder, there is a `manifest.json` which contains an overview of the shards; an example follows.

```
{"shard": "00000000", "num_sequences": 1024}
{"shard": "00000001", "num_sequences": 1024}
{"shard": "00000002", "num_sequences": 1024}
{"shard": "00000003", "num_sequences": 488}
```

Robotics datasets additionally have a `stats.json` within the `shards` folder, which contains statistics computed across all samples of the dataset. This computation requires worker nodes to first store local statistics in node memory, then communicate and gather across different nodes. For internal runs, this was tested on AWS EC2 with 60 nodes of `i4i.4xlarge`, but we have tested it locally as well.

A.2 Robotics-specific Details

A.2.1 Normalization

Actions and proprioceptive states are normalized during dataloading time and denormalized at inference time. Normalization is handled by a `RoboticsNormalizer` class, which supports four normalization methods: standard deviation, min-max, and two percentile-based variants (`percentile_1_99` and `percentile_5_95`). The choice of percentile-based normalization is useful for action fields that contain outliers, as it avoids compressing the bulk of the distribution to a narrow band. Statistics are precomputed across the full dataset during preprocessing and stored in a `stats.json` file alongside each dataset shard.

Normalization scope Normalization can be applied at two scopes. In *global* scope, the scalars `mean` and `scale` are applied uniformly across all timesteps in a sequence. In *per-timestep* scope, each timestep within the action window has its own mean and scale derived from statistics computed at that relative offset in the trajectory. Per-timestep normalization is particularly useful for relative action representations, where the distribution of predicted displacements can vary considerably between early and late steps of the prediction horizon. When working with cropped sequences (see Section A.2.3), per-timestep statistics are aligned to the anchor timestep so that indices into the statistics tensor correspond correctly to the tensor's time axis.

Merging statistics When training on multiple datasets simultaneously (Section A.1.5), users may wish to use the joint distribution across all datasets rather than any individual one. Since datasets are processed individually with their own per-dataset `stats.json` files, we support merging multiple `stats.json` files together. Means are computed as sample-count-weighted averages. Standard deviations are merged using the law of

total variance, $\sigma^2_{\text{overall}} = \mathbb{E}[\sigma_i^2]$. Min and max statistics are obtained as element-wise minima and maxima across datasets. Percentiles cannot be merged exactly from summary statistics alone; instead, each dataset retains a serialized t-digest sketch [20] during preprocessing, and the sketches are merged at training time to recover approximate percentiles of the pooled distribution. All statistics are computed and merged per action-space dimension, and optionally per timestep within the prediction window when using per-timestep normalization scope.

A.2.2 Absolute vs. Relative Actions

VLA Foundry supports both absolute and relative action representations, which are stored as separate fields in the dataset. Absolute actions are poses expressed in the world frame (e.g., end-effector XYZ position and 6D rotation). Relative actions are computed with respect to the robot's actual pose at the anchor timestep, i.e., the frame at which a prediction is made.

Formally, let $T_{\text{ref}} \in SE(3)$ denote the actual end-effector pose at the anchor timestep and $T_t \in SE(3)$ the action pose at future timestep t. The relative action is defined as

$$T_t^{\text{rel}} = T_{\text{ref}}^{-1} \cdot T_t,$$

where the product is the standard $SE(3)$ group operation. Rotations are represented in the 6D continuous rotation format [76] throughout, with conversion to and from $SO(3)$ matrices performed via Gram–Schmidt orthogonalization. VLA Foundry's preprocessing scripts generate both absolute and relative fields given configurations defining which fields form poses, and the practitioner selects which to use via the `--action_fields` configuration during training.

A.2.3 Past/Future Action Window

During dataset preprocessing, each training sample is constructed around an *anchor timestep* t within an episode. The low-dimensional window centered at t spans $[t - N_{\text{past}}, t + N_{\text{future}}]$, where N_{past} and N_{future} are configurable preprocessing parameters (`past_lowdim_steps` and `future_lowdim_steps`). This produces a tensor of $N_{\text{past}} + 1 + N_{\text{future}}$ timesteps per sample. Including past timesteps allows the model to condition on recent action history and proprioceptive context; predicting multiple future timesteps allows for temporal action chunking [75].

