PRIMITIVEVLA: LEARNING REUSABLE MOTION PRIMITIVES FOR EFFICIENT AND GENERALIZABLE ROBOTIC MANIPULATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Vision-Language-Action (VLA) models offer a promising route to general-purpose robotic policies, yet downstream adaptation remains data-intensive, with limited transfer to new task compositions. Direct instruction-to-control fine-tuning treats multi-stage demonstrations as complete task trajectories, offering little structure for reusing recurring interaction patterns. We propose PrimitiveVLA, a primitive-centric Disassembly-and-Assembly framework. During fine-tuning, a vision-language model (VLM) infers primitive sequences, while a large language model (LLM) localizes their temporal boundaries to construct primitive-aligned samples. During inference, the VLM generates a primitive flow; the VLA executes it in a closed loop with an LLM-generated causal switching program. Unified Multimodal Representation (UMR) maps instance-specific descriptions to unified primitive instructions while grounding objects and targets through mask-augmented observations. Across Libero, RLBench, and real-robot experiments, PrimitiveVLA improves data efficiency and zero-shot transfer to unseen task compositions and long-horizon tasks while maintaining or improving in-distribution performance.

1 INTRODUCTION

The convergence of Large Language Models (LLMs) and robotic control has spurred the rise of Vision-Language-Action (VLA) models. By projecting visual observations, instructions, and actions into a unified latent space, models like RT-2 (Zitkovich et al., 2023) and OpenVLA (Kim et al., 2025b) demonstrate impressive capabilities in open-world manipulation. Trained on internet-scale datasets (e.g., Open X-Embodiment (O'Neill et al., 2024) and DROID (Khazatsky et al., 2024)), these generalist policies promise a "foundation model" moment for embodied AI.

However, adapting pre-trained VLAs to downstream domains via fine-tuning remains challenging. Despite vast pre-training, VLAs often exhibit brittle performance with weak cross-task generalization. Benchmarks like AGNOSTOS (Zhou et al., 2025) reveal a significant struggle with novel object-skill combinations. We argue that limited data and the organization of fine-tuning supervision both contribute to this challenge. As illustrated in Figure 1(a), the conventional **Direct Instruction-to-Control Mapping** paradigm treats each multi-stage demonstration as a complete task-level trajectory, so distinct physical motion patterns are mixed during fine-tuning. By mapping high-level instructions directly to low-level control, this formulation couples shared motion patterns to task-specific context without explicitly aligning them across tasks, leaving cross-task reuse to emerge implicitly. Recent models such as $\pi_{0.7}$ (Physical Intelligence et al., 2026) incorporate semantic subtask annotations during training and can generate subtask instructions during inference. While these annotations describe semantic subtasks, our focus is on explicitly aligning motion segments from different tasks under shared primitive categories.

To this end, we introduce the **Primitive-Centric Disassembly-and-Assembly** paradigm, which disassembles task demonstrations into reusable primitive-aligned segments. For example, pulling a cabinet and pulling a drawer can provide supervision for the same *Pull* primitive. Grouping segments by primitive type reveals more coherent motion structure than grouping them by task, as reflected by the silhouette scores in Figure 1(b) (Appendix D.1). These segments serve as independent samples for VLA fine-tuning. At inference, a planner generates an ordered primitive execution flow, and the VLA

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

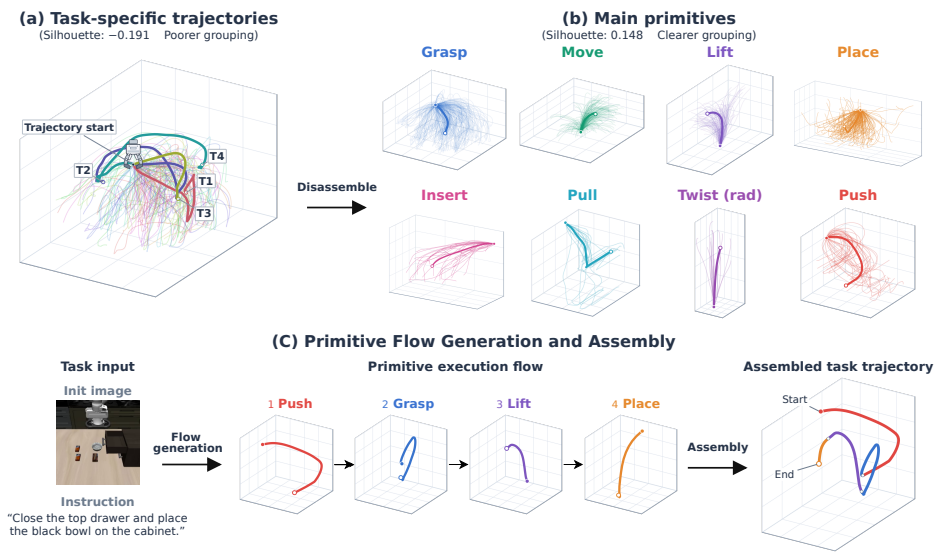


Figure 1: **Direct instruction-to-control mapping vs. Primitive-Centric Disassembly-and-Assembly.** (a) Task-level grouping of trajectories from the Libero-90 training set mixes distinct physical motion patterns, yielding a silhouette score of -0.191 and poorer grouping. (b) Disassembly groups the same trajectories by primitive type, revealing recurring motion patterns and clearer grouping, with the silhouette score increasing to 0.148. (c) During inference, a planner generates an ordered primitive execution flow from the task input, which the VLA executes in closed loop to assemble a complete task trajectory. The full action-space calculation is provided in Appendix D.1.

executes the flow in a closed loop to assemble a complete task trajectory (Figure 1(c)). By exposing recurring motion patterns across tasks, this paradigm supports **compositional generalization** through primitive reuse in task compositions absent from fine-tuning, and **long-horizon execution** through their ordered assembly into longer multi-stage flows.

Realizing this paradigm requires satisfying three practical requirements:

1. **Reliable primitive-level data construction:** Public datasets primarily provide task-level demonstrations, so primitive sequences and their temporal boundaries must be inferred accurately to construct valid fine-tuning samples.
2. **Effective sharing with execution grounding:** Task-specific descriptions vary across demonstrations of the same primitive, while the object, target, and execution attributes remain essential for action selection. The representation should therefore share primitive semantics across tasks without discarding execution-relevant information.
3. **Robust hierarchical execution:** Inference must produce an ordered primitive flow, and causal switching must detect completion on time. Early transitions can interrupt unfinished primitives, whereas late transitions can stall execution or disturb the environment.

PrimitiveVLA implements this paradigm with an automated primitive disassembly pipeline, Unified Multimodal Representation (UMR), and a primitive assembly controller. During fine-tuning, a vision-language model (VLM) infers each task's primitive sequence, and an LLM-generated state/action program localizes the temporal boundaries of the inferred primitives (Sec. 4.2). UMR then maps each retained segment to a unified primitive instruction with task-critical execution attributes and a mask-augmented observation that preserves object and target grounding (Sec. 4.1). These transformed segments serve as independent VLA fine-tuning samples. During inference, a VLM planner generates the primitive execution flow. UMR transforms the current primitive instruction and observation using the same interface before the VLA predicts actions. A causal switching program then detects primitive completion and advances execution to the next primitive (Sec. 4.3).

Our contributions are threefold: (1) we introduce a **Primitive-Centric Disassembly-and-Assembly** paradigm that organizes VLA fine-tuning around reusable primitive-aligned segments and composes learned primitives during inference; (2) we develop **PrimitiveVLA**, which combines automated

primitive-sequence reasoning and temporal boundary segmentation for data construction, UMR for semantic sharing and visual grounding, and causal primitive switching for hierarchical execution; and (3) we evaluate the framework across multiple VLA backbones, simulation benchmarks, and physical-robot experiments, showing improvements in **data efficiency, compositional generalization, and multi-stage execution** while maintaining or improving in-distribution performance.

2 Related Work

2.1 Vision-Language-Action Models

Vision-Language-Action (VLA) models have advanced generalist robot learning. Beyond the RT series (Brohan et al., 2022; Zitkovich et al., 2023) and VIMA (Jiang et al., 2022), open-weights models include OpenVLA (Kim et al., 2025b), Octo (Ghosh et al., 2024), and RoboFlamingo (Li et al., 2023). Diffusion Policy (Chi et al., 2025) and RDT (Liu et al., 2024) explore diffusion-based action generation, while VideoVLA (Shen et al., 2025) and CoT-VLA (Zhao et al., 2025) incorporate visual prediction into control. Other advances include the fine-tuning recipe of OpenVLA-OFT (Kim et al., 2025a) and flow-matching action generation in the π_0 series (Black et al., 2024; 2025; Physical Intelligence et al., 2026). Our work focuses on the organization of fine-tuning supervision. PrimitiveVLA segments demonstrations into shared motion primitives and aligns their representations across tasks, pooling supervision for recurring motions while preserving execution grounding.

2.2 Primitives and Hierarchical Robot Learning

Hierarchical control decomposes tasks into modular skills, from options (Sutton et al., 1999) and DMPs (Ijspeert et al., 2013) to learned latent representations. Continuous latent models such as SPiRL (Pertsch et al., 2021) and OPAL (Ajay et al., 2021), together with discretized representations such as BeT (Shafiullah et al., 2022), VQ-BeT (Lee et al., 2024), and UAE (Zheng et al., 2025a), encode behavior into compact representations. Our method instead uses explicitly defined motion primitives to organize cross-task VLA supervision.

Language-model-based methods such as SayCan (Ichter et al., 2023) and Code as Policies (Liang et al., 2022) connect semantic reasoning with skill selection or control-code generation. Other approaches explore decomposition for robot learning and control (Zhang et al., 2024a; Duan et al., 2025; Xu et al., 2025). Recently, $\pi_{0.7}$ (Physical Intelligence et al., 2026) combines semantic subtask supervision with high-level instruction generation and world-model-generated subgoal images. PrimitiveVLA explicitly aligns cross-task motion segments under shared primitive categories, using state/action-based boundary localization and unified multimodal representations for VLA fine-tuning.

3 Preliminaries

3.1 Problem Formulation

Base Formulation. We formulate robotic manipulation as a sequential decision-making problem. At t, the policy receives RGB observations o_t and proprioceptive state s_t, and predicts a 7-dimensional action a_t comprising a 6-DoF delta pose and a gripper command. The available observation modalities depend on the backbone. A demonstration is represented as $\tau = \{(o_t, s_t, a_t)\}_{t=1}^{T}$.

Direct Instruction-to-Control Mapping vs. Primitive-Centric Disassembly-and-Assembly. Standard fine-tuning conditions the policy on a high-level task instruction l:

$$\mathcal{L}_{\text{task}} = \mathbb{E}_{(o,s,a,l)\sim\mathcal{D}_{\text{task}}} \left[\ell_{\text{base}}(\theta; o, s, a, l)\right]. \tag{1}$$

This couples shared motion patterns to task-specific context without aligning them across tasks. Our paradigm instead disassembles each trajectory into primitive-aligned segments and applies UMR to construct $\mathcal{D}_{\text{prim}}$, pairing unified primitive instructions c with mask-augmented observations $\tilde{o}$:

$$\mathcal{L}_{\text{prim}} = \mathbb{E}_{(\tilde{o},s,a,c)\sim\mathcal{D}_{\text{prim}}} \left[\ell_{\text{base}}(\theta; \tilde{o}, s, a, c)\right]. \tag{2}$$

Both objectives retain the backbone-specific loss ℓ_{base} and per-sample sampling rule, while segmentation changes the sample pool from complete trajectories to primitive-aligned segments. This formulation pools cross-task supervision under shared primitive categories to support their later composition.

Table 1: The fixed set of 11 primitives supplied to the VLM for every dataset. Each primitive is specified by a recurring motion/contact structure and a detectable completion condition.

Category	Primitive	Kinematic Definition
Spatial Transport	**Grasp**	Approaches, seizes target, and performs preliminary lift.
	Place	Descends, releases object, and retreats vertically.
	Lift	Vertical translation ($+z$) while holding an object.
	Move	Planar translation (xy) maintaining object grasp.
Contact & Interaction	**Push**	Slides object on surface to target without grasping.
	Pull	Drags or retracts object towards target position.
	Insert	Aligns and inserts into a constrained slot.
	Press	Applies downward force on an object or surface.
Orientation	**Twist**	Rotates gripper (roll) for rotary mechanisms.
	Tilt	Reorients effector (pitch/yaw) to alter pose.
	Rotate	Follows articulation trajectory (e.g., lid).

3.2 Composable Primitives

To structure physical interactions, we manually define a fixed primitive set $\mathcal{C}$ of 11 elements (e.g., *Grasp*, *Push*, and *Twist*), grouped by spatial transport, contact & interaction, and orientation change (Table 1). A pattern is included when it (1) recurs across tasks, (2) has an interpretable motion or contact structure, (3) admits a completion condition detectable from robot states/actions, and (4) composes with other patterns into complete tasks. For example, *Grasp*, *Move*, and *Place* separate approach/closure/initial lifting, held-object transport, and descent/release/withdrawal. The same $\mathcal{C}$ is provided to the VLM for every dataset and remains unchanged throughout the current experiments. The set covers the standard-gripper manipulation tasks evaluated here (Appendix B.1).

4 Methodology

PrimitiveVLA implements the Disassembly-and-Assembly paradigm in two phases (Fig. 2). Fine-tuning disassembles task demonstrations into primitive-level samples, while inference assembles these primitives into complete tasks. During fine-tuning, a VLM infers the ordered primitive sequence and an LLM-generated program localizes the temporal boundary of each primitive (Sec. 4.2). Unified Multimodal Representation (UMR) then converts each segment into a unified primitive instruction, retaining execution attributes when needed, and a mask-augmented observation for VLA fine-tuning (Sec. 4.1). During inference, a VLM generates the task-specific primitive flow and an LLM generates the switching program (Sec. 4.3). In closed-loop execution, UMR forms the input for the current primitive, while causal switching advances the flow when that primitive is complete. Because UMR provides a shared interface for both phases, we introduce it first.

4.1 Unified Multimodal Representation (UMR)

UMR maps a primitive instruction and its RGB observation to a unified instruction and a mask-augmented observation:

$$\mathrm{UMR}(o_i, l_i^{\mathrm{prim}}) = \left(\tilde{o}_i = \mathrm{Overlay}(o_i, M_i),\ c_i = \Gamma(l_i^{\mathrm{prim}})\right), \tag{3}$$

where l_i^{prim} is an instance-specific primitive instruction, Γ maps it to a unified instruction c_i, and M_i grounds the referenced object or target. Spatial, directional, and terminal attributes are retained only when needed to distinguish the intended behavior. $\mathrm{Overlay}$ adds a colored mask while preserving the surrounding RGB scene.

Semantic unification. For all samples of primitive type p_i, Γ maps instance-specific language to a shared primitive instruction while retaining task-critical attributes. For example, "twist the knob clockwise" becomes "twist the grasped object clockwise." During training, these attributes come from the instruction; during inference, the planner supplies them with the primitive execution flow.

Visual grounding. Segment Anything Model (SAM) (Kirillov et al., 2023) initializes the masks and Cutie (Cheng et al., 2024) tracks them over time. The mask identifies the concrete object or

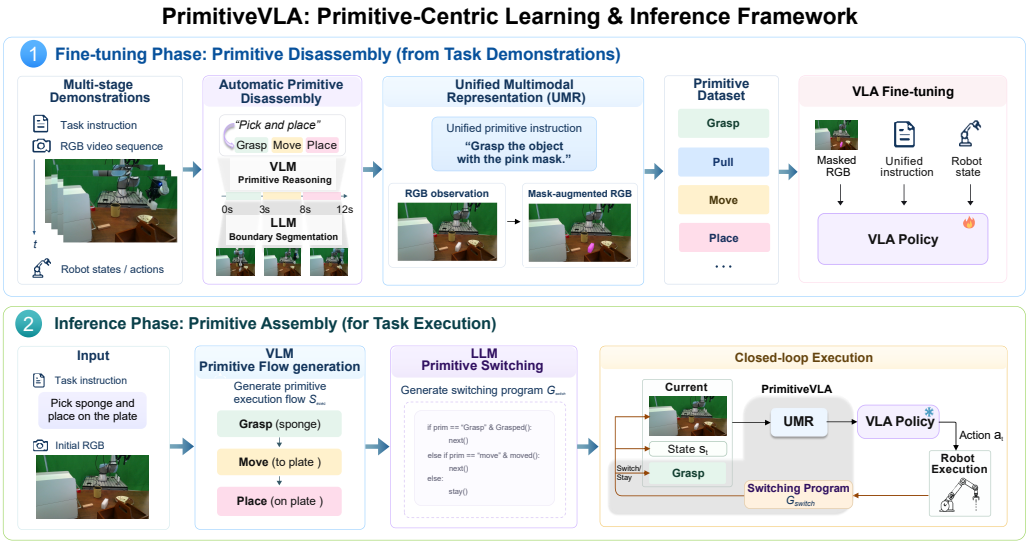


Figure 2: Overview of PrimitiveVLA. (1) In the fine-tuning phase, task demonstrations are disassembled into primitive-aligned segments: a VLM infers the primitive sequence, an LLM-generated segmentation program localizes the temporal boundaries, and UMR maps each segment to a unified primitive instruction and mask-augmented observations for VLA fine-tuning. (2) In the inference phase, a VLM generates a primitive execution flow and an LLM generates a causal switching program before execution. During closed-loop execution, UMR constructs the input for the current primitive, the VLA predicts actions, and the switching program detects completion and advances the flow.

target while preserving scene context. Appendix A.6 gives the complete instruction mappings, mask pipeline, and offline/online implementation details.

4.2 Fine-tuning Phase: Primitive Disassembly

Existing datasets predominantly provide complete task-level trajectories τ. To obtain reusable primitive supervision, we disassemble each trajectory with an automated pipeline consisting of two steps: inferring the ordered primitive sequence (Primitive Sequence Reasoning, Sec. 4.2.1) and localizing its temporal boundaries (Boundary Segmentation, Sec. 4.2.2). UMR then converts each retained segment into a unified primitive instruction and a mask-augmented observation (Sec. 4.1), which are used as independent VLA fine-tuning samples.

