AGENTIC WORLD ACTION MODEL

Anonymous authors
Paper under double-blind review

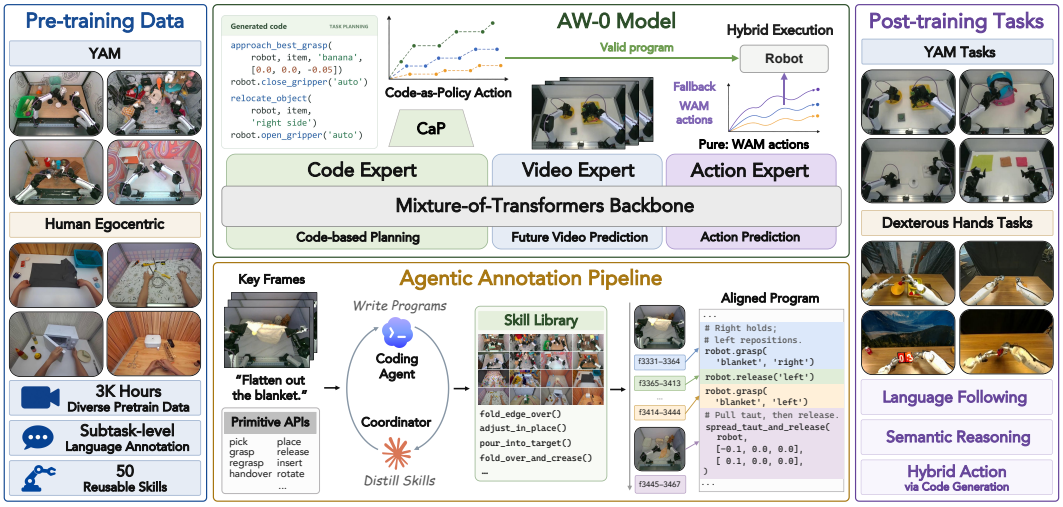


Figure 1: **Overview of AW-0.** AW-0 is an agentic world action model that unifies code-based planning with world action modeling by jointly denoising skill programs, future video, and control trajectories. An agentic pipeline discovers reusable skills and annotates robot and human demonstrations with programs for both pretraining and downstream fine-tuning. The resulting policy achieves strong performance on both parallel-gripper and dexterous-hand tasks.

ABSTRACT

Recent world action models provide visual foresight and future prediction for robot manipulation, but lack explicit mechanisms for compositional skill reuse. Agentic robotics combines code-based planning and reasoning with reusable skills, but often lacks predictive grounding in physical dynamics. We introduce **AW-0**, an agentic world action model framework that integrates code-based planning with joint video and action prediction. We jointly model code, future video, and robot actions as three coupled generative streams within a unified mixture-of-transformers architecture. Within this architecture a code expert generates parameterized skill routines and control logic while video and action experts generate future visual states and low-level motor commands. Code serves two complementary roles: it structures multi-step prediction and control within the world model and represents skills as reusable programs for subsequent agentic execution. We further introduce an automated curation pipeline that labels robot and human video trajectories with aligned programs and consolidates recurring routines into reusable skill libraries. Evaluations on RoboTwin 2.0 and physical bimanual manipulation tasks show improved task performance over VLA and WAM baselines, alongside gains in language following, semantic reasoning and robustness, demonstrating that agentic code enables connection between planning, prediction and execution within WAMs.

1 INTRODUCTION

Learning-based robotic manipulation has progressed from small task-specific visuomotor policies over a decade ago to broadly supervised foundation models that connect perception, language, and

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.

action (Levine et al., 2016; Zitkovich et al., 2023; Kim et al., 2024; Niu et al., 2024; Black et al., 2024; Niu et al., 2025). In parallel, self-supervised video prediction has long been valued as a learning signal to model the world: image inpainting extended to video prediction led to action-conditioned video predictors that can imagine the future before committing to an action (Pathak et al., 2016; He et al., 2021; Tong et al., 2022; Finn et al., 2016; Assran et al., 2025; Ye et al., 2026).