At episode boundaries, sequences are padded using a configurable padding strategy (copy, zero, or reflect). To avoid degenerate samples with excessive padding, samples whose required padding exceeds configurable thresholds (`max_padding_left`, `max_padding_right`) are discarded during preprocessing. The anchor timestep's position within the cropped window is stored in sample metadata as `anchor_relative_idx`, enabling downstream code to correctly align per-timestep normalization statistics and to separate past from future timesteps without re-parsing raw episode indices. Notably, the preprocessing past/future action window does not need to be identical to the past/future values used during training. This allows users to specify a larger window during preprocessing time, then work with a truncated subwindow during training.

A.2.4 Proprioception

Proprioceptive state is specified via a separate `--proprioception_fields` parameter, distinct from `--action_fields`. Typical proprioception fields include joint positions, joint velocities, and actual end-effector poses (XYZ and 6D rotation). During batch construction, the fields listed in `proprioception_fields` are each extracted, normalized, and concatenated along the feature dimension to form a single `proprioception` tensor of shape $[B, T_{\text{prop}}, D_{\text{prop}}]$. A key design difference from actions is that proprioception uses only the *past and current* timesteps within the window (i.e., indices $[0, t_{\text{anchor}}]$), whereas actions span the full past-and-future window. This reflects the causal structure of the problem: past proprioception is observed history available to the policy, while future proprioception is not available at inference time.

B Links to Checkpoints and Additional Resources

Project website: https://tri-ml.github.io/vla_foundry

Project code: https://github.com/TRI-ML/vla_foundry

Model weights: https://huggingface.co/collections/TRI-ML/vla-foundry

C LLM-VLM-VLA Details

C.1 Model Sizes

Table 3 details the different module sizes of the two architectures used in this report.

Table 3 Parameter count (billions). Embedding = VLM input embedding + output projection (lm_head) + ViT patch/position embed. Non-embed = LLM + Vision + Action head.

Model	Embedding	LLM	Vision	Action head	Total	Non-embed
FOUNDRY-VLA-1.7B	0.20	1.23	0.09	0.33	1.85	1.65
FOUNDRY-QWEN3VLA-2.1B	0.62	1.41	0.41	0.31	2.75	2.13

C.2 LLM Benchmarks

The following short descriptions of the benchmarks below are borrowed from [33].

- HellaSwag [73] (10,042 examples) is a 4-way multiple choice commonsense reasoning dataset, where the model is required to understand implicit context and common knowledge in order to correctly select the continuation to a context. HellaSwag is distributed under the MIT license as indicated in https://github.com/rowanz/hellaswag/blob/master/LICENSE.
- MMLU [24] (14,042 examples) is a 4-way multiple choice question answering dataset that covers 57 different domains and tasks, evaluating both world knowledge and problem solving capabilities. MMLU is distributed under the MIT license as indicated in https://github.com/hendrycks/test/blob/master/LICENSE.
- The ARC easy (2,376 examples) and ARC challenge (1,172 examples) datasets [15] contain four-way multiple choice questions taken from grade 3-9 science exams, where questions in the easy dataset require knowledge of basic science, and the challenge questions require some procedural reasoning. are distributed under the Creative Commons Attribution-Sharealike 4.0 International license as indicated in https://allenai.org/data/arc.
- PIQA [8] (1,838 examples) is a binary multiple choice question answering dataset that requires the model to use physical commonsense reasoning to answer correctly. PIQA is distributed under the Academic Free License v. 3.0 as indicated in https://github.com/ybisk/ybisk.github.io/tree/master/piqa.
- The Winogrande [57] (273 examples) is binary multiple choice pronoun resolution task where the model is given a context and asked to determine which entity a pronoun refers to, requiring the model to exhibit commonsense knowledge and contextual understanding. Winogrande is distributed under the Apache 2.0 license as indicated in https://github.com/allenai/winogrande/blob/master/LICENSE.
- OpenBookQA [43] (500 examples) is a 4-way multiple choice question answering dataset that requires the model to use multi-step reasoning and commonsense knowledge. OpenBookQA is distributed under the Apache 2.0 license as indicated in https://github.com/allenai/OpenBookQA/blob/main/LICENSE.
- BoolQ [14] (3,270 examples) is a binary question answering dataset where the model is expected to answer questions about relevant passages. BoolQ is distributed under the Creative Commons Share-Alike 3.0 license as indicated in https://huggingface.co/datasets/google/boolq.