4.2.1 Primitive Sequence Reasoning

The first step of disassembly is to determine the underlying execution structure. For a trajectory τ, a VLM f_{VLM} infers the ordered primitive sequence

$$\mathcal{S} = f_{\text{VLM}}(l, \mathcal{V}_\tau, \mathcal{C}), \tag{4}$$

where $\mathcal{S} = [(p_1, l_1^{\text{prim}}), \ldots, (p_m, l_m^{\text{prim}})]$, l is the task instruction, $\mathcal{V}_\tau$ is the RGB sequence, and $\mathcal{C}$ is the common primitive set (Sec. 3.2). Each $p_r \in \mathcal{C}$, while l_r^{prim} specifies the relevant object or target and required execution attributes. The sequence specifies primitive order and semantic instructions but not temporal boundaries. For dataset d, $\mathcal{S}_i$ denotes the sequence inferred for trajectory τ_i. Across trajectories, we collect primitive types as $\mathcal{P}_d = \bigcup_i \{p \mid (p, l^{\text{prim}}) \in \mathcal{S}_i\}$, which is passed to the LLM (Sec. 4.2.2). We also store each $(l_i, \mathcal{S}_i)$ pair in the Disassembly Library $\mathcal{D}$ for execution-flow generation (Sec. 4.3.1). Prompt and library-construction details are provided in Appendix A.1.

4.2.2 Boundary Segmentation

The LLM then generates a dataset-level segmentation program from $\mathcal{P}_d$, the corresponding primitive definitions, and the dataset-specific state/action format:

$$G_{\text{seg}}^{(d)} = f_{\text{LLM}}(\mathcal{P}_d,\ \text{Def}(\mathcal{P}_d),\ \text{Format}_d)\,. \tag{5}$$

Here, $\mathrm{Def}(\mathcal{P}_d)$ specifies the physical completion conditions, while Format_d specifies the dimensions, ordering, physical meanings, and whether the state/action values are absolute or relative. The resulting program is applied to the original trajectories; we omit its dataset superscript below. For primitive p_i starting at t_{start}, its endpoint is

$$t_{\mathrm{end}} = \min\{t \mid t \geq t_{\mathrm{start}} + \delta,\ G_{\mathrm{seg}}(p_i, s_{t-k:t+k}, a_{t-k:t+k}) = 1\}. \tag{6}$$

Here, k is the offline window radius and δ is the minimum-duration offset. The symmetric window captures local motion dynamics; for example, a *Grasp* segment ends when the gripper remains closed and the end effector begins a sustained lift. Appendix C.1.1 evaluates label and boundary accuracy, while Appendix A.2 gives the prompt and implementation details. The pseudocode is summarized in Appendix A.5.

From segmentation to VLA fine-tuning. For each trajectory, we traverse $\mathcal{S}$ in order. Starting from the first unassigned frame, G_{seg} selects the branch for p_i and locates its endpoint t_{end} using Eq. 6; segmentation then continues with p_{i+1}. If a non-final endpoint is missing, resolved segments are retained and the remaining frames are discarded; the final primitive receives all remaining frames. UMR maps each retained segment's instruction and RGB observations to a unified instruction c_i and mask-augmented observation $\tilde{o}_i$, retaining execution attributes when needed. These transformed segments are treated as independent samples and used to fine-tune the VLA with $\mathcal{L}_{\mathrm{prim}}$ in Eq. 2.

4.3 Inference Phase: Primitive Assembly

Primitive Assembly consists of primitive-flow generation (Sec. 4.3.1) and causal primitive switching (Sec. 4.3.2). Given the task instruction and initial observation, a VLM generates the ordered execution flow, while an LLM generates the switching program before execution. During rollout, UMR supplies the VLA with the unified instruction and mask-augmented observation for the current primitive, and the switching program advances the flow when its completion condition is satisfied.

4.3.1 Primitive Flow Generation

At the beginning, for a test task l_{test}, we retrieve the top-3 similar instruction–sequence pairs $(l, \mathcal{S})$ from the Disassembly Library $\mathcal{D}$ using semantic cosine similarity. Together with l_{test}, the initial observation o_0, and the common primitive set $\mathcal{C}$, these pairs condition the VLM:

$$\mathcal{S}_{\mathrm{exec}} = f_{\mathrm{VLM}}(l_{\mathrm{test}}, o_0, \mathcal{C}, \mathrm{Retrieve}(l_{\mathrm{test}}, \mathcal{D})). \tag{7}$$

The resulting $\mathcal{S}_{\mathrm{exec}} = [(p_i, l_i^{\mathrm{prim}})]_{i=1}^{N}$ specifies the primitive order, relevant objects or targets, and any required execution attributes. The retrieved pairs provide primitive-order priors from fine-tuning. Appendix A.3 gives the retrieval procedure and complete inference prompt.

4.3.2 Primitive Switching

Before execution, the LLM receives the primitive set $\mathcal{C}$, its definitions, the offline segmentation program $G_{\mathrm{seg}}^{(d)}$, and the causal history length L_{hist}, then generates the switching program once:

$$G_{\mathrm{switch}} = f_{\mathrm{LLM}}\Big(\mathcal{C}, \mathrm{Def}(\mathcal{C}), G_{\mathrm{seg}}^{(d)}, L_{\mathrm{hist}}\Big). \tag{8}$$

Because future frames are unavailable online, $\mathrm{Def}(\mathcal{C})$ and $G_{\mathrm{seg}}^{(d)}$ are converted into causal completion conditions over $\mathcal{H}_t = \{(s_j, a_j)\}_{j=t-L_{\mathrm{hist}}+1}^{t}$. A transition from p_i to p_{i+1} occurs when

$$\mathrm{SwitchTrigger}_t \iff G_{\mathrm{switch}}(p_i, \mathcal{H}_t) = 1. \tag{9}$$

Switching uses only the primitive type and causal state history, while UMR provides execution-relevant semantic and visual inputs to the VLA. Appendix A.4 provides the generation prompt; Appendix A.5 gives the pseudocode and code correspondence.

Closed-loop execution. At each step, UMR maps the current primitive instruction and RGB observation to $(c_i, \tilde{o}_t)$, and the VLA predicts actions from $(\tilde{o}_t, s_t, c_i)$. The robot executes the actions and updates the observation and state history, while a monitoring thread evaluates $G_{\mathrm{switch}}(p_i, \mathcal{H}_t)$. When the condition is satisfied, the controller advances to the next entry in $\mathcal{S}_{\mathrm{exec}}$ and updates its instruction and object mask. This loop continues until task completion.

Table 2: Libero evaluation on in-distribution tasks and zero-shot unseen task compositions. All entries are success rates (%); parentheses in the last four columns show percentage-point changes from the matched backbone. Shading separates input-modality groups, and within each group, bold marks the strongest result in each column. Backbone inputs are language (L), global observations (I_g), wrist observations (I_w), and proprioceptive states (S); "–" denotes unavailable results.

	Small-Scale				Large-Scale	Compositional Generalization	
Model	Object	Spatial	Goal	**Mean** (3 suites)	90	90-Novel	Long
Global-view inputs: L, I_g							
TraceVLA (Zheng et al., 2025b)	84.60	85.20	75.10	81.63	–	–	–
SpatialVLA (Qu et al., 2025)	88.20	89.90	78.60	85.57	–	–	–
OpenVLA (Kim et al., 2025b)	87.40	85.80	74.00	82.40	70.90	9.25	4.50
OpenVLA + PrimitiveVLA	**90.60**	**91.20**	**82.20**	**88.00** (+5.60)	**79.88** (+8.98)	**45.50** (+36.25)	**38.50** (+34.00)
Global-view + wrist-view + proprioception inputs: L, I_g, I_w, S							
Diffusion Policy (Chi et al., 2025)	92.50	78.30	68.30	79.70	–	–	–
Octo (Ghosh et al., 2024)	85.70	78.90	84.60	83.07	–	–	–
π_0 (Black et al., 2024)	96.80	98.80	95.80	97.13	–	–	–
OpenVLA-OFT (Kim et al., 2025a)	97.40	98.60	96.80	97.60	90.39	13.50	3.75
OpenVLA-OFT + PrimitiveVLA	**98.40**	**99.40**	**97.80**	**98.53** (+0.93)	94.70 (+4.31)	71.00 (+57.50)	66.50 (+62.75)
$\pi_{0.5}$ (Black et al., 2025)	98.20	98.40	96.00	97.53	**96.80**	56.00	30.50
$\pi_{0.5}$ + PrimitiveVLA	98.00	98.20	95.80	97.33 (-0.20)	96.19 (-0.61)	**75.50** (+19.50)	**80.25** (+49.75)

5 EXPERIMENTS

In this section, we evaluate PrimitiveVLA in simulation and on a physical robot to address three research questions:

- **RQ1 (Performance & Efficiency):** Can VLA models fine-tuned via our framework demonstrate superior data efficiency compared to standard VLAs under identical data conditions, and match the full-data baseline performance even with 50% training data?
- **RQ2 (Compositional Generalization and Long-Horizon Execution):** Can VLA models fine-tuned via our framework recombine learned primitives into unseen task compositions and assemble them into longer multi-stage execution flows?
- **RQ3 (Component Analysis):** What are the roles and effects of Primitive Disassembly and UMR within the PrimitiveVLA system?

5.1 EXPERIMENTAL SETUP

Simulation Benchmarks. We use Libero (Liu et al., 2023) for fine-tuning and in-distribution (ID) evaluation, covering Libero-Spatial, Libero-Object, and Libero-Goal (10 tasks each) and an 83-task Libero-90 subset after excluding seven low-quality tasks. We independently apply the data-cleaning protocol implemented in OpenVLA's code to the Libero demonstrations; dataset-level trajectory counts are reported in Appendix B.2 . For out-of-distribution (OOD) evaluation, we use eight Libero-90-Novel compositions unseen during fine-tuning and eight Libero-Long tasks composed of subtasks seen during Libero-90 fine-tuning. We additionally evaluate RLBench (James et al., 2020) on ten diverse interaction tasks, using 60 successful demonstrations per task. Complete task lists and primitive sequences are provided in Appendix Table 9 and Table 10.

Real-World Environment. To validate physical transferability, we use a UR5e arm with a Robotiq 2F-85 gripper and two RealSense L515 cameras (one third-person and one wrist-mounted, shown in the Appendix). We evaluate 11 tasks: six in-distribution (ID), three compositional generalization, and two long-horizon tasks. The latter five are evaluated zero-shot. Complete task lists and primitive sequences are also provided in Appendix Table 9.

Base Models. To demonstrate that PrimitiveVLA works across different VLA backbones, we evaluate three representative backbones on the simulation benchmarks: OpenVLA (Kim et al., 2025b) (a standard Transformer-based VLA); OpenVLA-OFT (Kim et al., 2025a) (an optimized fine-tuning variant of OpenVLA); and $\pi_{0.5}$ (Black et al., 2025) (a state-of-the-art foundation model using flow matching). The physical-robot experiments use $\pi_{0.5}$.

Metrics. We report the average success rate. Simulation tasks are evaluated over 50 trials, while real-world and RLBench tasks are evaluated over 20 trials. More details are listed in the Appendix B.

Table 3: **Data efficiency analysis.** We compare four settings across VLA backbones, varying data ratio (50% vs. 100%) and method (baseline vs. PrimitiveVLA). For 50% data, we retain the first half of each task's demonstrations; PrimitiveVLA applies the same segmentation pipeline as for 100%.

		Small-Scale			Large-Scale	
Model	**Data Scale**	Libero-Object	Libero-Spatial	Libero-Goal	Libero-90	**Mean** (4 suites)
OpenVLA (Base)	50%	83.40%	73.40%	73.60%	66.02%	74.11%
	100%	87.40%	85.80%	74.00%	70.90%	79.53%
OpenVLA + Ours	50%	87.60%	87.00%	73.20%	72.60%	80.10%
	100%	**90.60%**	**91.20%**	**82.20%**	**79.88%**	**85.97%**
OpenVLA-OFT (Base)	50%	96.20%	97.00%	91.00%	87.00%	92.80%
	100%	97.40%	98.60%	96.80%	90.39%	95.80%
OpenVLA-OFT + Ours	50%	94.80%	97.80%	96.00%	92.12%	95.18%
	100%	**98.40%**	**99.40%**	**97.80%**	**94.70%**	**97.58%**

Table 4: RLBench success rates (%). T1–T10 follow the task order in Table 9. Each PrimitiveVLA row is compared with standard fine-tuning for the same backbone; bold denotes the higher value or a tie within each pair.

Backbone	**Training**	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	**Mean**
$\pi_{0.5}$	Standard	**10%**	**40%**	**55%**	5%	20%	**40%**	15%	55%	**15%**	40%	29.50%
	PrimitiveVLA	**10%**	**40%**	**55%**	**10%**	**55%**	**40%**	**45%**	**65%**	5%	**50%**	**37.50%**
OpenVLA	Standard	0%	45%	**55%**	0%	15%	**45%**	10%	75%	5%	10%	26.00%
	PrimitiveVLA	**10%**	**70%**	**55%**	**10%**	**60%**	15%	**20%**	**90%**	**20%**	**40%**	**39.00%**
OpenVLA-OFT	Standard	0%	**5%**	0%	0%	0%	**15%**	5%	0%	0%	**5%**	3.00%
	PrimitiveVLA	0%	**5%**	**10%**	0%	0%	**15%**	**10%**	0%	0%	**5%**	**4.50%**

5.2 Data Efficiency (RQ1)

We assess data efficiency across VLA architectures on Libero using full and 50% data, and further evaluate performance under diverse interactions and object configurations on RLBench.

Full-data Libero and cross-backbone RLBench results. PrimitiveVLA improves OpenVLA and OpenVLA-OFT on most Libero settings (Table 2), while largely preserving the strong $\pi_{0.5}$ ID performance. On RLBench, the mean success rate increases from 29.50% to 37.50% for $\pi_{0.5}$, from 26.00% to 39.00% for OpenVLA, and from 3.00% to 4.50% for OpenVLA-OFT (Table 4).

Matching Full-Data Performance with 50% Data. Under the 50% data protocol, PrimitiveVLA achieves higher mean success rates than the corresponding baseline for both backbones and improves most individual Libero settings. As shown in Table 3, OpenVLA-based PrimitiveVLA reaches 80.10%, slightly exceeding the 100%-data OpenVLA baseline of 79.53%. For OpenVLA-OFT, the 50%-data PrimitiveVLA variant reaches 92.12% on Libero-90, which is 1.73 percentage points higher than the 100%-data baseline. These results suggest that primitive-aligned supervision can provide useful training signals for recurring behaviors under data-constrained settings.

5.3 Compositional Generalization and Long-Horizon Execution (RQ2)

We fine-tune on Libero-90 only and evaluate zero-shot transfer to Libero-90-Novel and Libero-Long. Libero-90-Novel recombines objects, locations, and task logic, whereas Libero-Long assembles familiar subtasks into longer flows. PrimitiveVLA reuses the Libero-90 prompt templates, Disassembly Library, and switching program in both settings (Table 2).

Compositional generalization. OpenVLA, OpenVLA-OFT, and $\pi_{0.5}$ obtain 9.25%, 13.50%, and 56.00% on Libero-90-Novel. PrimitiveVLA raises these results to 45.50%, 71.00%, and 75.50%, respectively, demonstrating transfer to new combinations of objects, locations, and operations in the evaluated tasks. Breakdowns by novel objects, locations, object–location pairs, and task structures appear in Appendix C.2, while video-verified primitive-transfer results appear in Appendix D.2.

Long-horizon execution. For Libero-Long, PrimitiveVLA uses its flow and switching controller for subtask transitions; baselines use the same flow-based switching logic internally only for reset detection. The arm resets only between subtasks, while object states are preserved. OpenVLA and OpenVLA-OFT obtain below 5% in this setting, while $\pi_{0.5}$ obtains 30.50%. PrimitiveVLA improves all three backbones and raises $\pi_{0.5}$ to 80.25%.

Table 5: **Ablation study.** Results on ID, OOD, and Long-Horizon tasks. "w/o UMR" removes unified primitive instructions and object masks while retaining primitive disassembly and switching. "w/o Disass." uses full-task instructions and object masks, omitting primitive-aligned training and primitive switching. All other settings are unchanged.

Base Model	Setting	Libero-90 (ID)	Libero-90-Novel (OOD)	Libero-Long
OpenVLA	Baseline	70.90%	9.25%	4.50%
	w/o UMR (Disass. Only)	73.90%	21.50%	28.25%
	w/o Disass. (Full-task + Mask)	79.60%	33.75%	4.25%
	Ours (PrimitiveVLA)	**79.88%**	**45.50%**	**38.50%**
OpenVLA-OFT	Baseline	90.39%	13.50%	3.75%
	w/o UMR (Disass. Only)	89.60%	15.00%	52.25%
	w/o Disass. (Full-task + Mask)	94.30%	60.00%	39.75%
	Ours (PrimitiveVLA)	**94.70%**	**71.00%**	**66.50%**

Table 6: **Real-world success rates.** Results for in-distribution tasks (T1–T6), zero-shot compositional generalization (T7–T9), and zero-shot long-horizon execution (T10–T11). Task identifiers follow the order listed in Appendix Table 9.

	In-Distribution (ID)							Compositional Generalization				Long-Horizon Execution		
Model	T1	T2	T3	T4	T5	T6	**Mean**	T7	T8	T9	**Mean**	T10	T11	**Mean**
$\pi_{0.5}$	60%	75%	60%	45%	95%	85%	70%	15%	35%	10%	20%	0%	20%	10%
$\pi_{0.5}$ **+ Ours**	**85%**	**95%**	**85%**	**85%**	**100%**	**90%**	**90%**	**40%**	**50%**	**80%**	**57%**	**60%**	**70%**	**65%**

5.4 Ablation Studies (RQ3)

We use ablation studies to examine the contributions of Primitive Disassembly and Unified Multimodal Representation (UMR) across ID, OOD, and long-horizon tasks (Table 5).