World Action Models (WAMs) couple visual prediction with visuomotor policies, grounding control in representations of how the physical world evolves (Ye et al., 2026; Kim et al., 2026; Yuan et al., 2026). This predictive grounding is particularly valuable for complex manipulation, where successful execution depends on anticipating object motion, contact, and the consequences of low-level actions. Yet predicted video and action chunks do not by themselves specify how a long-horizon task should be decomposed: which reusable skills to invoke, how to parameterize them, or how to compose them across subtasks. Predicting the world and/or actions in general can also be expensive, especially as it can be difficult to establish limits on how many degrees of freedom need to be "world modeled" (or teleoperated, in the supervised setting).

Agentic robot systems provide complementary structure to WAMs: a programmatic harness that selects reusable skills and tools, binds them to the current scene, tracks progress, and replans as observations change (Liang et al., 2023; Fu et al., 2026; Zhang et al., 2026; Xiao et al., 2026; Lu et al., 2026). This programmatic representation makes the structure of multi-step tasks explicit, specifying which skills and tools to invoke, with what arguments, and in what sequence. Yet agentic robotics often lacks predictive grounding in real-world physical dynamics and does not leverage a learned model of how the world will change as the robot acts.

We advocate for an architecture that combines the foresight and action execution of WAMs with the cognitive efficiency, reusable skills, and self-improvement capabilities of agentic robotics. We propose **AW-0**, an agentic WAM framework that incorporates code-based planning and generation directly into world action modeling via a grounded code expert that harnesses broad programming priors to synthesize modular skills featuring explicit parameters and control logic. Code guides synchronized video and control modules within a mixture-of-transformers architecture, simultaneously denoising discrete code structures, predicted visual observations, and low-level motor commands.

We first show that such agents can be realized by adding a code expert to a WAM, leveraging multi-sensor action and code generation in a planning stage. We develop a modular generation framework: planning via code generation significantly improves WAM performance, by appropriately conditioning the long-horizon generations required for complex robotic tasks. An automated curation pipeline extracts systematic routines into reusable program libraries, mapping code segments directly onto robotic and human video trajectories. This approach bridges high-level semantic reasoning with predictive physical dynamics, establishing a unified language for compositionality across distinct environments and tasks. We further demonstrate agentic skill reuse via a hybrid deployment strategy which selects between raw action chunk execution with programmatic skill invocation. Empirical evaluations across RoboTwin 2.0 benchmarks and physical bimanual setups show substantial performance gains compared to standard VLA and WAM baselines, highlighting programmatic code as an effective structural backbone for predictive planning, skill discovery, and embodied robotic control.

Our primary contribution is the conceptualization and realization of agentic robotic skills within a world action model framework, showing that code generation improves action prediction in the world model, and that the code generated from the world model can be re-used in an agentic framework. Specific novel contribution elements include: a mixture-of-transformers framework integrating a code expert into world action models via simultaneous denoising of code, video, and control; a automated agentic pipeline designed for program annotation and skill discovery; a dataset of such structured skill programs; and a hybrid execution policy that dynamically toggles between programmatic skill invocation and direct deployment of action chunks from the WAM action expert.

2 Agentic World Action Model

We formulate AW-0 as a unified conditional generative model over code, video, and actions, which respectively describe structured skill plans, future scene dynamics, and continuous robot motion. We represent a skill program as a discrete token sequence $C_0 = (c_1, \ldots, c_{N_c})$, future observations