C.3 VLM Benchmark

COCO Captions [11] (5,000 validation examples) is an image captioning dataset where the model is given an image and is required to generate a natural language description capturing the salient objects, actions, and scene

context. Each image is paired with five human-written reference captions, and model outputs are evaluated using standard metrics such as CIDEr and BLEU. COCO Captions annotations are distributed under the Creative Commons Attribution 4.0 International license as indicated in https://cocodataset.org/#termsofuse. The images retain their original Flickr licenses, and use of the images must abide by the Flickr Terms of Use.

C.4 Image Encoding Details

Figure 10 shows the image encoding operation with an explicit representation of the "pixel-shuffle" pooling operation. Note that "pixel-shuffle" is usually the opposite operation [59] used for super-resolution. We label it "unshuffle" in the figure for clarity.

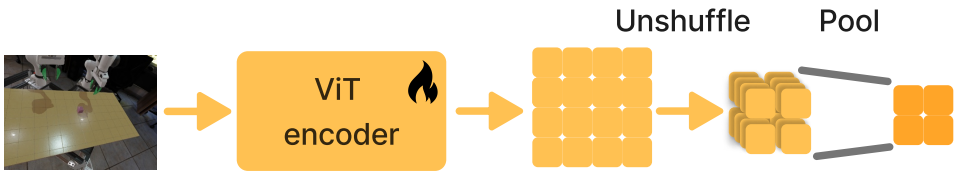


Figure 10 Representation of the pixel-shuffle operation [41] used for patch pooling, reducing the number of tokens passed to the downstream VLM

C.5 Training Parameters

Table 4 shows the main training parameters used to train our different VLA models.

Model	LR	Schedule	Warmup	Total samples	Batch size
Foundry-VLA-1.7B-full	5×10^{-5}	cosine	1,000	102,400,000	1,024
Foundry-VLA-1.7B-sim	5×10^{-5}	cosine	1,000	102,400,000	1,024
Foundry-VLA-1.7B-real	5×10^{-5}	cosine	1,000	102,400,000	1,024
Foundry-VLA-1.7B-ST	5×10^{-5}	cosine	1,000	5,120,000	512
Foundry-VLA-1.7B-FT	5×10^{-6}	cosine	1,000	1,024,000	512
Foundry-VLA-1.7B-FT-sim	5×10^{-6}	cosine	1,000	1,024,000	512
Foundry-Qwen3VLA-2.1B	5×10^{-5}	cosine	1,000	100,000,000	1,024
Foundry-Qwen3VLA-2.1B-ST	5×10^{-5}	cosine	1,000	2,000,000	512
Foundry-Qwen3VLA-2.1B-FT	5×10^{-6}	cosine	1,000	1,024,000	512

Table 4 Training hyperparameters for Foundry VLA model variants. All models use AdamW with cosine learning-rate schedule and 1,000 warmup steps. MT variants train for ∼100M samples at batch 1,024; per-task ST trains for 2–5M samples at batch 512; per-task FT fine-tunes from the MT checkpoint for 1M samples at 10× lower LR.

C.6 VLA Dataset Details

As shown in Tables 5 and 6, our subset of simulation and real data differs from the training split used in [65] to train LBM. While a small number of episodes were dropped during pre-processing, the overall dataset size is slightly larger primarily due to differences in filtering criteria and the inclusion of data previously reserved for validation. Of the internal real and simulated data, the LBM models are trained on the data under column LBM; all other models are trained on the data under column VLA Foundry. Importantly, the multi-task pretrained LBM model is trained on a larger dataset which includes open source OXE [16] data; refer to [65] for further details. Table 7 shows the number of training samples per dataset split used to train VLA Foundry models; the number of samples generated by a single demonstration episode depends on the length of each demonstration and preprocessing configurations such as padding. While the internally collected real and simulated data is largely shared between Foundry-VLA-1.7B, Foundry-Qwen3VLA-2.1B, and LBM, the VLA Foundry models use substantially more finetuning data on the unseen tasks compared to the single task and finetuned versions of LBM, which we do not compare to in this technical report. Instructions

on how to download the tar files used to train the sim data only variants of VLA Foundry models can be found in the released codebase.