The ablation results reveal complementary benefits from visual grounding and primitive-based execution. With full-task instructions and object masks but no primitive decomposition or switching, OpenVLA-OFT improves from 13.50% to 60.00% on Libero-90-Novel. Primitive Disassembly without UMR raises its Libero-Long success rate from 3.75% to 52.25%. Combining Primitive Disassembly with full UMR achieves 71.00% on Novel and 66.50% on Long, suggesting that their combination is beneficial for the evaluated compositional and long-horizon tasks. Additional ablations in Appendix C.1 evaluate Primitive Disassembly, UMR, and primitive switching and compare them with alternative designs. We also analyze online switching reliability in Appendix D.3.

5.5 Real-World Evaluation

We deployed PrimitiveVLA on a UR5e robot using the $\pi_{0.5}$ backbone. We collected demonstrations with spatial variation and evaluated six in-distribution tasks and five zero-shot tasks spanning compositional generalization and long-horizon execution (Table 6). PrimitiveVLA improves mean success from 70% to 90% on ID tasks, from 20% to 57% in the compositional generalization setting, and from 10% to 65% in the long-horizon execution setting.

6 Conclusion

PrimitiveVLA organizes VLA fine-tuning around primitive disassembly and ordered assembly. Primitive disassembly converts task demonstrations into reusable primitive-aligned samples, while UMR combines unified primitive language with object-mask grounding. During inference, flow generation and causal switching assemble these primitives into complete task executions. Across our simulation and real-robot evaluations, PrimitiveVLA improves performance under reduced data and on compositional and long-horizon tasks while maintaining or improving in-distribution performance.

Limitations. PrimitiveVLA currently relies on a manually defined fixed primitive set for single-arm, standard-gripper manipulation of rigid and articulated objects. Dexterous, deformable, bimanual, and fine-contact tasks may require revising this set or using finer-grained interaction primitives. In addition, history-based switching relies on robot state and action histories, which can make termination ambiguous during contact-rich interactions. Future work will explore automatic primitive discovery, adaptive granularity, and joint visual–state switching for broader manipulation settings.

AI USE STATEMENT

Generative AI tools were used in two capacities. First, as components of PrimitiveVLA, a VLM infers primitive sequences and localizes task-relevant entities, while an LLM generates dataset-level segmentation and causal switching programs. These outputs are used to transform existing demonstrations into primitive-aligned training samples and to manage primitive transitions during inference, as described and evaluated in the paper. Second, during manuscript preparation and revision, generative AI tools assisted with translation, language-level drafting and editing to improve clarity and organization, LaTeX formatting, and review of the manuscript. In particular, they provided feedback on the methodology, experimental design, ablation coverage, statistical reporting, and the scope of the conclusions. The authors independently evaluated these suggestions and decided which revisions to adopt.

Generative AI tools were not used to generate synthetic evaluation data, originate the conceptual framework, or formulate mathematical claims or proofs. All AI-assisted content and model-generated programs were reviewed and verified by the authors, and the model-generated programs used within PrimitiveVLA were tested through the experiments reported in the paper. The authors take full responsibility for the final manuscript.

REPRODUCIBILITY STATEMENT

Prompts, task lists, data-processing procedures, model settings, evaluation protocols, and additional analyses are provided in Appendices A–D. The generated programs, trained models, and processing code will be released upon acceptance, subject to the licenses of the underlying assets.

REFERENCES

Anurag Ajay, Aviral Kumar, Pulkit Agrawal, Sergey Levine, and Ofir Nachum. Opal: Offline primitive discovery for accelerating offline reinforcement learning. In *International Conference on Learning Representations*, 2021.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. pi_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Manuel Y Galliker, et al. $pi_0.5$: a vision-language-action model with open-world generalization. In *9th Annual Conference on Robot Learning*, 2025.

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.

Ho Kei Cheng, Seoung Wug Oh, Brian Price, Joon-Young Lee, and Alexander Schwing. Putting the object back into video object segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3151–3161, 2024.

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.

DeepSeek-AI. Deepseek-v3 technical report, 2024. URL `https://arxiv.org/abs/2412.19437`.

Jiafei Duan, Wentao Yuan, Wilbert Pumacay, Yi Ru Wang, Kiana Ehsani, Dieter Fox, and Ranjay Krishna. Manipulate-anything: Automating real-world robots using vision-language models. In *Conference on Robot Learning*, pp. 5326–5350. PMLR, 2025.

Dibya Ghosh, Homer Rich Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, Jianlan Luo, et al. Octo: An open-source generalist robot policy. In *Robotics: Science and Systems*, 2024.

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng

Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In Karen Liu, Dana Kulic, and Jeff Ichnowski (eds.), *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 14–18 Dec 2023. URL https://proceedings.mlr.press/v205/ichter23a.html.

Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. Dynamical movement primitives: learning attractor models for motor behaviors. *Neural computation*, 25(2):328–373, 2013.

Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J Davison. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 5(2):3019–3026, 2020.

Yunfan Jiang, Agrim Gupta, Zichen Zhang, Guanzhi Wang, Yongqiang Dou, Yanjun Chen, Li Fei-Fei, Anima Anandkumar, Yuke Zhu, and Linxi Fan. Vima: General robot manipulation with multimodal prompts. *arXiv preprint arXiv:2210.03094*, 2(3):6, 2022.

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 Baijal, Mateo Guaman Castro, Daphne Chen, Qiuyu Chen, Trinity Chung, Jaimyn Drake, Ethan Paul Foster, Jensen Gao, David Antonio Herrera, Minho Heo, Kyle Hsu, Jiaheng Hu, Donovon Jackson, Charlotte Le, Yunshuang Li, 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. In *Proceedings of Robotics: Science and Systems*, Delft, Netherlands, July 2024. doi: 10.15607/RSS.2024.XX.120.

Moo Jin Kim, Chelsea Finn, and Percy Liang. Fine-tuning vision-language-action models: Optimizing speed and success. *arXiv preprint arXiv:2502.19645*, 2025a.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P Foster, Pannag R Sanketi, Quan Vuong, et al. Openvla: An open-source vision-language-action model. In *Conference on Robot Learning*, pp. 2679–2713. PMLR, 2025b.

Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything. *arXiv:2304.02643*, 2023.

Seungjae Lee, Yibin Wang, Haritheja Etukuru, H. Jin Kim, Nur Muhammad Mahi Shafiullah, and Lerrel Pinto. Behavior generation with latent actions. In *Forty-first International Conference on Machine Learning (ICML)*, 2024.

Xinghang Li, Minghuan Liu, Hanbo Zhang, Cunjun Yu, Jie Xu, Hongtao Wu, Chilam Cheang, Ya Jing, Weinan Zhang, Huaping Liu, Hang Li, and Tao Kong. Vision-language foundation models as effective robot imitators. *arXiv preprint arXiv:2311.01378*, 2023.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *arXiv preprint arXiv:2209.07753*, 2022.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36: 44776–44791, 2023.

Songming Liu, Lingxuan Wu, Bangguo Li, Hengkai Tan, Huayu Chen, Zhengyi Wang, Ke Xu, Hang Su, and Jun Zhu. Rdt-1b: a diffusion foundation model for bimanual manipulation. In *The Thirteenth International Conference on Learning Representations*, 2024.

Abby O'Neill, Abdul Rehman, Abhiram Maddukuri, Abhishek Gupta, Abhishek Padalkar, Abraham Lee, Acorn Pooley, Agrim Gupta, Ajay Mandlekar, Ajinkya Jain, et al. Open x-embodiment: Robotic learning datasets and rt-x models: Open x-embodiment collaboration 0. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6892–6903. IEEE, 2024.

Karl Pertsch, Youngwoon Lee, and Joseph Lim. Accelerating reinforcement learning with learned skill priors. In *Conference on robot learning*, pp. 188–204. PMLR, 2021.

Physical Intelligence, Bo Ai, Ali Amin, Raichelle Aniceto, Ashwin Balakrishna, Greg Balke, Kevin Black, George Bokinsky, Shihao Cao, Thomas Charbonnier, Vedant Choudhary, Foster Collins, Ken Conley, Grace Connors, James Darpinian, Karan Dhabalia, Maitrayee Dhaka, Jared DiCarlo, Danny Driess, Michael Equi, Adnan Esmail, Yunhao Fang, Chelsea Finn, Catherine Glossop, Thomas Godden, Ivan Goryachev, Lachlan Groom, Haroun Habeeb, Hunter Hancock, Karol Hausman, Gashon Hussein, Victor Hwang, Brian Ichter, Connor Jacobsen, Szymon Jakubczak, Rowan Jen, Tim Jones, Gregg Kammerer, Ben Katz, Liyiming Ke, Mairbek Khadikov, Chandra Kuchi, Marinda Lamb, Devin LeBlanc, Brendon LeCount, Sergey Levine, Xinyu Li, Adrian Li-Bell, Vladislav Lialin, Zhonglin Liang, Wallace Lim, Yao Lu, Enyu Luo, Vishnu Mano, Nandan Marwaha, Aikys Mongush, Liam Murphy, Suraj Nair, Tyler Patterson, Karl Pertsch, Allen Z. Ren, Gavin Schelske, Charvi Sharma, Baifeng Shi, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, Will Stoeckle, Jiaming Tang, Jimmy Tanner, Shalom Tekeste, Marcel Torne, Kyle Vedder, Quan Vuong, Anna Walling, Haohuan Wang, Jason Wang, XuDong Wang, Chris Whalen, Samuel Whitmore, Blake Williams, Charles Xu, Sukwon Yoo, Lili Yu, Wuming Zhang, Zhuoyang Zhang, and Ury Zhilinsky. $\pi_{0.7}$: a steerable generalist robotic foundation model with emergent capabilities. 2026. URL https://arxiv.org/abs/2604.15483.

Delin Qu, Haoming Song, Qizhi Chen, Yuanqi Yao, Xinyi Ye, Yan Ding, Zhigang Wang, JiaYuan Gu, Bin Zhao, Dong Wang, et al. Spatialvla: Exploring spatial representations for visual-language-action model. *arXiv preprint arXiv:2501.15830*, 2025.

Qwen Team. Qwen3.5-omni technical report, 2026. URL https://arxiv.org/abs/2604.15804.

Nur Muhammad Shafiullah, Zichen Cui, Ariuntuya (Arty) Altanzaya, and Lerrel Pinto. Behavior transformers: Cloning k modes with one stone. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 22955–22968. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/90d17e882adbdda42349db6f50123817-Paper-Conference.pdf.

Yichao Shen, Fangyun Wei, Zhiying Du, Yaobo Liang, Yan Lu, Jiaolong Yang, Nanning Zheng, and Baining Guo. Videovla: Video generators can be generalizable robot manipulators, 2025. URL https://arxiv.org/abs/2512.06963.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Qwen Team. Qwen2.5-vl technical report, 2025a. URL https://arxiv.org/abs/2502.13923.

Qwen Team. Qwen3-vl technical report, 2025b. URL https://arxiv.org/abs/2511.21631.

Feng Xu, Guangyao Zhai, Xin Kong, Tingzhong Fu, Daniel FN Gordon, Xueli An, and Benjamin Busam. Stare-vla: Progressive stage-aware reinforcement for fine-tuning vision-language-action models. *arXiv preprint arXiv:2512.05107*, 2025.

Kaidong Zhang, Pengzhen Ren, Bingqian Lin, Junfan Lin, Shikui Ma, Hang Xu, and Xiaodan Liang. Pivot-r: Primitive-driven waypoint-aware world model for robotic manipulation. *Advances in Neural Information Processing Systems*, 37:54105–54136, 2024a.

Zichen Zhang, Yunshuang Li, Osbert Bastani, Abhishek Gupta, Dinesh Jayaraman, Yecheng Jason Ma, and Luca Weihs. Universal visual decomposer: Long-horizon manipulation made easy. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6973–6980. IEEE, 2024b.

Qingqing Zhao, Yao Lu, Moo Jin Kim, Zipeng Fu, Zhuoyang Zhang, Yecheng Wu, Zhaoshuo Li, Qianli Ma, Song Han, Chelsea Finn, Ankur Handa, Tsung-Yi Lin, Gordon Wetzstein, Ming-Yu Liu, and Donglai Xiang. Cot-vla: Visual chain-of-thought reasoning for vision-language-action models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1702–1713, June 2025.

Jinliang Zheng, Jianxiong Li, Dongxiu Liu, Yinan Zheng, Zhihao Wang, Zhonghong Ou, Yu Liu, Jingjing Liu, Ya-Qin Zhang, and Xianyuan Zhan. Universal actions for enhanced embodied foundation models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 22508–22519, June 2025a.

Ruijie Zheng, Yongyuan Liang, Shuaiyi Huang, Jianfeng Gao, Hal Daumé III, Andrey Kolobov, Furong Huang, and Jianwei Yang. Tracevla: Visual trace prompting enhances spatial-temporal awareness for generalist robotic policies. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Jiaming Zhou, Ke Ye, Jiayi Liu, Teli Ma, Zifan Wang, Ronghe Qiu, Kun-Yu Lin, Zhilin Zhao, and Junwei Liang. Exploring the limits of vision-language-action manipulations in cross-task generalization. *arXiv preprint arXiv:2505.15660*, 2025.

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

APPENDIX

Organization of the appendix.

- Appendix A provides method details, prompts, pseudocode, and UMR instruction mappings.
- Appendix B describes task lists, data processing, training configurations, and evaluation settings.
- Appendix C reports component ablations and additional compositional generalization experiments.
- Appendix D analyzes action patterns, primitive transfer, online switching, VLM/LLM errors and hallucination mitigation, RLBench performance, compositional generalization, data efficiency, and latency.
- Appendix E describes the supplementary videos.
- Appendix F lists the existing assets used in this work.
- Appendix G presents the release plan.

A MORE DETAILS ABOUT METHOD

A.1 FINE-TUNING: PRIMITIVE SEQUENCE REASONING

Primitive sequence reasoning determines which primitives occur in a demonstration and in what order, while the subsequent state/action program localizes their temporal boundaries. We use a VLM for this semantic stage because state/action signals alone do not specify the intended primitive identity or task structure. This separation provides the boundary segmenter with an ordered semantic prior and avoids asking one procedure to infer both primitive labels and transition times.

To facilitate structured disassembly, we feed the VLM with a comprehensive input tuple: the high-level task description, the sampled RGB frame sequence of the demonstration, and the same fixed primitive set $\mathcal{C}$ used for every dataset. The VLM generates a primitive sequence for each task by selecting from $\mathcal{C}$. The union $\mathcal{P}_d$ records which primitive labels occur in dataset d and is subsequently provided to the LLM when generating the dataset-level segmentation program.

The VLM outputs are stored for reuse in the execution framework. For each task, the VLM produces an instruction–sequence pair $(Inst, Seq)$, where $Inst$ is the task instruction and Seq is the corresponding primitive sequence. These pairs are archived in a Disassembly Library $\mathcal{D}$. During inference, the execution flow generator queries this library to retrieve and adapt relevant primitive sequences for the new task.

Specifically, we adopt Qwen3-VL (Team, 2025b) as the VLM in our framework, due to its strong video understanding and logical reasoning capabilities, as well as its suitability as an open-source model. Here is an illustrative example of the prompt used for primitive sequence reasoning:

System Prompt: Primitive Reasoning

[System Role]

You are a robotic reasoning expert. Your goal is to decompose a long-horizon manipulation video into a structured sequence of physical primitives based on the provided primitive set and merging rules.

[Primitive Definitions]

1. **grasp**: Move toward an object, seize it, and lift slightly.
2. **lift**: Move upward significantly (z-axis) with closed gripper to reach a height, possibly with minor xy adjustment.
3. **move**: Large-scale movement in the xy-plane while the gripper is closed.
4. **place**: Descend to release an object at target location and lift away.

5. **push**: Apply force to slide an object to a target position.
6. **press**: Apply downward force on an object or a surface.
7. **twist**: Rotate the gripper around its central axis (e.g., turning a knob).
8. **tilt**: Rotate around a joint beyond the wrist (adduction/abduction of the arm).
9. **rotate**: Rotate along a fixed axis of an object (e.g., opening a laptop).
10. **pull**: Pull an object toward a target position.
11. **insert**: Insert an object or the gripper into a gap/slot.

[Merging Rules]

- **Object-Independent Pre-movement**: Motions without object interaction (e.g., approaching a mug) should be merged into the subsequent primitive (e.g., use *grasp* instead of *move + grasp*).
- **State Maintenance**: If the arm maintains a grip, label as *move* unless it aligns with *lift* or *push*. Do not fragment unintentional drifts or jitter.

[Input]
Task Instruction: "{task}"
Video Observation: [Sequence of visual frames $I_{1:T}$]

[Output Requirement]
Reasoning: [Explain the flow and why certain drifts were merged.]
Sequence: [Primitive_1(Instruction), Primitive_2(Instruction), ...]
Each primitive instruction should identify the relevant object or target and preserve execution-relevant attributes, such as rotation direction, when required by the task.

A.2 Fine-tuning: Boundary Segmentation

The VLM supplies the primitive sequence but is not used as the final boundary criterion. As analyzed in Appendix C.1.1, its temporal estimates are less accurate than state-based segmentation. We therefore execute a deterministic program generated from explicit primitive criteria and trajectory fields.

However, manually authoring a segmentation program for each specific task is as labor-intensive as designing individual reward functions. To overcome this, we utilize an LLM-driven automated code generation pipeline. Our core philosophy is a unified segmentation standard applied consistently within each dataset. By providing the LLM with the dataset-specific primitive types $\mathcal{P}_d$ derived from primitive sequence reasoning, their semantic definitions, and generic segmentation criteria, we generate a program that extracts reusable physical primitives across tasks.

To ensure robustness, the system incorporates the following design considerations:

- **Temporal Window Consistency:** Each primitive's termination is determined from a local window containing states before and after a candidate moment. This prevents premature triggers caused by sensor noise and promotes more consistent physical grounding.
- **Dataset-Level Threshold Calibration:** Providing segmentation exemplars reduces the need for manual threshold adjustment. Calibration is performed once per dataset using only training demonstrations, which are disjoint from policy-evaluation rollouts in both simulation and real-world settings. The segmentation program and its thresholds are then held fixed across all tasks in that setting.

The semantic primitive definitions are shared across datasets, while the concrete segmentation program is dataset-specific. We use separate programs for LIBERO, RLBench, and the real-world data, each adapted to the corresponding state/action representation and primitive types. The complete dataset-specific segmentation programs will be released upon acceptance.