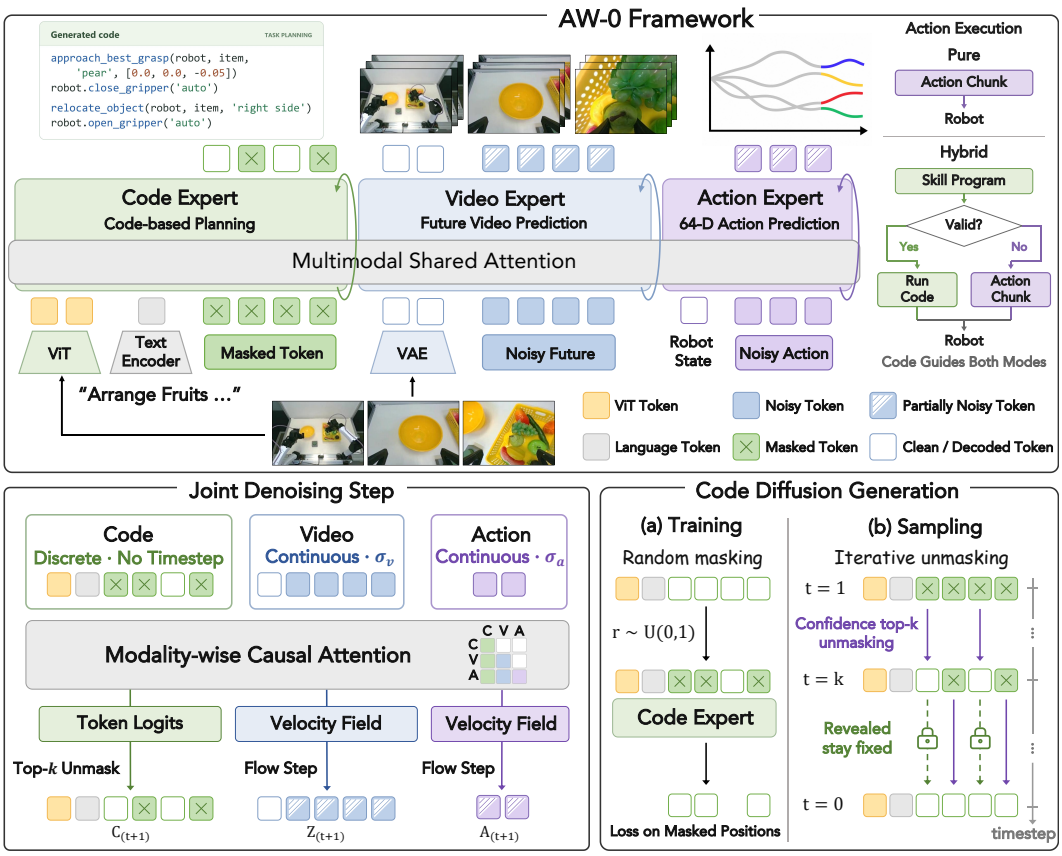


Figure 2: **AW-0 Framework.** AW-0 includes a Mixture-of-Transformers backbone with three experts generating code, video and action, respectively, which are denoised jointly, with video and action under continuous flow matching and the code under discrete masked diffusion.

as continuous video latents Z_0 encoded by a pretrained variational autoencoder, and robot motion as an action chunk $A_0 = (a_t, \ldots, a_{t+H-1})$ over a horizon of H steps.

The model is conditioned on the current multi-view observations $\mathcal{O}_t$, a language instruction L, the current robot state S_t, and a compact memory M_t of the preceding program slices. Together, these inputs define the conditioning context for our joint generative model:

$$\mathcal{C}_t = \{\mathcal{O}_t, L, S_t, M_t\}, \qquad p_\theta(C_0, Z_0, A_0 \mid \mathcal{C}_t). \tag{1}$$

The unified conditional generation proceeds by jointly denoising the three modalities. Given masked code $\widetilde{C}_r$ and noisy continuous states Z_{σ_v} and A_{σ_a}, the unified model predicts

$$(\ell^c_\theta, v^v_\theta, v^a_\theta) = F_\theta(\widetilde{C}_r, Z_{\sigma_v}, A_{\sigma_a}, \sigma_v, \sigma_a; \mathcal{C}_t), \tag{2}$$

where r is the code-mask ratio, σ_v and σ_a are the video and action noise levels, ℓ^c_θ are code-token logits, and v^v_θ, v^a_θ are the video and action velocity predictions. The formulation unifies structured skill programming, future dynamics prediction, and continuous control in one generative process.