Table 5 Dataset overview. Previous work incorrectly categorized the "PushBox" simulation task as a real task.

	LBM		VLA Foundry	
Split	Tasks	Episodes	Tasks	Episodes
Real	362	46,063	361	47,068
Sim	41	7,348	42	7,548
Total	403	53,411	403	54,616

Table 6 Simulation evaluation tasks. Seen tasks are used in multitask training. Unseen tasks are held out.

		Episodes	
#	Task	LBM	VLA Foundry
Seen tasks			
1	BimanualPlaceAppleFromBowlIntoBin	196	200
2	BimanualPlaceFruitFromBowlIntoBin	196	200
3	BimanualPutRedBellPepperInBin	196	200
4	BimanualPutSpatulaOnPlateFromDryingRack	196	200
5	BimanualPutSpatulaOnPlateFromTable	196	200
6	BimanualStackPlatesOnTableFromDryingRack	196	200
7	BimanualStoreCerealBoxUnderShelf	196	200
8	PlaceCupByCoaster	196	200
9	PushCoasterToCenterOfTable	196	200
10	PushCoasterToMug	196	200
11	PutBananaOnSaucer	49	50
12	PutKiwiInCenterOfTable	49	50
13	PutMugOnSaucer	196	200
14	PutSpatulaInUtensilCrock	196	200
15	TurnCupUpsideDown	490	500
16	TurnMugRightsideUp	490	500
Unseen tasks			
17	BimanualPlaceAvocadoFromBowlIntoBin	196	375
18	BimanualPutSpatulaOnPlateFromUtensilCrock	195	400
19	PutMugInCenterOfTable	294	300

D Additional Simulation Evaluation Analysis

In this section, we provide additional results in from the simulation evaluation.

D.1 Comparison of OS and CS Variants of LBM Eval

Due to code changes between the paper submission and the final open-sourcing of the simulation benchmark, the evaluation results may differ slightly from [65]. To provide context, we show the evaluation results here compared to evaluating the models on the original (closed source) simulation benchmark. We observe that the vast majority of the simulation training demonstrations were collected using a version of the simulation much closer to `lbm_eval_cs`.

Table 7 Training data samples in VLA Foundry.

Split	Tasks	Episodes	Training samples
Real	361	47,068	17,156,497
Sim	42	7,548	1,647,049
Total	403	54,616	18,803,546

D.2 Comparison of FOUNDRY-VLA-1.7B and FOUNDRY-QWEN3VLA-2.1B

We also provide direct comparisons of FOUNDRY-QWEN3VLA-2.1B and FOUNDRY-VLA-1.7B models in Figure 7a and **??**.

E Additional Qualitative Simulation Figures

Figure 14 provides example snapshots of randomly sampled failure episodes from the FOUNDRY-QWEN3VLA-2.1B checkpoint as a companion to Figure 6. Figure 16 gives an example of raw sensor measurements from `lbm_eval_oss`. Figure 15 shows temporal examples of successful and non-successful rollouts for qualitative purposes.

Figure 11 Comparison of checkpoints on `lbm_eval_oss` (OSS) and `lbm_eval_cs` (CS). In aggregate, the performance of both the FOUNDRY-VLA-1.7B single task checkpoints and the FOUNDRY-QWEN3VLA-2.1B multi task checkpoint is weaker on the open source version of the benchmark, which can be considered a distribution shifted version of the closed source version.