We adopt DeepSeek-V3 (DeepSeek-AI, 2024) as the LLM in our framework because of its strong logical reasoning and code-generation capabilities and its open-weight availability. The following prompt illustrates the real-world calibration setting, using exemplars from six real-world task types covering five primitive types to generate a boundary-segmentation program that determines the termination index of each primitive segment:

System Prompt: Code Generation for Primitive Segmentation

[System Role]
You are an expert AI programming assistant. Generate a Python function to determine the termination index of robotic primitives based on the provided physical criteria and trajectory examples.

[Primitive Segmentation Criteria for Real-World Tasks]

1. **grasp**: Gripper state remains constant (closed) for current and future frames; action is "close" (1); terminates when future frames show significant movement in xy or z.
2. **lift**: Significant z-axis increase over past frames; terminates when upward motion stops, begins descending, or xy motion ceases.
3. **move**: Large xy displacement over past frames; terminates when xy motion stops or z begins a sharp descent.
4. **place**: Gripper state is "open" (0) for current and past frames; terminates when a z-axis lift (departure) is detected.
5. **push**: Significant xy displacement over past frames while z remains stable; terminates when xy motion ceases.

[In-Context Exemplar]
// Example: Segmentation at $t = 40$
Task: "Pick up cup" | **Primitive**: "grasp" | **Split Index (t)**: 40

Index	Pos	x	y	z	grip	Window Logic
35	$t-5$	0.120	0.450	0.020	0.98	Window Start: Gripper stable
...	...	...	...	...	...	...
40	t	**0.121**	**0.450**	**0.022**	**0.99**	**Target Split Point**
...	...	...	...	...	...	...
45	$t+5$	0.126	0.455	0.045	0.99	Window End: Sustained z-lift

Split Logic: At index 40, the criterion is met because for all $i \in [35, 45]$, the gripper is closed, and the velocity trend within this temporal window shows a transition to vertical rising.

[Functional Requirements]

- **Input**: states ($T \times 7$), actions ($T \times 7$), primitive_name, t_start, is_last.
- **Temporal Consistency**: Evaluation must start from $t_{\text{start}} + 10$.
- **Window-Based Decision**: Analyze local trends in a window (e.g., 5-10 frames) to ensure physical state changes are sustained and not noise-induced.

[Output Format]
Return a standalone Python function: def get_segmentation_index(...):

The offline procedure is summarized in Figure 3: SEGMENTTRAJECTORY scans candidate frames after the minimum-duration offset, while GETRULE(P) selects the primitive-specific completion criterion. These names are explanatory abstractions and need not appear as separate Python functions.

A.3 INFERENCE: PRIMITIVE EXECUTION FLOW GENERATION

During inference, the robot generates a primitive execution flow for a target task from the initial RGB observation, without a pre-recorded demonstration video. We use retrieval-augmented execution flow generation to ground the VLM output in primitive sequences observed during fine-tuning.

We maintain a Disassembly Library $\mathcal{D}$ containing successful $(Inst, Seq)$ pairs exclusively from tasks included in the fine-tuning dataset. For a given test instruction, we calculate the cosine similarity between its semantic embedding and all instructions in $\mathcal{D}$. The top-3 most similar tasks are retrieved and provided to the VLM as in-context priors. The test instruction, common primitive set $\mathcal{C}$, initial RGB observation, and retrieved exemplars jointly condition the generated execution flow on the robot's known physical capabilities.

We use Qwen3-VL as the base model. The following is an illustrative prompt for retrieval-augmented primitive execution flow generation:

System Prompt: VLM-based Primitive Reasoning (Inference)
[Context]
Given the **Initial Environment Image** and the **Full Task Instruction**, decompose the task into an ordered sequence of primitives and provide corresponding specific instructions for each.

[Primitive Merging & Selection Rules]

- **Motion Consolidation:** Movements independent of a specific object should be merged into the subsequent interaction. For example, use *grasp* directly instead of *move + grasp*.
- **Motion-Subspace Merging:** Adjacent primitives may be merged when they occupy the same dominant motion subspace, such as consecutive planar xy motions or consecutive vertical z motions, provided that no grasp, release, contact, or target-state transition separates them.
- **Z-Axis Prioritization:** For tasks with significant height requirements (e.g., target position is elevated), prioritize using *lift* or *place* over horizontal primitives.
- **Multi-Object Labeling:** If the task involves multiple sub-goals (e.g., picking multiple objects), append numerical suffixes (*_1*, *_2*) to repeated primitives to avoid ambiguity.

[Primitive Definitions (excerpt)]

◇ **grasp:** Approaching, closing the gripper on an object, and slightly lifting.
◇ **lift:** Moving upward significantly in the z-direction while the gripper is closed to meet height constraints; may involve xy displacement.
◇ **move:** Large-scale horizontal movement in the xy-plane while the gripper is closed.
◇ **place:** Descending to place an object at a target position followed by a departure lift.
◇ **push:** Pushing an object across a surface to a target location.
The complete definitions for all primitives in $\mathcal{C}$, including *insert*, *pull*, *press*, *twist*, *tilt*, and *rotate*, are provided in the prompt context.

[Retrieved Prior References (In-Context)]
The following "Task → Primitive Sequence" mappings are retrieved from the Disassembly Library based on semantic similarity. Use them as references for typical decomposition strategies:
• *pick sponge and place on the plate*: [grasp, move, place]
• *pick cup and place on the cabinet*: [grasp, lift, place]
• *push in the bottom drawer*: [push]

[Target Task]
Current Instruction: "task"
Initial Image: [Visual input showing current scene configuration]

[Expected Output]
Based on the image and instruction, provide a reasonable primitive-level decomposition. *Output Format*: Reasoning: [Explain the decomposition.]
Sequence: [Primitive_1(Instruction), Primitive_2(Instruction), ...]
Each primitive instruction should identify the relevant object or target and preserve execution-relevant attributes when required by the task.

A.4 INFERENCE: PRIMITIVE SWITCHING

To preserve temporal consistency and minimize latency during inference, we employ a generated-program controller for primitive switching. Unlike offline segmentation, the inference controller has no access to future states. The symmetric temporal window is therefore replaced by a history window containing only historical proprioception and executed actions.

To maintain consistency between fine-tuning and inference, we provide the original offline segmentation program as a reference within the prompt. We utilize DeepSeek V3 to automatically simplify and adapt this logic into a lightweight, real-time switching program. This aligns the physical transition criteria across both phases.

For the real-world setting, the switching program is generated from the same fixed, dataset-calibrated offline program constructed using the six real-world training task types covering five primitive types. It receives the fixed offline segmentation program as reference and generates the causal switching program before execution. The following prompt shows the real-world switching configuration:

System Prompt: Real-time Primitive Switching Generation
[System Role]
You are a robotic software engineer. Your task is to generate a Python function get_next_primitive for online execution. You must **strictly adhere** to the original primitive definitions and segmentation conditions provided below, adapting them **only** in implementation to fit a 10-step history buffer.

[Input Context]
- **History States** ($s_{t-9:t}$): Last 10 frames [7-dim: pos, axis-angle, gripper_state].
- **History Actions** ($a_{t-9:t}$): Last 10 frames [7-dim: relative_pos, relative_axis-angle, gripper_action].
- **Current Primitive**: The name of the primitive being executed.
- **Primitive Sequence**: The generated execution flow (list of strings).

[Implementation Constraints]
- **Causal Logic Only**: You have NO access to future frames. The "Current Frame" in the offline logic corresponds to the **last frame** (index 9) of the input history.
- **Strict Condition Adherence**: Do not modify the physical thresholds (e.g., $0.02, 0.08, 0.005$) or the qualitative definitions from the reference code.
- **Window Adaptation**: For conditions that originally required a future window, use the tail end of the 10-step history to verify if those trends have **already started to manifest** at the current moment t.

[Reference Criteria for the Five Real-World Primitives]
// You must use the exact logic for grasp, lift, move, place, and push as defined here: //

1. **grasp**: The gripper state remains constant across recent history frames, the gripper action remains in the closed state, and the relevant horizontal or z-axis motion has started to appear near the current frame.
2. **lift**: Significant z-axis increase over several "past" frames in the buffer; the current/recent frames show a stop in rising, a continuous descending trend, or a cessation of movement in both xy directions.
3. **move**: Significant x or y movement over several "past" frames; movement in xy directions tends to stop in recent frames, or the z-axis begins a sharp descent.
4. **place**: Gripper state is "open" across recent and "past" frames, accompanied by a lift in the z direction.
5. **push**: Significant x or y movement over several "past" frames; xy movement tends to stop in recent frames, with minimal change in the z direction throughout.

[Reference Code]
(The complete offline segmentation program is provided in the prompt context as reference code.) The switching program is generated once before execution and remains fixed throughout the rollout. No evaluation trajectories or manually annotated evaluation boundaries are used to generate or calibrate this program.

[Function Logic]

→ If current_primitive is the last in seq: return current_primitive.

→ If check_[primitive]_condition returns *True*: return next_primitive from seq.

→ Otherwise: return current_primitive.

[Output Format]
Return the Python function: def get_next_primitive(states, actions, curr_name, seq):

The online procedure is summarized in Figure 3: CHECKSWITCH evaluates the causal completion condition, CALCSTATISTICS computes motion statistics, and GETCONDITION(P) selects the primitive-specific branch. The generated wrapper retains the current primitive when the condition is false and returns the next entry when it is true.

Task Termination. The final primitive remains active until the episode terminates, allowing continued closed-loop execution to correct incomplete attempts. Task success is determined by the simulator in simulation and verified manually in real-world experiments, rather than by the primitive completion signal. Each episode ends upon success or when its maximum execution budget is reached.

A.5 GENERATED PSEUDOCODE AND CODE CORRESPONDENCE

This subsection summarizes the schematic pseudocode for the offline segmentation and online switching programs generated from the corresponding LLM prompts in Appendix A.2 and Appendix A.4, respectively, and maps their abstractions to the generated Python implementations. The pseudocode is explanatory; implementations may inline branches or use helper functions.

Algorithm 1: Primitive Segmentation

Input: Trajectory T, Primitive P, start t_0
function SEGMENTTRAJECTORY(T, P, t_0)
  if P is last **then return** $|T| - 1$
  $\Phi \leftarrow$ GETRULE(P), $\delta \leftarrow 10$
  for $t = t_0 + \delta$ **to** $|T| - 1$ **do**
    if $\Phi(T, t)$ **then return** t
  end for
  return Null

Algorithm 2: Primitive Switching

Input: History Window $\mathcal{H}_t$, Primitive P
function CHECKSWITCH($\mathcal{H}_t$, P)
  if $|\mathcal{H}_t| <$ WindowSize **then return** False
  $s_{\text{stat}} \leftarrow$ CALCSTATISTICS($\mathcal{H}_t$)
  $\Phi \leftarrow$ GETCONDITION(P)
  if $\Phi(s_{\text{stat}})$ **then return** True
  end if
  return False

Figure 3: **Offline segmentation and online switching.** Explanatory pseudocode for the generated programs G_{seg} and G_{switch}. SEGMENTTRAJECTORY may inspect past and future states/actions, whereas CHECKSWITCH is generated once before execution and evaluates causal history. GETRULE/GETCONDITION select primitive-specific criteria, and CALCSTATISTICS computes motion statistics. The corresponding code is given in Table 7.

Table 7: **Pseudocode-to-code correspondence.** Correspondence between the abstractions in Figure 3 and the generated Python programs. A generated program may implement an entry as an inline branch or helper function; the pseudocode names are explanatory and are not required API names.

Pseudocode abstraction	Generated-code counterpart	Role
Offline segmentation (G_{seg})		
SEGMENTTRAJECTORY	`get_segmentation_index`	Scans candidate frames after the minimum-duration offset and returns the primitive boundary.
GETRULE(P)	Primitive-specific branch selected by `primitive_name`	Selects the offline completion criterion for P.
Online switching (G_{switch})		
CHECKSWITCH	Completion-decision block in `get_next_primitive`	Evaluates the selected condition; the generated wrapper retains the current primitive or returns the next entry in the execution flow.
CALCSTATISTICS($\mathcal{H}_t$)	Inline or helper calculations	Computes end-effector displacement, gripper state, and motion trends over the causal history window.
GETCONDITION(P)	Branch selected by P: `check_[primitive]_condition`	Selects the completion criterion for P, evaluated over the causal history window.

A.6 UNIFIED MULTIMODAL REPRESENTATION

To harmonize diverse motion patterns across different tasks, we introduce the Unified Multimodal Representation (UMR) as a standardized interface.

1. Motivation for UMR

Without UMR, primitive instructions retain instance-specific object names, target names, and spatial relations. The same primitive then appears through different language inputs across tasks, such as "grasp the black bowl" and "grasp the red cup." This variation expands the instruction space and makes shared motion patterns less explicit across tasks. However, directly replacing these descriptions with a unified instruction removes the instance-specific grounding required for execution: "grasp the object," for example, does not identify which object should be manipulated. UMR therefore aligns shared primitive semantics while preserving the concrete object and target information needed for execution.

UMR divides these roles between modalities: unified language expresses the primitive and retained execution attributes, while object masks ground the referenced entities and their locations in the unchanged RGB scene. Table 15 empirically compares retaining instance-specific language, removing it without grounding, and combining unified language with masks.

2. Instruction Mapping

When UMR is active, each instance-specific primitive instruction is mapped to the base instruction template for its primitive type, with necessary execution attributes retained. Table 8 lists these base templates. For example, "twist the knob clockwise" becomes "Twist the grasped object clockwise," while instructions requiring no additional attributes use the template directly.

Table 8: **UMR primitive instruction templates.** The listed instructions are base templates. Optional execution attributes, such as spatial relations, approach directions, rotation directions, or terminal states, are appended when required by the task. `color_1` and `color_2` are custom color indices representing the semantic masks of the manipulated object and the target location/object, respectively.

Primitive Type	Base Unified Primitive Instruction
grasp	"Grasp the masked object with `color_1`"
lift	"Lift the object with `color_1`"
move	"Move to above the object with `color_2`"
place	"Place in the object with `color_2`"
push	"Push the object with `color_1`"
press	"Press the object with `color_2`"
pull	"Pull the object with `color_1`"
insert	"Insert into the area with `color_2`"
twist	"Twist the grasped object"
rotate	"Rotate the object with `color_1`"
tilt	"Tilt the object with `color_1`"

For example, consider the instruction "pick up the right black bowl and place it in the tray." The VLM produces the primitive sequence $[grasp, move, place]$. UMR maps these primitives to "Grasp the masked object with `color_1`," "Move to above the object with `color_2`," and "Place in the object with `color_2`," where `color_1` identifies the right black bowl and `color_2` identifies the tray. The "right" relation is used during grounding to select the correct object mask, while any additional direction or terminal-state attribute is retained in the primitive instruction when required.

3. Visual Grounding and Tracking

For each task, we first extract the target objects mentioned in the original instruction. We utilize the Qwen2.5-VL-72B-Instruct (Team, 2025a) model to perform zero-shot object detection. The model is queried with a specific prompt to provide the spatial coordinates of the objects:

System Prompt: Visual Object Detection

```
[System Role]
You are an object detection agent. Given an image, your task is to
identify and provide the coordinates of a specific target object.

[Instruction]
Find the bounding box of the following object: "{obj}".

[Input]
The image is: [Visual input of the current environment]

[Output Format]
You must respond strictly in JSON format as follows:

{
  "bbox": [x1, y1, x2, y2]
}

Note: The coordinates [x1, y1, x2, y2] represent the top-left and
bottom-right corners of the bounding box normalized to the image
dimensions.
```

The VLM returns a JSON-formatted bounding box $[x_1, y_1, x_2, y_2]$, which identifies the initial location of the task-relevant object (assigned as `color_1`) or the target area (assigned as `color_2`).

Once the bounding box is obtained, the system proceeds with the following steps to ensure spatial consistency:

- **Segmentation**: The bounding box is passed to the Segment Anything Model (SAM) (Kirillov et al., 2023), which generates a high-fidelity initial mask for the object.
- **Continuous Tracking**: To handle dynamic scenes and robot movements, the Cutie (Cheng et al., 2024) video object segmentation model is employed. Cutie tracks the initial mask throughout the entire trajectory, providing real-time mask updates with minimal latency.

This multi-stage visual pipeline ensures both high-quality data generation and real-time execution efficiency through a decoupled deployment strategy.

For offline data processing, this automated workflow allows us to convert large-scale raw primitive trajectories into UMR-formatted trajectories with high semantic precision, eliminating the need for manual annotation.

For online inference, the computational overhead is strategically distributed: Qwen2.5-VL-72B-Instruct and SAM are invoked only once during pre-execution initialization to ground the task context. Once execution begins, mask tracking is handed over to Cutie. This design keeps the per-step overhead of UMR small, as quantified in Appendix D.8, while maintaining responsive control and alignment with task-specific visual observations in OOD settings.

By combining these tracked masks with unified primitive instructions, the UMR provides a standardized interface where the visual information within the mask supplies the necessary task context, ensuring the VLA can understand "where and what" to act upon despite the simplified text input.

B Experiment Setting Details

B.1 Detailed Task Lists

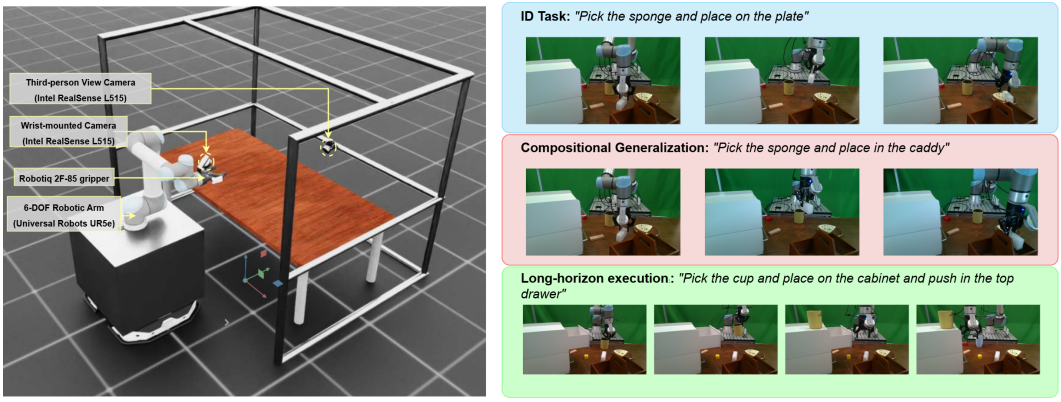


Figure 4: Real-world setup and representative executions. Left: the robot, third-person camera, wrist camera, and tabletop workspace. Right: examples from the in-distribution, compositional generalization, and long-horizon execution settings.

The tasks utilized in our experiments span various simulation environments and real-world scenarios (summarized in Table 9). To provide a clear mapping between high-level instructions and executable actions, we also include the disassembly results (i.e., the corresponding primitive sequences) for each task in the same table. The task design logic is as follows:

- **In-Distribution and Large-scale fine-tuning Benchmarks (Simulation ID):** This category includes `Libero-Spatial`, `Libero-Object`, `Libero-Goal` and `Libero-90` (Liu et al., 2023). Across these suites, `LIBERO-90` serves as the primary benchmark for large-scale fine-tuning. We use 83 tasks for fine-tuning after data quality filtering. Table 10 lists all the 90 tasks and their primitive sequences, with tasks excluded from fine-tuning marked by an asterisk. Models fine-tuned on this subset are evaluated on both in-distribution tasks and broader generalization benchmarks.
- **Complex Distribution Adaptation Evaluation (RLBench):** Consistent with the fine-tuning evaluation protocol of Libero, RLBench (James et al., 2020) also adopts a "fine-tune

then test" paradigm. Given that the object distribution in Libero is relatively consistent, we introduce RLBench to examine the model's ability to learn and execute tasks in environments with higher complexity, more diverse distributions, and multifaceted interaction types (e.g., involving higher degrees of freedom).

- **Compositional Generalization Benchmarks (OOD):** `Libero-90-Novel` and `Libero-Long` evaluate zero-shot transfer to unseen task compositions after fine-tuning only on `Libero-90`. For PrimitiveVLA, the prompt templates, Disassembly Library, and switching program remain fixed from the Libero-90 setting across both evaluations.
  - **`Libero-90-Novel`:** Designed by recombining scenes and objects from Libero-90, this suite challenges the model to perform tasks on novel objects, execute long-horizon tasks formed by stitching two sub-tasks through a common component, and perform existing tasks under varying environmental configurations.
  - **`Libero-Long` (Libero-10):** This benchmark evaluates long-horizon execution through ordered assembly of primitives learned from `Libero-90`. We evaluate eight tasks whose constituent subtasks are represented in the fine-tuning subset; the remaining two involve unseen subtasks and are excluded. All methods reset the arm to its initial pose only between subtasks, preserving object states. PrimitiveVLA detects these transitions through its execution flow and switching controller; baselines use the same flow-based switching logic internally for reset detection, without conditioning their action policies on the flow.
- **Real-world Experiments (Real-world Exp):** Conducted in a laboratory setting to validate performance in physical scenarios. As illustrated in Fig. 4, we present the robot, multi-view cameras, tabletop workspace, and example rollouts. Six tasks provide in-distribution data; five additional tasks are evaluated zero-shot to assess compositional generalization and long-horizon execution. Prompt templates, the Disassembly Library, and the switching program are shared across these settings.

Execution Budgets. For LIBERO-Spatial, Object, Goal, and Long, we follow the corresponding OpenVLA evaluation step limits. LIBERO-90 and LIBERO-90-Novel use a maximum of 400 environment steps, reflecting their intermediate task horizons. RLBench also uses a maximum of 400 environment steps.

B.2 Training Data Statistics

Data Collection and Processing. The training dataset is consolidated from three primary sources, each with specific collection and cleaning protocols:

- **Libero Datasets:** Simulation data are sourced from the official Libero benchmark. We exclusively apply the OpenVLA (Kim et al., 2025b) data cleaning protocol to these subsets to filter out unsuccessful attempts and redundant segments, ensuring only high-quality successful trajectories are retained.
- **RLBench:** Data are generated using native demonstration scripts, with 60 successful trajectories per task and 600 trajectories in total.
- **Real-world Data:** A total of 160 trajectories were human-collected in a laboratory setting. This includes 30 trajectories for each of the 4 manipulation tasks (Pick sponge, Pick cup, Pick cube in drawer, Pick cube in caddy) and 20 trajectories for each of the 2 simpler *push* tasks.

Half-Data Protocol. For the 50% data setting, we retain the first half of the original demonstrations within each task. The baseline and PrimitiveVLA use the same retained trajectories, with primitive segmentation applied only for PrimitiveVLA. Retrieval exemplars and segmentation calibration examples are also drawn exclusively from the sampled 50% of demonstrations. Within each backbone, the 50% and 100% settings use the same optimization steps and batch size. All other preprocessing, representation, and execution procedures remain unchanged.

Post-Segmentation Statistics. To facilitate primitive-level learning, we apply the generated segmentation program to successful trajectories. When the segmentation program fails to identify a valid termination for a primitive, we retain the valid segments before that point and discard the remainder of

Table 9: Representative task list and primitive sequences for all benchmarks. The complete LIBERO-90 inventory is provided in Table 10.

Suite	Task Instruction (Full Name)	Primitive Sequence
Libero-Spatial	pick up the black bowl between the plate and the ramekin and place it on the plate	[grasp, move, place]
	pick up the black bowl from table center and place it on the plate	[grasp, move, place]
	pick up the black bowl in the top drawer of the wooden cabinet and place it on the plate	[grasp, move, place]
	pick up the black bowl next to the cookie box and place it on the plate	[grasp, move, place]
	pick up the black bowl next to the plate and place it on the plate	[grasp, move, place]
	pick up the black bowl next to the ramekin and place it on the plate	[grasp, move, place]
	pick up the black bowl on the cookie box and place it on the plate	[grasp, move, place]
	pick up the black bowl on the ramekin and place it on the plate	[grasp, move, place]
	pick up the black bowl on the stove and place it on the plate	[grasp, move, place]
	pick up the black bowl on the wooden cabinet and place it on the plate	[grasp, move, place]
Libero-Object	pick up the alphabet soup and place it in the basket	[grasp, move, place]
	pick up the bbq sauce and place it in the basket	[grasp, move, place]
	pick up the butter and place it in the basket	[grasp, move, place]
	pick up the chocolate pudding and place it in the basket	[grasp, move, place]
	pick up the cream cheese and place it in the basket	[grasp, move, place]
	pick up the ketchup and place it in the basket	[grasp, move, place]
	pick up the milk and place it in the basket	[grasp, move, place]
	pick up the orange juice and place it in the basket	[grasp, move, place]
	pick up the salad dressing and place it in the basket	[grasp, move, place]
	pick up the tomato sauce and place it in the basket	[grasp, move, place]
Libero-Goal	open the middle drawer of the cabinet	[pull]
	open the top drawer and put the bowl inside	[pull, grasp, lift, place]
	push the plate to the front of the stove	[press, push]
	put the bowl on the plate	[grasp, move, place]
	put the bowl on the stove	[grasp, move, place]
	put the bowl on top of the cabinet	[grasp, lift, place]
	put the cream cheese in the bowl	[grasp, move, place]
	put the wine bottle on the rack	[grasp, lift, place]
	put the wine bottle on top of the cabinet	[grasp, lift, place]
	turn on the stove	[grasp, twist]
Libero-90	LIVING_ROOM_SCENE1_put_the_black_bowl_on_the_plate	[grasp, move, place]
	LIVING_ROOM_SCENE4_stack_left_bowl_on_right_bowl_and_place_them_in_tray	[grasp, move, place, grasp, move, place]
	KITCHEN_SCENE3_turn_on_the_stove_and_put_the_frying_pan_on_it	[grasp, twist, grasp, move, place]
	KITCHEN_SCENE10_put_the_butter_in_the_top_drawer_and_close_it	[grasp, lift, place, push]
	STUDY_SCENE4_put_the_middle_book_on_the_shelf	[grasp, lift, tilt]
	... (Total 83 tasks used for fine-tuning)	...
Libero-90-Novel	KITCHEN_SCENE1_open_the_top_drawer_put_the_black_bowl_in_it_and_close_it	[pull, grasp, lift, place, push]
	KITCHEN_SCENE2_stack_the_front_bowl_on_the_back_bowl	[grasp, move, place]
	KITCHEN_SCENE10_open_the_top_drawer_put_the_chocolate_pudding_in_it_and_close_it	[pull, grasp, lift, place, push]
	KITCHEN_SCENE10_close_the_middle_drawer	[push]
	KITCHEN_SCENE10_put_the_black_bowl_on_the_wooden_cabinet	[grasp, lift, place]
	LIVING_ROOM_SCENE4_pick_up_the_right_bowl_and_place_it_in_the_tray	[grasp, move, place]
	LIVING_ROOM_SCENE5_put_the_yellow_and_white_mug_on_the_left_plate	[grasp, move, place]
	LIVING_ROOM_SCENE2_pick_up_the_butter_and_place_it_in_the_basket	[grasp, move, place]
Libero-Long	turn on the stove and put the moka pot on it	[grasp, twist, grasp, move, place]
	put the black bowl in the bottom drawer of the cabinet and close it	[grasp, move, place, push]
	put both moka pots on the stove	[grasp, move, place, grasp, move, place]
	put both the alphabet soup and the cream cheese box in the basket	[grasp, move, place, grasp, move, place]
	put both the alphabet soup and the tomato sauce in the basket	[grasp, move, place, grasp, move, place]
	put both the cream cheese box and the butter in the basket	[grasp, move, place, grasp, move, place]
	put the white mug on the left plate and put the yellow and white mug on the right plate	[grasp, move, place, grasp, move, place]
	put the white mug on the plate and put the chocolate pudding to the right of the plate	[grasp, move, place, grasp, move, place]
RLBench	meat_off_grill	[grasp, move, place]
	lamp_off	[move, press]
	push_button	[move, press]
	open_jar	[grasp, twist, move, place]
	lift_numbered_block	[grasp, lift]
	open_wine_bottle	[grasp, twist, lift]
	take_off_weighing_scales	[grasp, lift, move, place]
	take_lid_off_saucepan	[grasp, lift]
	put_rubbish_in_bin	[grasp, lift, place]
	take_umbrella_out_of_umbrella_stand	[grasp, lift]
Real-World	**In-Distribution (ID):**	
	Pick sponge and place on the plate	[grasp, move, place]
	Pick cup and place on the top of cabinet	[grasp, lift, place]
	Pick cube and place in the top drawer	[grasp, lift, place]
	Pick cube and place in the left caddy	[grasp, move, place]
	Push in the top drawer	[push]
	Push in the bottom drawer	[push]
	Compositional Generalization:	
	Pick sponge and place in the left caddy (unseen pair)	[grasp, move, place]
	Pick cube and place on the top of cabinet (unseen pair)	[grasp, lift, place]
	Pick the blue cup and place on the top of cabinet (novel object)	[grasp, lift, place]
	Long-Horizon Execution:	
	Pick cup then push drawer	[grasp, lift, place, push]
	Pick sponge and place on the plate, then Pick cube and place in the left caddy	[grasp, move, place, grasp, move, place]

Table 10: All 90 LIBERO-90 tasks with VLM-generated primitive sequences.

ID	Task Instruction (Full Name)	Primitive Sequence
1	LIVING_ROOM_SCENE1_pick_up_the_alphabet_soup_and_put_it_in_the_basket	[grasp, move, place]
2	LIVING_ROOM_SCENE2_pick_up_the_alphabet_soup_and_put_it_in_the_basket	[grasp, move, place]
3	LIVING_ROOM_SCENE3_pick_up_the_alphabet_soup_and_put_it_in_the_tray	[grasp, move, place]
4	LIVING_ROOM_SCENE4_pick_up_the_chocolate_pudding_and_put_it_in_the_tray	[grasp, move, place]
5	LIVING_ROOM_SCENE1_pick_up_the_cream_cheese_box_and_put_it_in_the_basket	[grasp, move, place]
6	LIVING_ROOM_SCENE3_pick_up_the_cream_cheese_and_put_it_in_the_tray	[grasp, move, place]
7	LIVING_ROOM_SCENE1_pick_up_the_ketchup_and_put_it_in_the_basket	[grasp, move, place]
8	LIVING_ROOM_SCENE3_pick_up_the_ketchup_and_put_it_in_the_tray	[grasp, move, place]
9*	LIVING_ROOM_SCENE2_pick_up_the_butter_and_put_it_in_the_basket	[grasp, move, place]
10	LIVING_ROOM_SCENE3_pick_up_the_butter_and_put_it_in_the_tray	[grasp, move, place]
11	LIVING_ROOM_SCENE2_pick_up_the_milk_and_put_it_in_the_basket	[grasp, move, place]
12	LIVING_ROOM_SCENE2_pick_up_the_orange_juice_and_put_it_in_the_basket	[grasp, move, place]
13	LIVING_ROOM_SCENE4_pick_up_the_salad_dressing_and_put_it_in_the_tray	[grasp, move, place]
14	LIVING_ROOM_SCENE1_pick_up_the_tomato_sauce_and_put_it_in_the_basket	[grasp, move, place]
15	LIVING_ROOM_SCENE2_pick_up_the_tomato_sauce_and_put_it_in_the_basket	[grasp, move, place]
16	LIVING_ROOM_SCENE3_pick_up_the_tomato_sauce_and_put_it_in_the_tray	[grasp, move, place]
17	LIVING_ROOM_SCENE4_pick_up_the_black_bowl_on_the_left_and_put_it_in_the_tray	[grasp, move, place]
18	LIVING_ROOM_SCENE4_stack_the_left_bowl_on_the_right_bowl_and_place_them_in_the_tray	[grasp, move, place, grasp, move, place]
19	LIVING_ROOM_SCENE4_stack_the_right_bowl_on_the_left_bowl_and_place_them_in_the_tray	[grasp, move, place, grasp, move, place]
20	LIVING_ROOM_SCENE5_put_the_red_mug_on_the_left_plate	[grasp, move, place]
21	LIVING_ROOM_SCENE5_put_the_red_mug_on_the_right_plate	[grasp, move, place]
22	LIVING_ROOM_SCENE5_put_the_white_mug_on_the_left_plate	[grasp, move, place]
23	LIVING_ROOM_SCENE5_put_the_yellow_and_white_mug_on_the_right_plate	[grasp, move, place]
24	LIVING_ROOM_SCENE6_put_the_chocolate_pudding_to_the_right_of_the_plate	[grasp, move, place]
25	LIVING_ROOM_SCENE6_put_the_red_mug_on_the_plate	[grasp, move, place]
26	LIVING_ROOM_SCENE6_put_the_white_mug_on_the_plate	[grasp, move, place]
27	KITCHEN_SCENE1_put_the_black_bowl_on_the_plate	[grasp, move, place]
28	KITCHEN_SCENE3_put_the_frying_pan_on_the_stove	[grasp, move, place]
29	KITCHEN_SCENE3_put_the_moka_pot_on_the_stove	[grasp, move, place]
30	KITCHEN_SCENE1_put_the_black_bowl_on_top_of_the_cabinet	[grasp, lift, place]
31	KITCHEN_SCENE2_put_the_black_bowl_at_the_back_on_the_plate	[grasp, move, place]
32	KITCHEN_SCENE2_put_the_black_bowl_at_the_front_on_the_plate	[grasp, move, place]
33	KITCHEN_SCENE2_put_the_middle_black_bowl_on_the_plate	[grasp, move, place]
34	KITCHEN_SCENE2_put_the_middle_black_bowl_on_top_of_the_cabinet	[grasp, lift, place]
35	KITCHEN_SCENE2_stack_the_black_bowl_at_the_front_on_the_black_bowl_in_the_middle	[grasp, move, place]
36	KITCHEN_SCENE2_stack_the_middle_black_bowl_on_the_back_black_bowl	[grasp, move, place]
37	KITCHEN_SCENE4_put_the_black_bowl_in_the_bottom_drawer_of_the_cabinet	[grasp, move, place]
38	KITCHEN_SCENE4_put_the_black_bowl_on_top_of_the_cabinet	[grasp, lift, place]
39	KITCHEN_SCENE4_put_the_wine_bottle_on_the_wine_rack	[grasp, lift, place]
40	KITCHEN_SCENE3_turn_on_the_stove	[grasp, twist]
41	KITCHEN_SCENE3_turn_on_the_stove_and_put_the_frying_pan_on_it	[grasp, twist, grasp, move, place]
42	KITCHEN_SCENE1_open_the_top_drawer_of_the_cabinet_and_put_the_bowl_in_it	[pull, grasp, lift, place]
43	KITCHEN_SCENE5_put_the_black_bowl_in_the_top_drawer_of_the_cabinet	[grasp, lift, place]
44	KITCHEN_SCENE5_put_the_black_bowl_on_the_plate	[grasp, move, place]
45	KITCHEN_SCENE5_put_the_black_bowl_on_top_of_the_cabinet	[grasp, lift, place]
46	KITCHEN_SCENE7_put_the_white_bowl_on_the_plate	[grasp, move, place]
47	KITCHEN_SCENE7_put_the_white_bowl_to_the_right_of_the_plate	[grasp, move, place]
48	KITCHEN_SCENE8_put_the_right_moka_pot_on_the_stove	[grasp, move, place]
49	KITCHEN_SCENE8_turn_off_the_stove	[grasp, twist]
50	KITCHEN_SCENE9_put_the_frying_pan_on_the_cabinet_shelf	[grasp, lift, insert]
51	KITCHEN_SCENE9_put_the_frying_pan_on_top_of_the_cabinet	[grasp, lift, place]
52	KITCHEN_SCENE9_put_the_frying_pan_under_the_cabinet_shelf	[grasp, insert]
53	KITCHEN_SCENE9_put_the_white_bowl_on_top_of_the_cabinet	[grasp, lift, place]
54	KITCHEN_SCENE9_turn_on_the_stove	[grasp, twist]
55	KITCHEN_SCENE9_turn_on_the_stove_and_put_the_frying_pan_on_it	[grasp, twist, grasp, move, place]
56	KITCHEN_SCENE10_close_the_top_drawer_of_the_cabinet_and_put_the_black_bowl_on_top_of_it	[push, grasp, lift, place]
57	KITCHEN_SCENE10_put_the_black_bowl_in_the_top_drawer_of_the_cabinet	[grasp, lift, place]
58	KITCHEN_SCENE10_put_the_butter_at_the_back_in_the_top_drawer_of_the_cabinet_and_close_it	[grasp, lift, place, push]
59	KITCHEN_SCENE10_put_the_butter_at_the_front_in_the_top_drawer_of_the_cabinet_and_close_it	[grasp, lift, place, push]
60	KITCHEN_SCENE10_put_the_chocolate_pudding_in_the_top_drawer_of_the_cabinet_and_close_it	[grasp, lift, place, push]
61	STUDY_SCENE1_pick_up_the_book_and_place_it_in_the_front_compartment_of_the_caddy	[grasp, lift, insert]
62	STUDY_SCENE1_pick_up_the_book_and_place_it_in_the_left_compartment_of_the_caddy	[grasp, lift, insert]
63	STUDY_SCENE1_pick_up_the_book_and_place_it_in_the_right_compartment_of_the_caddy	[grasp, lift, insert]
64	STUDY_SCENE2_pick_up_the_book_and_place_it_in_the_back_compartment_of_the_caddy	[grasp, lift, insert]
65	STUDY_SCENE2_pick_up_the_book_and_place_it_in_the_front_compartment_of_the_caddy	[grasp, lift, insert]
66	STUDY_SCENE2_pick_up_the_book_and_place_it_in_the_left_compartment_of_the_caddy	[grasp, lift, insert]
67	STUDY_SCENE2_pick_up_the_book_and_place_it_in_the_right_compartment_of_the_caddy	[grasp, lift, insert]
68	STUDY_SCENE3_pick_up_the_book_and_place_it_in_the_front_compartment_of_the_caddy	[grasp, lift, insert]
69	STUDY_SCENE3_pick_up_the_book_and_place_it_in_the_left_compartment_of_the_caddy	[grasp, lift, insert]
70	STUDY_SCENE3_pick_up_the_book_and_place_it_in_the_right_compartment_of_the_caddy	[grasp, lift, insert]
71	STUDY_SCENE1_pick_up_the_yellow_and_white_mug_and_place_it_to_the_right_of_the_caddy	[grasp, move, place]
72	STUDY_SCENE3_pick_up_the_red_mug_and_place_it_to_the_right_of_the_caddy	[grasp, move, place]
73	STUDY_SCENE3_pick_up_the_white_mug_and_place_it_to_the_right_of_the_caddy	[grasp, move, place]
74	STUDY_SCENE4_pick_up_the_book_in_the_middle_and_place_it_on_the_cabinet_shelf	[grasp, lift, tilt]
75	STUDY_SCENE4_pick_up_the_book_on_the_left_and_place_it_on_top_of_the_shelf	[grasp, lift, tilt]
76	STUDY_SCENE4_pick_up_the_book_on_the_right_and_place_it_on_the_cabinet_shelf	[grasp, tilt]
77	STUDY_SCENE4_pick_up_the_book_on_the_right_and_place_it_under_the_cabinet_shelf	[grasp, tilt]
78	KITCHEN_SCENE1_open_the_bottom_drawer_of_the_cabinet	[pull]
79	KITCHEN_SCENE1_open_the_top_drawer_of_the_cabinet	[pull]
80	KITCHEN_SCENE2_open_the_top_drawer_of_the_cabinet	[pull]
81	KITCHEN_SCENE4_close_the_bottom_drawer_of_the_cabinet	[push]
82	KITCHEN_SCENE4_close_the_bottom_drawer_of_the_cabinet_and_open_the_top_drawer	[push, pull]
83	KITCHEN_SCENE5_close_the_top_drawer_of_the_cabinet	[push]
84	KITCHEN_SCENE10_close_the_top_drawer_of_the_cabinet	[push]
85*	KITCHEN_SCENE7_open_the_microwave	[rotate]
86*	KITCHEN_SCENE5_put_the_ketchup_in_the_top_drawer_of_the_cabinet	[grasp, lift, tilt]
87*	KITCHEN_SCENE6_close_the_microwave	[grasp, move, place, rotate]
88*	KITCHEN_SCENE6_put_the_yellow_and_white_mug_to_the_front_of_the_white_mug	[grasp, move, place]
89*	KITCHEN_SCENE4_put_the_wine_bottle_in_the_bottom_drawer_of_the_cabinet	[grasp, tilt]
90*	LIVING_ROOM_SCENE6_put_the_chocolate_pudding_to_the_left_of_the_plate	[grasp, move, place]