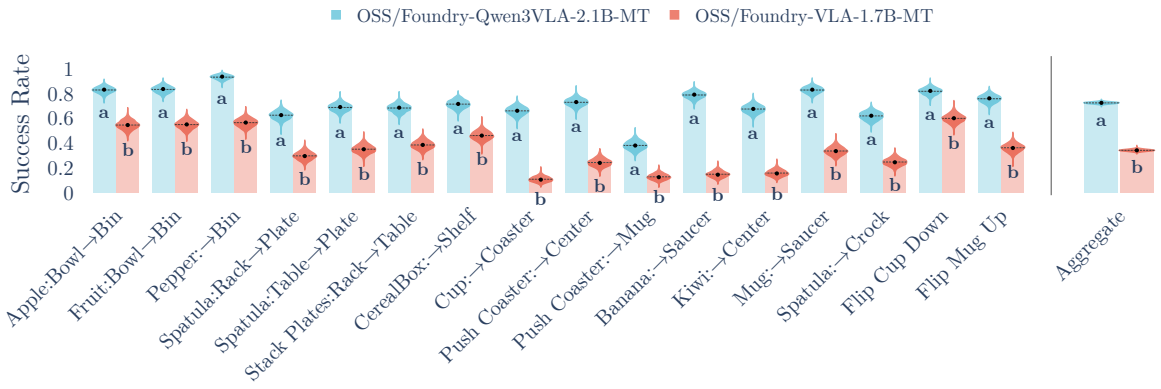
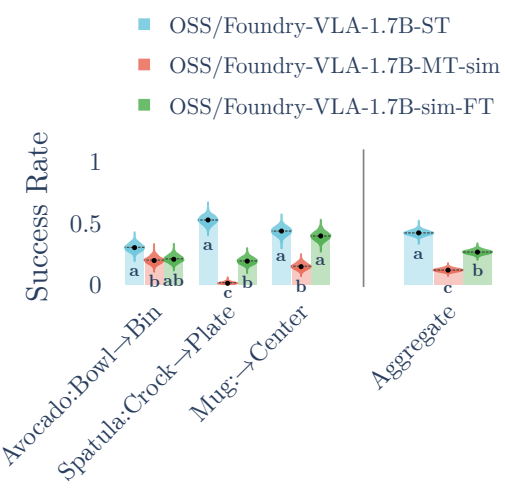
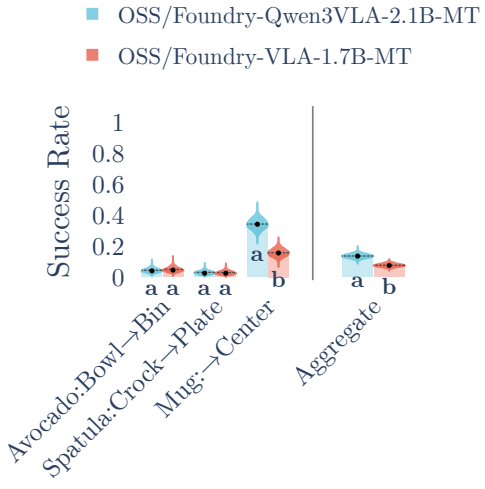


Figure 12 Comparison of FOUNDRY-QWEN3VLA-2.1B and FOUNDRY-VLA-1.7B MT models (seen tasks). The FOUNDRY-QWEN3VLA-2.1B out performs than FOUNDRY-VLA-1.7B in aggregate over the seen tasks.



(a) Foundry-VLA-1.7B-sim performance on unseen tasks



(b) Comparison of Foundry-Qwen3VLA-2.1B and Foundry-VLA-1.7B on unseen tasks.

Figure 13 Simulation results on `lbm_eval_oss` (unseen tasks). All models demonstrate some non-zero success rates 0-shot.