* indicates tasks that contribute no fine-tuning samples due to unavailable demonstrations or demonstration/segmentation quality issues. Primitive sequences are VLM-generated task decompositions and do not imply that the corresponding task demonstrations were used for fine-tuning.

the trajectory. Each primitive segment is treated as a separate trajectory. Sampling and action-chunk handling at segment boundaries follow the corresponding backbone baseline's trajectory sampling and end-of-trajectory rules. Table 11 summarizes the training trajectories and resulting primitive counts.

Table 11: Training trajectories and resulting primitive counts.

Benchmark	Original trajectories	Primitive segments	Processing source
Libero-Object	456	1,368	Independent cleaning via OpenVLA code
Libero-Spatial	432	1,293	Independent cleaning via OpenVLA code
Libero-Goal	428	1,121	Independent cleaning via OpenVLA code
Libero-90	3,688	10,546	Independent cleaning via OpenVLA code
RLBench	600	1,616	Script-generated (60/task)
Real-world ID	160	400	Human-collected (20–30/task)
Total	**5,764**	**16,344**	–

Primitive Distribution. The consolidated training sets exhibit a long-tail distribution across different primitive categories (Fig. 5). Spatial-transport primitives (Grasp, Place, Move, and Lift) account for most segments, while the remaining interaction types form the long tail.

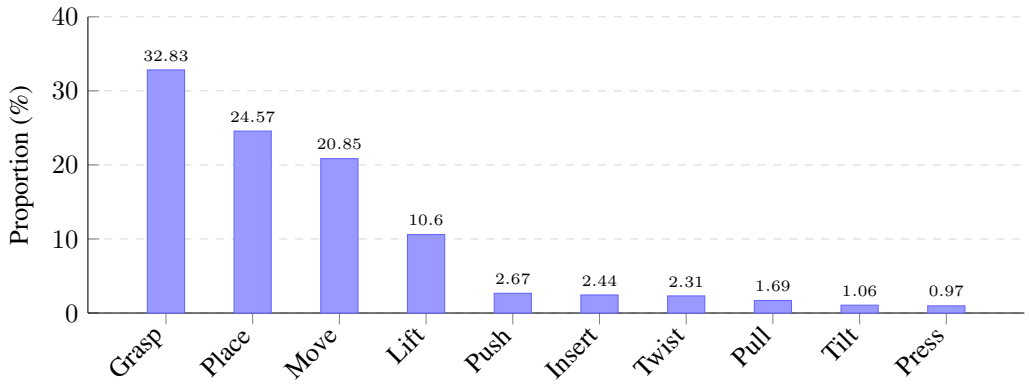


Figure 5: Primitive distribution across the consolidated training sets, ordered from most to least frequent.

B.3 Model Settings

Model Configurations. We evaluate our framework using three distinct VLA model configurations: OpenVLA (Kim et al., 2025b), OpenVLA-OFT (Kim et al., 2025a) (Optimized Fine-Tuning variant of OpenVLA), and $\pi_{0.5}$ (Black et al., 2025). All models primarily follow their official default configurations, with specific adjustments to batch size, training iterations, and action chunk sizes as detailed below:

- **OpenVLA:** As a single-step prediction model, we focus on training iterations. For the `Libero-Spatial`, `Object`, and `Goal` benchmarks, we follow the official fine-tuning setting of 50k iterations. For `Libero-90` and `RLBench`, we increase the training to 100k iterations to accommodate the significantly larger dataset scale and increased task complexity. The fine-tuning command uses a `batch_size` of 4 with 4 gradient accumulation steps.
- **OpenVLA-OFT (Optimized Fine-Tuning):** This variant employs an optimized fine-tuning strategy tailored for Libero. We set the training duration to 150k iterations for both Libero and RLBench, with an action chunk size of 5, a `batch_size` of 2, and 4 gradient accumulation steps.

- $\pi_{0.5}$**:** Based on the default training scale for Libero, $\pi_{0.5}$ is trained for 30k iterations across all simulation and real-world benchmarks. The `batch_size` parameter is set to 56. We use an action chunk size of 10 for simulation, which is increased to 20 in real-world experiments to enhance the model's robustness against physical perturbations and environmental noise inherent in live execution.

All fine-tuning for OpenVLA and OpenVLA-OFT was conducted on an 8-GPU L40S(48g) server, taking 60 and 100 hours, respectively. Fine-tuning for $\pi_{0.5}$ was completed on an 8-GPU A100(80g) server in 20 hours. For Libero benchmarks, evaluations were performed on the same hardware used for fine-tuning (L40S for OpenVLA/OpenVLA-OFT and A100 for $\pi_{0.5}$). In contrast, all RLBench and real-world experiments were carried out on an RTX 4090.

We do not quantify variability across training seeds; small success-rate differences should therefore be interpreted cautiously.

C Additional Experiments

C.1 Component Ablations

The main paper reports coarse-grained system ablations in Table 5. Here, we provide more fine-grained analyses of the three main components of PrimitiveVLA: primitive disassembly, unified multimodal representation, and primitive switching. Primitive disassembly converts task demonstrations into primitive-aligned training units and depends on reliable primitive labels and temporal boundaries. Unified multimodal representation describes the motion at the primitive level while grounding the manipulated object or target. Primitive switching assembles the ordered primitive flow during execution and determines when the controller advances to the next primitive.

These components are functionally coupled through the training–execution interface. Primitive-level training organizes the units used by the execution flow, and primitive switching operates on these units. Therefore, a fully factorial ablation that removes primitive-level training while retaining primitive switching would introduce an artificial interface mismatch. We instead evaluate each component within a valid interface: Appendix C.1.1 compares alternative boundary-segmentation methods; Appendix C.1.2 compares instruction and visual-grounding representations; and Appendix C.1.3 compares VLA-predicted and code-controlled switching under the same primitive flow and representation.

Unless otherwise stated, OpenVLA is used as the fine-tuning backbone, and all ablations follow the training and evaluation protocol described in the main paper.

C.1.1 Primitive Disassembly and Segmentation

A useful primitive dataset requires both a correct ordered primitive sequence and reliable temporal boundaries. We therefore evaluate disassembly at two levels: annotation-based segmentation quality and downstream policy performance. The primitive sequence and downstream representation are kept fixed across segmentation methods, so this comparison focuses on the effect of temporal boundary construction.

We manually annotated 180 trajectories sampled from 18 Libero-90 tasks, with 10 trajectories per task and 590 ground-truth segment-end boundaries in total, including the final endpoint of each trajectory. One author labeled the primitive identities and segment-end boundaries, and another author reviewed the annotations. These annotations were used only for evaluation, not to tune the segmentation program or its thresholds. The 18 tasks used in this evaluation, listed in Table 12, collectively cover all primitive types present in the LIBERO-90 fine-tuning dataset. The VLM-generated primitive labels achieve 97.3% segment-level accuracy against these annotations.

For boundary localization, predicted and ground-truth boundaries are matched one-to-one within a tolerance of ± 7 frames. Precision, recall, and F1 are computed from the pooled true-positive, false-positive, and false-negative counts across all annotated trajectories. Table 13 compares our state/action-based method with direct VLM segmentation using Qwen3.5-Omni-Plus (Qwen Team, 2026) and Universal Visual Decomposer (UVD) (Zhang et al., 2024b). Count Error is the signed difference between the predicted and ground-truth (GT) segment counts (Pred. $-$ GT).

Table 12: The 18 Libero-90 tasks used to evaluate primitive disassembly and the resulting downstream policies.

#	Scene	Task
1	LIVING_ROOM_SCENE4	pick up the chocolate pudding and put it in the tray
2	LIVING_ROOM_SCENE4	stack the left bowl on the right bowl and place them in the tray
3	KITCHEN_SCENE2	put the middle black bowl on top of the cabinet
4	KITCHEN_SCENE3	turn on the stove and put the frying pan on it
5	KITCHEN_SCENE1	open the top drawer of the cabinet and put the bowl in it
6	KITCHEN_SCENE10	put the butter at the back in the top drawer of the cabinet and close it
7	STUDY_SCENE4	pick up the book in the middle and place it on the cabinet shelf
8	STUDY_SCENE1	pick up the book and place it in the front compartment of the caddy
9	KITCHEN_SCENE9	put the frying pan on the cabinet shelf
10	STUDY_SCENE3	pick up the white mug and place it to the right of the caddy
11	KITCHEN_SCENE4	close the bottom drawer of the cabinet and open the top drawer
12	STUDY_SCENE4	pick up the book on the right and place it under the cabinet shelf
13	LIVING_ROOM_SCENE6	put the chocolate pudding to the right of the plate
14	LIVING_ROOM_SCENE2	pick up the milk and put it in the basket
15	LIVING_ROOM_SCENE5	put the white mug on the left plate
16	KITCHEN_SCENE8	put the right moka pot on the stove
17	KITCHEN_SCENE10	close the top drawer of the cabinet and put the black bowl on top of it
18	KITCHEN_SCENE3	turn on the stove

Table 13: Primitive-boundary localization on 180 manually annotated Libero-90 trajectories with 590 boundaries.

Method	Precision@$\pm$7f	Recall@$\pm$7f	F1@$\pm$7f	Count Error
Qwen3.5-Omni-Plus	42.71%	42.71%	42.71%	0
UVD	33.90%	50.34%	40.52%	+286
PrimitiveVLA (ours)	**95.40%**	**94.92%**	**95.16%**	−3

We compare three ways of obtaining the boundaries. For each trajectory, the same VLM first generates an ordered primitive sequence. UVD then performs pure visual segmentation from the image sequence and aligns the resulting segments with the generated primitive order afterward. Qwen3.5-Omni-Plus receives the primitive sequence, the complete image sequence, and the state sequence, and then predicts the segmentation points. PrimitiveVLA uses an LLM to generate the state/action code described in the main paper. The generated code is then executed on the state trajectory to locate the transitions between primitives.

As illustrated in Table 13, PrimitiveVLA reaches 95.16% F1 and predicts only three fewer segments than the annotations. Qwen3.5-Omni-Plus obtains the correct total number of segments, but its boundaries are often misplaced. UVD has higher recall than Qwen3.5-Omni-Plus but substantially over-segments the trajectories. These results show that the total segment count alone is not sufficient; accurate disassembly also requires boundaries that follow the actual interaction stages.

Figure 6 gives a qualitative comparison on two multi-stage tasks. UVD merges the drawer-opening and bowl-grasping interactions in task (a) and over-segments a continuous movement in task (b). Direct VLM segmentation produces imprecise transitions in task (a) and stops before the complete horizon in task (b). PrimitiveVLA produces segments that follow the interaction stages in both trajectories.

We next evaluate how the boundary segmentation affects downstream policy learning. For each of the 18 tasks listed in Table 12, we use the same VLM-generated primitive sequence for UVD, Qwen3.5-Omni-Plus, and PrimitiveVLA, while using the temporal boundaries produced by each method. We then fine-tune a separate OpenVLA policy on the segmented data from each method, using the same training settings and task split.

Since this subset contains fewer tasks than the full Libero-90 setting, we use 50,000 optimization steps, matching the small-scale training setting in the main paper. All other training and evaluation settings remain unchanged. The boundary locations produced by the alternative methods may not

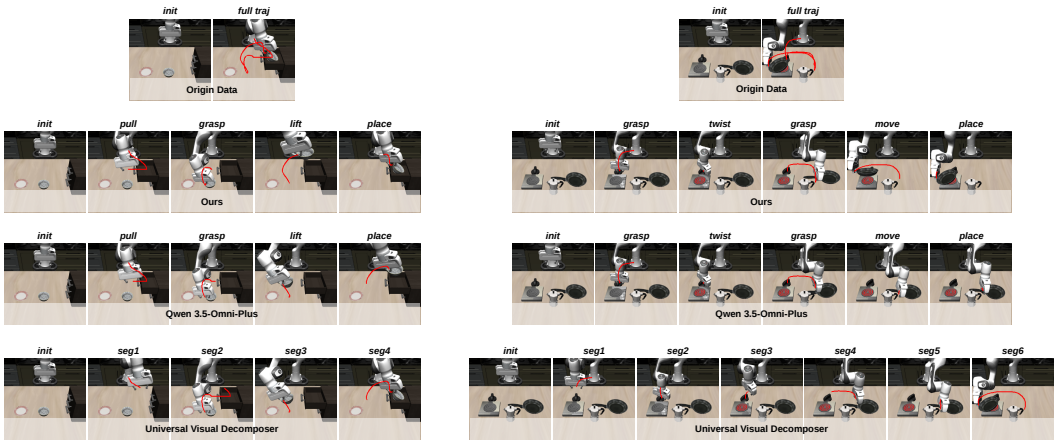


Figure 6: Qualitative comparison of primitive disassembly on Libero-90. The tasks are (a) *open the top drawer of the cabinet and put the bowl in it* and (b) *turn on the stove and put the frying pan on it*.

Table 14: Downstream OpenVLA success rates (%) using the same VLM-generated primitive sequences with temporal boundaries produced by different methods.

Segmentation method	VLA-predicted switching	Code-controlled switching
UVD	63.78	60.33
Qwen3.5-Omni-Plus	62.33	59.00
PrimitiveVLA (ours)	**66.78**	**67.44**

coincide with the completion signal used by the switching program. We therefore evaluate each policy with the two control modes defined in Sec. C.1.3: VLA-predicted switching and code-controlled switching based on the completion signal from the state/action program. This accounts for possible mismatches between the predicted boundaries and the switching criterion.

Table 14 shows, under VLA-predicted switching, UVD, Qwen3.5-Omni-Plus, and PrimitiveVLA achieve success rates of 63.78%, 62.33%, and 66.78%, respectively. Under code-controlled switching, the corresponding results are 60.33%, 59.00%, and 67.44%. PrimitiveVLA achieves the highest success rate under both control modes, improving over UVD by 3.00 and 7.11 percentage points, respectively. Together with the boundary-localization results, these findings indicate that more accurate temporal disassembly produces more useful primitive-level training samples for downstream policy learning.

C.1.2 Unified Multimodal Representation Alternatives

For the representation ablation, the policy must identify both what motion to perform and where to perform it. Unified multimodal representation separates these roles: unified primitive language describes a reusable motion without relying on instance-specific names, while visual grounding identifies the relevant object or target region. We compare instruction and visual representations while keeping the primitive segmentation and execution protocol fixed.

All variants use the same underlying demonstration data, backbone, and training setting. The informative-instruction + mask and Full UMR variants differ only in language unification, with segmentation, visual grounding, and execution control held fixed. Both variants preserve the same execution-relevant attributes when present. The mask-without-decomposition variant uses the full-task instruction and masks task-relevant objects, without primitive-level completion checks or switching. All variants retain RGB observations; "None" denotes the absence of additional visual markers.

For endpoint grounding, a predictor takes an informative primitive instruction and the current RGB observation as input and predicts the image-plane coordinates (u, v) of the end-effector position

Table 15: Instruction and visual representation ablation with the OpenVLA backbone. ID, Novel, and Long denote Libero-90, Libero-90-Novel, and Libero-Long, respectively.

Variant	Instruction	Visual	ID	Novel	Long
Unified primitive instructions only	Unified primitive	None	47.20%	12.00%	8.25%
Endpoint grounding	Unified primitive	Endpoint marker	63.10%	33.25%	8.25%
Informative primitive instructions only	Informative primitive	None	73.90%	21.50%	28.25%
Informative primitive instructions + mask	Informative primitive	Object mask	77.32%	40.00%	31.25%
Mask without decomposition	Full-task	Object mask	79.60%	33.75%	4.25%
Full UMR	Unified primitive	Object mask	**79.88%**	**45.50%**	**38.50%**

Table 16: Primitive-switching alternatives with unified primitive instructions and object masks. Numbers in parentheses indicate absolute percentage-point improvements over VLA-predicted switching.

Backbone	Switching	Libero-90	90-Novel	Long
OpenVLA	VLA-predicted	78.35%	40.00%	36.25%
OpenVLA	Code-controlled	**79.88%** (+1.53)	**45.50%** (+5.50)	**38.50%** (+2.25)
OpenVLA-OFT	VLA-predicted	45.11%	43.75%	0.00%
OpenVLA-OFT	Code-controlled	**94.70%** (+49.59)	**71.00%** (+27.25)	**66.50%** (+66.50)
$\pi_{0.5}$	VLA-predicted	95.93%	69.25%	79.00%
$\pi_{0.5}$	Code-controlled	**96.19%** (+0.26)	**75.50%** (+6.25)	**80.25%** (+1.25)

at primitive completion. Ground-truth endpoints are obtained by projecting the segment-end end-effector position into the image using the camera parameters. The predictor is trained on the same demonstration dataset used for VLA fine-tuning. During VLA training, ground-truth endpoint markers are overlaid on the RGB observations. At inference time, the endpoint is re-estimated before each action chunk and overlaid on the current RGB image. The VLA retains unified primitive instructions, with segmentation and switching unchanged from Full UMR.

As illustrated in Table 15, without additional visual markers, informative primitive instructions achieve 73.90%, compared with 47.20% for unified primitive instructions. Adding object masks raises these results to 77.32% and 79.88%, respectively, demonstrating the benefit of explicit instance grounding. With object masks, Full UMR improves over informative primitive instructions by 2.56, 5.50, and 7.25 percentage points on Libero-90, 90-Novel, and Long, respectively. Endpoint grounding achieves 63.10% on Libero-90, compared with 79.88% for Full UMR. The endpoint marker specifies a single location, while the object mask delineates the relevant object or target region. Mask grounding with full-task instructions reaches 79.60% on Libero-90 but only 33.75% and 4.25% on 90-Novel and Long. These results support the benefit of primitive decomposition and execution control beyond object masking alone.

Overall, UMR combines reusable primitive language with explicit visual grounding, while primitive decomposition supports their use in multi-stage task execution and compositional generalization.

C.1.3 Primitive Switching Alternatives

A correct primitive sequence also requires reliable transitions between primitives. The controller should switch only after the current primitive has been completed; an early switch interrupts the current interaction, while a late switch can repeat or extend it. We therefore compare VLA-predicted switching with code-controlled switching based on the completion signal from the state/action program. Both methods use the same primitive flow, representation, segmentation, backbone, and training protocol. The comparison is performed across OpenVLA, OpenVLA-OFT, and $\pi_{0.5}$.

For VLA-predicted switching, we assign a binary change label of 1 to the final two frames of each primitive segment and 0 to all preceding frames. This label is appended to the action vector and trained jointly using the same action prediction loss as the corresponding backbone. At inference

Table 17: Further zero-shot compositional generalization evaluation. Scene abbreviations follow the main paper. Values are success rates (%), and group means are arithmetic means over tasks.

Setting	Scene	Task	OpenVLA	PrimitiveVLA (ours)
Novel object				
	Kit-1	Ramekin on the plate	18.00	**62.00**
	Liv-4	BBQ sauce in the tray	36.00	**46.00**
	Liv-4	Cookie in the tray	0.00	**88.00**
		Mean	18.00	**65.33**
Novel location				
	Study-3	Book on the right	18.00	**78.00**
	Kit-10	Black bowl in the middle drawer	74.00	**80.00**
	Liv-5	White mug on the left plate with swapped initial positions	0.00	**10.00**
	Liv-5	Yellow-and-white mug on the right plate with swapped initial positions	0.00	**12.00**
	Kit-10	Close the middle drawer	2.00	**4.00**
		Mean	18.80	**36.80**
Novel pair				
	Kit-2	Stack the front black bowl on the back black bowl	2.00	**76.00**
	Kit-10	Put the black bowl on top of the cabinet	20.00	**94.00**
	Liv-4	Pick up the right black bowl and put it in the tray	0.00	**78.00**
	Liv-5	Put the yellow-and-white mug on the left plate	0.00	**22.00**
	Liv-2	Pick up the butter and put it in the basket	40.00	**70.00**
		Mean	12.40	**68.00**
Novel structure				
	Kit-2	Stack the front black bowl on the middle black bowl, then place both on the cabinet	0.00	**30.00**
	Kit-1	Open the top drawer, put the bowl inside, and close it	10.00	**20.00**
	Kit-10	Open the top drawer, put the chocolate pudding inside, and close it	0.00	0.00
		Mean	3.33	**16.67**
		All 16 tasks	13.75	**48.13**

time, predicted change values are thresholded at 0.5 and stored in a queue. Switching requires two consecutive positive predictions to suppress isolated signals. The switching condition is checked before each action is executed. When a switch is triggered, the remaining actions in the current action chunk are discarded, the queue is cleared, and execution proceeds to the next primitive.

Table 16 shows that code-controlled switching performs better across all three backbones and all evaluation settings, with the largest gains for OpenVLA-OFT. One possible explanation is that its MLP action head has difficulty learning the sparse change signal, although this experiment does not isolate that mechanism. The gains are also consistent across backbones on 90-Novel and Long, suggesting that explicit completion signals help reduce transition errors in unseen and long-horizon task compositions.

Overall, code-controlled switching improves the reported results across all three backbones and provides more consistent primitive transitions under the evaluated settings.

C.2 Further Evaluation of Compositional Generalization

The main paper evaluates compositional generalization on Libero-90-Novel using an overall success rate. To examine the different sources of this generalization, we organize the evaluation into four settings and extend it to 16 tasks. In the *novel-object* setting, the interaction and object location remain the same while the object changes. In the *novel-location* setting, the object and interaction remain the same while the initial or target location changes. In the *novel-pair* setting, the object and target location are familiar separately, but their combination is absent from the 83-task fine-tuning subset. The butter–basket task is listed in the complete LIBERO-90 inventory but is excluded from the fine-tuning subset because its demonstrations are unavailable. In the *novel-structure* setting, familiar objects and locations are used in a new primitive sequence or a new multi-stage task. The 16 tasks include the eight tasks from the main paper and eight additional tasks. Table 17 lists the four settings, the corresponding scenes, and the tasks used in each setting.

We use the same OpenVLA backbone and Libero-90 training data as in the main experiments. PrimitiveVLA uses the primitive instructions, object masks, and switching procedure described in

the paper. All tasks are evaluated zero-shot, without demonstrations from the test tasks or additional task-specific fine-tuning. Each simulation task is evaluated over 50 trials.

As shown in Table 17, PrimitiveVLA achieves a higher mean success rate in all four settings. The success rate increases from 18.0% to 65.3% for novel objects, from 18.8% to 36.8% for novel locations, from 12.4% to 68.0% for novel pairs, and from 3.3% to 16.7% for novel structures. Averaged over all 16 tasks, the success rate increases from 13.8% to 48.1%, a gain of 34.4 percentage points. PrimitiveVLA improves 15 of the 16 tasks, and the remaining task has zero success for both methods. The largest gain is observed for novel object–location pairs, while novel structures remain the most difficult setting. These results show that **PrimitiveVLA helps the model reuse learned motions when objects, locations, and task compositions change.**

D Additional Discussions

D.1 Cross-Task Action-Pattern Analysis

Primitive Sequence Reasoning (Sec. 4.2.1) and State-Based Boundary Segmentation (Sec. 4.2.2) identify the primitive sequence and temporal boundaries, respectively; this disassembly step produces primitive-aligned segments, but it does not by itself establish that instances of the same primitive share a common low-level motion pattern across tasks. We therefore analyze the original action trajectories and compare same-primitive segments with different-primitive segments across tasks using the Libero-90 training set, which contains 3,688 original trajectories and 10,546 resulting primitive segments (Table 11). Each primitive segment contains the end-effector trajectory and two gripper coordinates. We translate each trajectory to its initial position, apply the project-defined yaw normalization, and resample the end-effector and gripper trajectories to 72 points along the motion path. The distance between two primitive segments is defined as

$$D_{ij} = \frac{1}{72} \sum_{t=1}^{72} \sqrt{\left\| \frac{\Delta x_{ijt}}{0.1\,\mathrm{m}} \right\|_2^2 + \left(\frac{\theta_{ijt}}{1\,\mathrm{rad}} \right)^2 + \left\| \frac{\Delta g_{ijt}}{0.1\,\mathrm{m}} \right\|_2^2}, \tag{10}$$

where θ_{ijt} is the relative end-effector rotation angle and g contains the two gripper values. This distance measures the difference in translation, rotation, and gripper motion between two primitive segments.

Using the same trajectory distance, we calculate the silhouette score and the mean within-group and between-group distances on the same primitive segments, first grouped by task and then by primitive type. The silhouette score measures both within-group compactness and between-group separation, with a higher value indicating that the grouping better matches the underlying motion patterns. The mean within-group distance measures motion similarity inside each group, while the mean between-group distance measures the separation between groups. We further calculate the same-primitive cross-task distance between instances of the same primitive from different tasks, the different-primitive cross-task distance between different primitives from different tasks, and the within-task primitive silhouette among primitive types in each task.

Table 18 shows, compared with grouping by task, grouping by primitive type produces a higher silhouette score (0.1479 vs. -0.1906), a lower mean within-group distance (0.8806 vs. 1.3765), and a higher mean between-group distance (1.7646 vs. 1.5750). Together, these results show that segments of the same primitive are more compact and segments from different primitive types are more clearly separated than when the same segments are grouped by task. The same-primitive cross-task distance is 0.8895, which is 49.57% lower than the different-primitive cross-task distance of 1.7638, showing that similar motion patterns recur across tasks. The within-task primitive silhouette is 0.6621, further showing that primitive motion patterns remain clearly separated within each task.

Overall, the raw demonstrations exhibit recurring cross-task motion structure under the specified segmentation procedure and trajectory-distance metric. Appendix D.2 separately evaluates whether the trained policy can execute the corresponding primitives in new task contexts.

D.2 Video-Verified Primitive Transfer Analysis

To examine whether learned primitive behaviors transfer across task contexts, we use the OpenVLA + Full UMR checkpoint from the main experiments, fine-tuned on the 83-task `LIBERO-90` subset.

Table 18: Action-pattern statistics for primitive segments grouped by task or primitive type.

Metric	Grouped by Task	Grouped by Primitive Type
Silhouette ↑	−0.1906	0.1479
Mean within-group distance ↓	1.3765	0.8806
Mean between-group distance ↑	1.5750	1.7646

Table 19: The eight ID tasks used in the matched primitive-transfer analysis. All tasks belong to the 83-task fine-tuning subset.

ID	Task instruction
1	LIVING_ROOM_SCENE4_pick_up_the_black_bowl_on_the_left_and_put_it_in_the_tray
2	KITCHEN_SCENE10_close_the_top_drawer_of_the_cabinet
3	KITCHEN_SCENE10_put_the_black_bowl_in_the_top_drawer_of_the_cabinet
4	KITCHEN_SCENE10_put_the_chocolate_pudding_in_the_top_drawer_of_the_cabinet_and_close_it
5	KITCHEN_SCENE1_open_the_top_drawer_of_the_cabinet_and_put_the_bowl_in_it
6	KITCHEN_SCENE2_stack_the_black_bowl_at_the_front_on_the_black_bowl_in_the_middle
7	LIVING_ROOM_SCENE3_pick_up_the_butter_and_put_it_in_the_tray
8	LIVING_ROOM_SCENE5_put_the_yellow_and_white_mug_on_the_right_plate

We evaluate the eight ID tasks listed in Table 19 and the same eight-task `Libero-90-Novel` suite used in the main experiments (Table 9), using the first 20 recorded episodes for each task without selection. The ID tasks were selected from the fine-tuning subset to provide comparable task contexts for this analysis, with similar scene configurations and primitive sequences to those in the Novel suite. The same primitive representation and execution procedure are used in both settings. The Novel tasks are evaluated zero-shot, without demonstrations from the test tasks or additional task-specific fine-tuning. Two authors manually verify the recorded rollouts, and the resulting labels are used only for this post-hoc analysis.

Primitive intervals are identified from the execution log retained during testing. At each step, the active primitive is the primitive selected by the switching controller, and consecutive records with the same active primitive define one primitive interval. The log determines the temporal scope of each attempt, while video verification determines whether the intended object or target was manipulated and whether the primitive reached its physical completion condition. A primitive interval is considered valid only when all preceding primitives in the execution flow have been manually verified as correctly completed. If a preceding primitive interacts with the wrong object or target, produces an empty grasp, or otherwise fails its physical completion condition, all subsequent intervals in that episode are marked as not reached and excluded from the corresponding primitive-level denominator. If the current primitive is reached but does not correctly manipulate the intended object or target, it is counted as a failed attempt.

This validity rule prevents later primitive intervals from being counted once an episode has deviated from the intended task flow. In an OOD evaluation, a preceding error may cause the policy to execute an interaction that resembles a familiar training task or object configuration. Although this interaction may appear successful in isolation, counting it could overestimate the completion rate of the intended primitive. We therefore include a primitive in the transfer analysis only if it is reached through a correctly completed intended prefix.

For each valid execution, we use y to denote a video-verified success and n to denote a failure. The conditional completion rate is then computed as $y/(y+n)$, where $y+n$ is the number of valid attempts for that primitive. Episodes that fail before reaching a primitive remain task-level failures, even though they are excluded from the denominator of subsequent primitives.

As shown in Table 20, the matched ID tasks yield high conditional completion rates for all six primitive types. On `Libero-90-Novel`, *grasp*, *lift*, and *place* remain highly reliable across changed task contexts, whereas *move*, *pull*, and *push* show larger decreases. This difference indicates that interaction-specific primitives are more sensitive to changed geometry, object configuration, and spatial relations.

The strict completion rate of *push* on the Novel tasks is low ($5/31 = 16.1\%$), but video inspection shows that many failed attempts still contain the intended closing action and partial drawer motion.

Table 20: Video-verified conditional primitive-completion rates on the first 20 episodes of each matched ID and `Libero-90-Novel` task. The OpenVLA + Full UMR checkpoint from the main experiments is used in both settings. Each y/n entry reports successful and failed valid executions, respectively; the rate is computed as $y/(y+n)$.

Primitive	**ID**		**Novel**	
	y/n	**Rate**	y/n	**Rate**
Grasp	131/6	95.6%	105/14	88.2%
Lift	54/0	100.0%	30/1	96.8%
Move	77/0	100.0%	53/20	72.6%
Place	126/5	96.2%	82/1	98.8%
Pull	17/3	85.0%	19/21	47.5%
Push	38/0	100.0%	5/26	16.1%

In particular, failures often occur when the middle drawer cannot be fully closed because the handles of adjacent drawers obstruct its motion. We retain the strict endpoint criterion for the reported completion rate, while distinguishing failures to reach the intended endpoint from failures to invoke the push behavior.

These behavioral results complement the cross-task action-pattern analysis in Appendix D.1, which shows that training demonstrations exhibit more coherent motion structure when grouped by primitive type than by task. The video-verified results extend this data-level observation by showing that the trained policy can execute several of these primitive behaviors in new task contexts.

This interpretation is also consistent with the task-level Novel results, where PrimitiveVLA improves OpenVLA from 9.25% to 45.50% and also improves both OpenVLA-OFT and $\pi_{0.5}$. The failure analysis shows a corresponding shift from imitative interference and frozen behaviors toward physical execution and primitive-transition errors, consistent with improved behavior selection and initiation while highlighting the remaining execution challenges.

Together, these analyses support cross-task reuse of several learned primitive behaviors, while the lower conditional completion rates for *move*, *pull*, and *push* highlight remaining challenges under changed geometry and spatial relations.

A transition-level analysis of online switching timing on these trajectories is provided in Appendix D.3.

D.3 Transition-Level Switching Analysis

We evaluate the temporal reliability of PrimitiveVLA's switching controller on the same evaluation trajectories used in Appendix D.2. The evaluation includes the eight ID tasks listed in Table 19 and the same eight-task Libero-90-Novel suite used in the main experiments. For each task, we use the first 20 recorded episodes without selection. The same primitive representation and execution procedure are used in both settings.