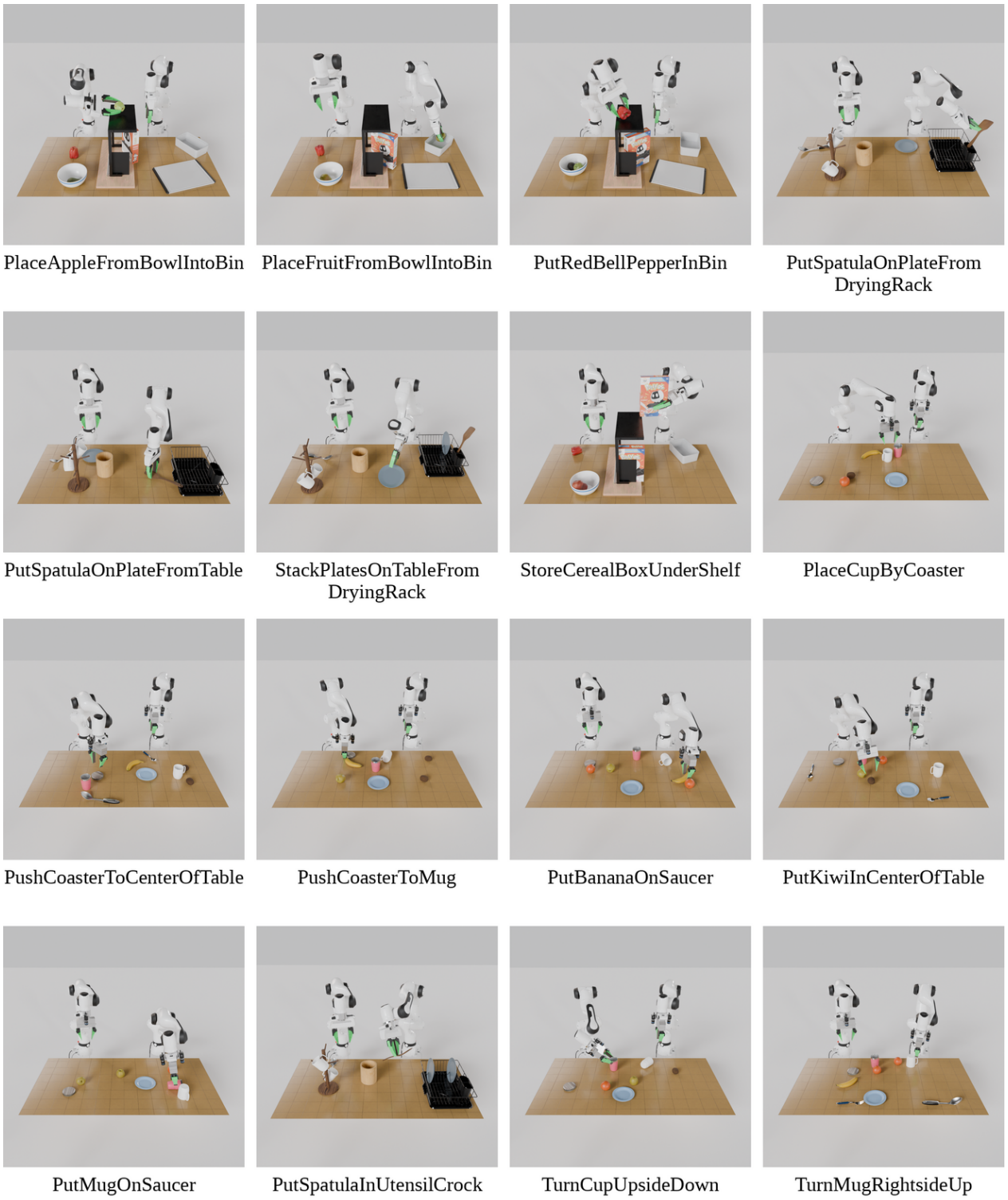


Figure 14 Overview of seen simulation evaluation tasks (failures). Here, we show a single still from about the midpoint of a failed rollout from FOUNDRY-QWEN3VLA-2.1B. Videos of selected successful and failed rollouts can be found at https://tri-ml.github.io/vla_foundry. Companion plot to Figure 6.