Each expected primitive transition is counted as one evaluation instance, so the reported N values refer to transition instances rather than complete episodes. Because the selected tasks contain different primitive compositions, the number of instances varies across transition types. Two authors manually verify the recorded rollouts, and the resulting labels are used only for this post-hoc switching analysis.

For each reachable transition, the execution flow specifies the ordered primitive transition and its completion condition. We manually inspect the recorded test video and identify the completion boundary τ of the current primitive. Because τ depends on the realized execution, its exact frame-level timing cannot be fixed independently before rollout. A switching event within $[\tau-7, \tau+7]$ frames is considered timely and labeled *Correct*. A switch before this window is labeled *Premature*, and a switch after this window is labeled *Late*. If the current primitive does not complete, any emitted switch is treated as *Premature*; if no switch is emitted, we record an *Execution Failure*. If the predecessor primitive fails and the current primitive is never reached, we record an *Unreachable* transition.

Table 21: **Aggregate transition-level switching analysis on ID and Novel evaluation trajectories.** The N column counts all expected transition instances, while Observed denotes Correct + Premature + Late. Correct, Premature, and Late are mapped to TP, FP, and FN under the windowed switching protocol. Switching Accuracy is computed over observed switching events, and Valid Transition Rate uses all expected transition instances.

Split	N	Correct	Premature	Late	Exec. Failure	Unreachable	Observed	Switching Accuracy	Valid Rate	Windowed Switching F1
ID	320	288	7	5	3	17	300	96.00%	90.00%	97.96%
Novel	360	230	7	6	29	88	243	94.65%	63.89%	97.25%

The switching evaluation uses a one-decision, windowed protocol for each reachable transition. Unlike the segmentation-boundary evaluation in Appendix C.1.1, which matches predicted boundaries against a fixed set of manually annotated reference boundaries, this analysis compares the observed switching event with the completion boundary determined from the realized rollout. The completion boundary defines one expected switching opportunity, namely emitting a switch within the tolerance window.

Under this convention, a switch within the window is a true positive (TP) because the controller emits the required switching decision at the expected time. A switch before the window is a false positive (FP) because the controller advances while the current primitive is still expected to continue. A switch only after the window is a false negative (FN) because the controller fails to make the required timely decision, even though it eventually switches. The late switch is counted only as an FN rather than as an additional FP, because each transition contributes one timing decision and the metric evaluates whether the timely switching opportunity was met, rather than whether any switch eventually occurred. Execution Failure and Unreachable transitions are reported separately because they do not constitute valid switching decisions for the current transition.

Let n_{c}, n_{p}, and n_{l} denote the numbers of *Correct*, *Premature*, and *Late* transitions, respectively. We compute the windowed switching precision, recall, and F1 as

$$P_{\mathrm{w}} = \frac{n_{\mathrm{c}}}{n_{\mathrm{c}} + n_{\mathrm{p}}}, \qquad R_{\mathrm{w}} = \frac{n_{\mathrm{c}}}{n_{\mathrm{c}} + n_{\mathrm{l}}}, \qquad F1_{\mathrm{w}} = \frac{2P_{\mathrm{w}}R_{\mathrm{w}}}{P_{\mathrm{w}} + R_{\mathrm{w}}}. \tag{11}$$

We also report the proportion of timely switches among classified switching events,

$$\text{Switching Accuracy} = \frac{\text{Correct}}{\text{Correct} + \text{Premature} + \text{Late}}, \tag{12}$$

and the proportion of timely transitions over all expected transitions,

$$\text{Valid Transition Rate} = \frac{\text{Correct}}{N}. \tag{13}$$

The first quantity measures switching timing conditional on a classified switching event, whereas the second includes execution failures and unreachable transitions. Thus, $F1_{\mathrm{w}}$ is a windowed switching score defined by the observed transition decisions, while Appendix C.1.1 reports a conventional segmentation-boundary F1 based on matching predicted boundaries to a complete set of manually annotated boundaries.

As shown in Table 21, PrimitiveVLA achieves 97.96% windowed switching F1 on ID transitions and 97.25% on Novel transitions. The switching accuracy among observed events is 96.00% for ID and 94.65% for Novel, indicating that premature and late switches are relatively infrequent. The valid transition rate over all expected transitions is 90.00% for ID and 63.89% for Novel. The difference mainly reflects primitive execution failures and unreachable transitions: Novel evaluation contains 29 execution failures and 88 unreachable instances, compared with 3 and 17 instances for ID. Thus, switching timing remains stable for observed transitions, while the lower valid-transition rate on Novel tasks is mainly caused by failures in primitive execution and their propagation to later transitions.

D.4 VLM/LLM Errors and Hallucination Mitigation

PrimitiveVLA constrains generative models at both stages where they are used. For execution-flow generation, the VLM selects only from the common fixed primitive set $\mathcal{C}$ and receives the top-3 related

instruction–sequence pairs from the Disassembly Library. These constraints reduce unsupported primitive choices and implausible ordering. For segmentation and switching, the LLM receives explicit primitive definitions, state/action formats, and reference segmentation logic. It generates each program once before dataset segmentation or task execution, after which the rollout executes fixed code without any VLM or LLM generation in the real-time control loop.

Residual errors remain possible, yet the complete system retains strong ID performance and improves the reported OOD baselines. The disassembly and grounding pipeline can produce two observable artifacts: primitive merging, in which adjacent motion phases are not cleanly separated, and mask drift, in which the mask is assigned to the wrong object or target. We manually screened a set of 830 original Libero training trajectories (10 per task across the 83 fine-tuning tasks) and 100 RLBench training trajectories (10 per task across the 10 tasks). Counting each trajectory once when either artifact was observed, we found artifacts in 24 Libero trajectories (2.9%) and 8 RLBench trajectories (8.0%). Generated physical-condition checks mitigate some primitive-merging cases. Mask errors mainly involved incorrect object or target assignment. These counts characterize artifact frequency in the screened training annotations and do not isolate their causal effect on policy performance. OOD tasks with novel objects may be more sensitive to grounding errors; nevertheless, the complete UMR system improves the corresponding reported OOD baselines.

D.5 Additional RLBench Analysis

The task-level RLBench results for T1–T10 are reported in Table 4. PrimitiveVLA improves the mean success rate for all three backbones. For $\pi_{0.5}$, the mean increases from 29.50% to 37.50%, corresponding to an absolute gain of 8.00 percentage points. For OpenVLA, it increases from 26.00% to 39.00%, a gain of 13.00 percentage points. For OpenVLA-OFT, it increases from 3.00% to 4.50%, corresponding to a 1.50 percentage-point gain and a 50.0% relative improvement.

Despite this relative improvement, the absolute success rate of OpenVLA-OFT remains low. This suggests that the backbone has difficulty handling the broader task, object-layout, and interaction variations in RLBench compared with LIBERO. PrimitiveVLA provides a measurable improvement in this more diverse evaluation setting, although the remaining low success rate indicates that the backbone still struggles with these variations.

D.6 Generalization Analysis

PrimitiveVLA supports compositional generalization by expressing unseen task compositions as sequences of familiar primitives.

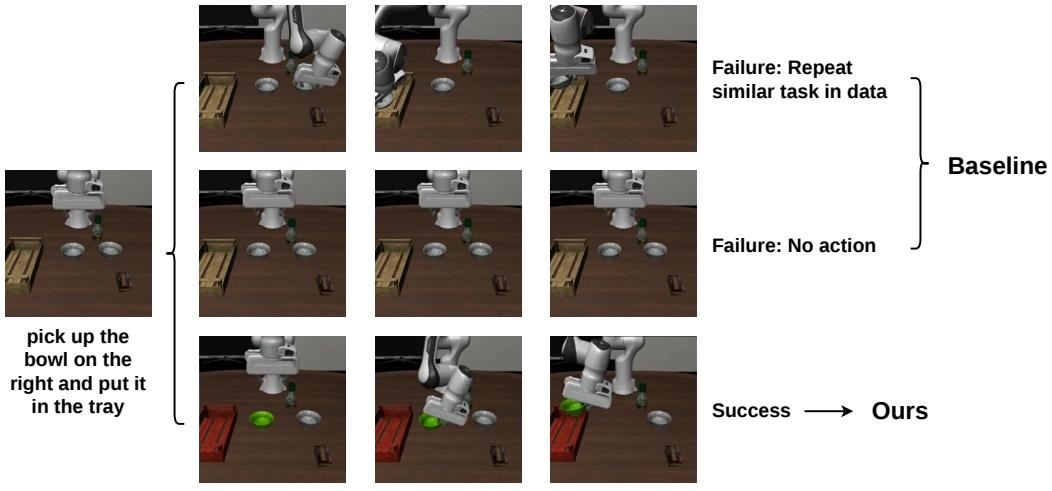
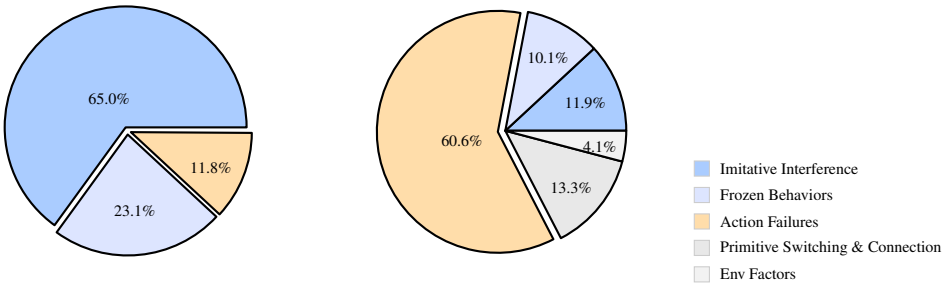


Figure 7: **Examples of generalization performance on an OOD task**: *"pick up the bowl on the right and put it in the tray"*.

As shown in Figure 7, standard VLA models often struggle with unseen task compositions due to two critical failure modes: (1) **Imitative interference**, where the model mistakenly repeats a similar but incorrect task from the training data. For example, when given an OOD task like *"pick up the bowl on the right and put it in the tray"*, a baseline VLA tends to execute a previously learned task, such as *"pick up the bowl on the left and put it in the tray"*, failing to adapt to the new instruction. (2) **Task-level ambiguity**, where complex, unseen instructions lead to *"frozen" behaviors* (the robot fails to act entirely);

PrimitiveVLA addresses these challenges by decomposing unseen task compositions into sequences of known, reusable interactions. Through object-mask grounding and unified primitive instructions, the OOD task is expressed as a series of primitives already present in the training data—such as *"grasp object with green mask"* or *"place in object with red mask"*. This transformation allows the model to execute unseen task compositions by combining in-distribution motion patterns.



(a) OpenVLA Baseline (b) PrimitiveVLA (Ours)

Figure 8: Failure mode distribution on Libero-90-Novel tasks. The results show a structural migration of errors from high-level semantic ambiguity to physical interaction challenges.

Shift in Error Patterns. We conducted a granular failure analysis on Libero-90-Novel using OpenVLA as the backbone, comparing standard fine-tuning with OpenVLA + PrimitiveVLA (Figure 8). The analysis includes 363 failed episodes for OpenVLA and 218 for OpenVLA + PrimitiveVLA, out of 400 evaluation episodes per method. The percentages below are computed among the failed episodes of each method.

- **Baseline: Dominance of Semantic Failures.** In the baseline configuration, Imitative Interference accounts for 65.0% of failures, followed by Frozen Behaviors at 23.1%. Together, these two categories constitute over 88% of baseline failures, suggesting that many failures involve selecting or initiating an inappropriate behavior under unfamiliar instructions or environments.
- **PrimitiveVLA (ours): Transition to Execution-Centric Challenges.** With PrimitiveVLA, the error landscape shifts significantly toward low-level execution.
  - **Significant Reductions:** The proportion of Imitative Interference drops sharply to 11.9%, and Frozen Behaviors decrease to 10.1%. This pattern is consistent with the view that task disassembly and object-mask grounding provide a clearer goal and execution flow, reducing some task-level ambiguity.
  - **Dominance of Action Failures:** Action Failures now become the primary error mode, accounting for 60.6% of PrimitiveVLA failures. This shift is consistent with the model attempting OOD tasks instead of defaulting to similar ID tasks or remaining "frozen." The resulting failures, often observed as trajectory deformations, reflect the difficulty of transferring learned motion patterns to OOD physical distributions, including unseen objects, novel environmental states, and unfamiliar spatial arrangements.
  - **Emerging Paradigm-Specific Errors:** New error types unique to our framework, including Primitive Switching (6.4%) and Motion Connection (6.9%), account for a combined 13.3%. These errors highlight the remaining gaps in the generated switching logic and the limited coverage of unseen primitive sequences in the current training data. Environment Factors account for the remaining 4.1% of failures.

D.7 Data efficiency analysis

To investigate why PrimitiveVLA maintains high success rates even under a half-scale (50% data) regime, we analyze the distribution of 11 identified *primitives* across the Libero benchmarks. The detailed statistics are summarized in Table 22.

Table 22: Primitive occurrences across the complete LIBERO task suites.

Primitive	Libero-90	Libero-Goal	Libero-Spatial	Libero-Object	Total
Grasp	86	8	10	10	114
Place	63	7	10	10	90
Move	49	3	10	10	72
Lift	28	4	0	0	32
Insert	12	0	0	0	12
Pull	5	2	0	0	7
Push	8	1	0	0	9
Press	0	1	0	0	1
Twist	5	1	0	0	6
Tilt	6	0	0	0	6
Rotate	2	0	0	0	2
Total Tasks	**90**	**10**	**10**	**10**	**120**

Note. Counts indicate primitive occurrences in task-level decompositions; repeated occurrences within a task are counted separately. The LIBERO-90 column covers all 90 tasks, including those excluded from fine-tuning.

Even when each benchmark is fine-tuned independently, the results are consistent with three factors that can improve data efficiency:

- **Primitive-aligned training pools supervision for recurring primitive categories.** In diverse suites such as *Libero-90*, task-level training associates each instruction with a complete trajectory, leaving supervision for recurring motion patterns distributed across different task contexts. PrimitiveVLA makes this shared structure explicit by organizing segments under common primitive categories. For instance, while there are 90 unique task descriptions in the complete LIBERO-90 suite, *grasp* occurs 86 times across their primitive sequences, illustrating how the same primitive recurs across different tasks. By pooling retained segments under shared primitive categories, demonstrations from different tasks contribute supervision to the same primitive, supporting data-efficient learning even with 50% fewer demonstration trajectories.
- We reduce the learning burden by aligning diverse tasks into a consistent structural format. In the *Libero-Spatial* and *Libero-Object* benchmarks, the primary challenge lies in environmental variations (e.g., object positions or categories) rather than logical complexity. PrimitiveVLA explicitly simplifies all 20 tasks in these suites into a consistent `[grasp, move, place]` sequence. This structural alignment shifts the learning burden from understanding multi-stage linguistic logic to mastering short-range primitives. Such simplification enables the model to reach better performance with significantly less data than baselines.
- **By isolating short primitives into independent training trajectories, PrimitiveVLA increases their relative sampling opportunity.** When action windows are sampled from complete task trajectories, short actions such as *lift* and *press* occupy fewer frames and are therefore less likely to be selected than longer motion phases. After segmentation, each retained primitive segment is sampled as an independent training trajectory, so its selection probability no longer depends directly on its duration. This gives short-duration primitives more direct exposure while preserving the backbone-specific sampling and action-chunk rules described in Appendix B.2.

D.8 Latency and Computational Overhead

PrimitiveVLA adds a modest computational overhead to the base VLA models during execution.

Primitive execution flow generation and LLM-based switching program generation occur once during pre-execution initialization and therefore add no per-step VLM/LLM calls during the rollout. The Python switching logic is lightweight during real-time execution.

Tracking Latency. We utilize Cutie for real-time mask tracking. Its latency scales with image resolution, reaching approximately 9 ms per frame at 256×256 and 30 ms per frame at 768×768. To maintain accurate grounding, we use 768×768 resolution and update the mask before each action-chunk generation. We report per-step latency by measuring the total wall-clock time for each action chunk, including mask tracking, VLA inference, and switching logic, and dividing it by the corresponding chunk size.

Table 23: **Inference latency comparison.** Reported values are average per-step latencies (ms), computed by dividing the wall-clock time of each action chunk by its chunk size. PrimitiveVLA latency includes mask tracking, VLA inference, and switching logic, whereas baseline latency measures the corresponding backbone action inference.

Model	**Chunk Size**	**Baseline Latency**	**PrimitiveVLA Latency**
OpenVLA	1	$\sim$500 ms	$\sim$540 ms
OpenVLA-OFT	5	$\sim$88 ms	$\sim$96 ms
$\pi_{0.5}$	10	$\sim$67 ms	$\sim$72 ms

As shown in Table 23, the proposed components increase the average per-step latency by approximately 8.0%, 9.1%, and 7.5% in the three reported settings, respectively. In our current experiments, the action chunk size is relatively small, so updating the mask once per chunk is sufficient. For future VLAs with substantially larger chunk sizes (e.g., > 20), asynchronous mask updates or updates at fixed step intervals could be used to maintain a similar execution frequency.

E Visual Results

We provide additional videos in the supplementary material to demonstrate PrimitiveVLA across various test scenarios.

F Existing Assets

Table 24: Details of existing assets used in PrimitiveVLA

Asset	Category	License	Original Source / URL
OpenVLA	Base VLA Model	MIT	https://github.com/openvla/openvla
OpenVLA-OFT	Base VLA Model	MIT	https://github.com/moojink/openvla-oft
$\pi_{0.5}$	Base VLA Model	Apache 2.0	https://github.com/physical-intelligence/openpi
Qwen3-VL	VLM	Qwen License	https://github.com/QwenLM/Qwen3-VL
Qwen2.5-VL-72B-instruct	VLM	Qwen License	-
DeepSeek-V3	LLM	DeepSeek License	https://github.com/deepseek-ai/DeepSeek-V3
SAM	Perception	Apache License	https://github.com/facebookresearch/segment-anything
Cutie	Perception	MIT	https://github.com/hkchengrex/Cutie
Libero	Benchmark	MIT	https://github.com/Lifelong-Robot-Learning/LIBERO
RLBench	Benchmark	custom (Free for research)	https://github.com/stepjam/RLBench

G RELEASE PLAN

Upon acceptance, we will release the datasets, trained models, and code, subject to the licenses of the original assets.