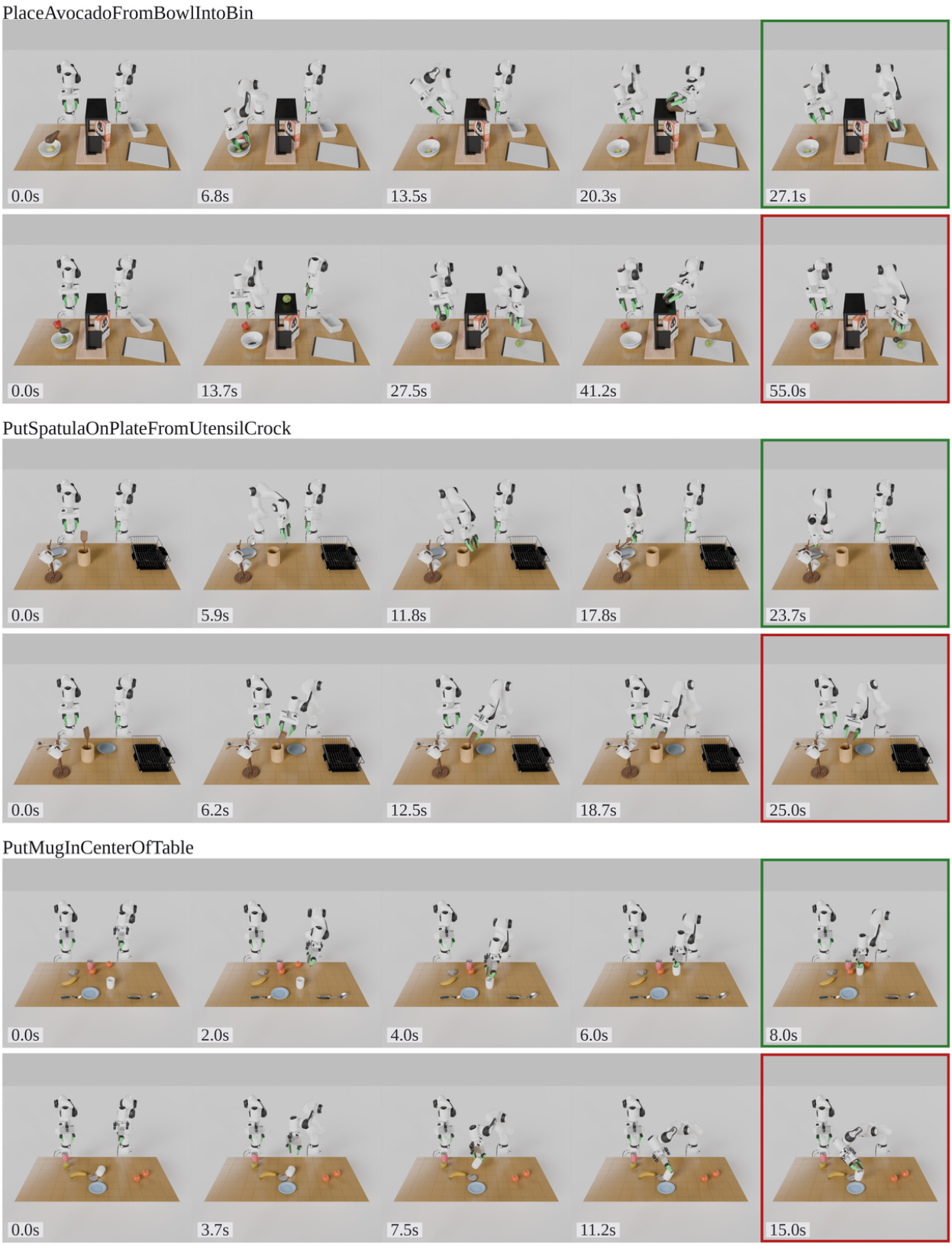


Figure 15 Example of success and failure rollouts for FOUNDRY-QWEN3VLA-2.1B on tasks unseen at training time. For each task, the top row is a success and the bottom row is a failure. The timeout for each task depends on benchmark definitions of `lbm_eval_oss`.

Figure 16 Example of sensor measurements at inference time. Image captured at approximately the same timestamp as the PlaceAppleFromBowlIntoBin render in Figure 6. The images are then post processed further for input to the VLA models such as Foundry-Qwen3VLA-2.1B and Foundry-VLA-1.7B. While some simulation stations include an extra wrist camera per arm, Foundry-Qwen3VLA-2.1B and Foundry-VLA-1.7B use only the four shared cameras for VLA training and inference. Refer to [65] for further details on the simulation stations.