2.1 Mixture of Transformer Architecture

As shown in Figure 2, AW-0 adopts a Mixture-of-Transformers (MoT) (Liang et al., 2025) architecture with three experts for skill planning, future visual prediction, and continuous control. Each expert keeps its own attention projections (W_Q, W_K, W_V, W_O), layer normalizations, and FFNs, communicating through masked joint attention at every transformer layer. The attention mask is bidirectional within each expert and blockwise causal across experts, following the order of code, video and action. We provide additional model and training details in Appendix A.

Code Expert. The code expert augments a pretrained diffusion LLM (Nie et al., 2025) with a frozen Qwen3-VL (Bai et al., 2025) ViT. Current multi-view observations are encoded into visual tokens and prepended to the instruction and program tokens to ground skill planning in the scene.

Video and Action Experts. The video expert uses a pretrained Cosmos-Predict2.5 (NVIDIA, 2025a) video DiT to model VAE-encoded future frames conditioned on observed frames, while a separate action DiT generates continuous action chunks. Both experts receive language embeddings through cross-attention, with additional robot-state conditioning for the action expert.

2.2 Joint Denoising Generation

At each denoising step, the MoT jointly predicts code logits and continuous velocities as defined in Eq. 2. The code stream supplies planning context to the video and action experts, while the action expert also reads video representations. These connections operate throughout generation, allowing all three modalities to evolve within the same iterative process.

Modality-Specific Corruption. Video and action generation use continuous flow matching. For $m \in \{v, a\}$, with $X_0^v = Z_0$ and $X_0^a = A_0$, we construct noisy inputs as

$$X_{\sigma_m}^m = (1-\sigma_m)X_0^m + \sigma_m \epsilon_m, \qquad \epsilon_m \sim \mathcal{N}(0, I), \qquad \sigma_m = \phi_{\gamma_m}(u_m) = \frac{\gamma_m u_m}{1 + (\gamma_m - 1)u_m}. \tag{3}$$

The original video sampling distribution p_{pre} is induced by $u_v = sb$, where $b \sim \text{Beta}(7, 1)$ and $s = 0.999$; actions use $u_a \sim U(0, 1)$. Both shifts are $\gamma_m = 5$. Code instead uses discrete absorbing-mask corruption. We sample a mask ratio $r \sim U(0, 1)$ and corrupt each token independently:

$$\widetilde{c}_i = \begin{cases} [\text{MASK}], & b_i = 1, \\ c_i, & b_i = 0, \end{cases} \qquad b_i \sim \text{Bernoulli}(r). \tag{4}$$

Observed frames, visual prefixes, and language conditions remain uncorrupted. Training samples the modality-specific noise levels independently. The continuous experts regress the velocity target $\epsilon_m - X_0^m$, while the code expert minimizes cross-entropy over masked program positions.

Noise Resampling for Predictive Guidance. To provide more informative video features for action prediction, we broaden the high-noise pretrained distribution with a uniform component:

$$u_v \sim q(u) = \alpha p_{\text{pre}}(u) + (1 - \alpha)p_{\text{ref}}(u), \qquad p_{\text{ref}} = U(0, 1). \tag{5}$$

Objective-specific importance weights balance video learning and predictive guidance:

$$w_v(u) = \frac{p_{\text{pre}}(u)}{q(u)}, \qquad w_g(u) = \frac{p_{\text{ref}}(u)}{q(u)}, \qquad \mathcal{L} = \mathbb{E}_{u_v \sim q}[\lambda_v \bar{w}_v \ell_v + \bar{w}_g(\lambda_a \ell_a + \lambda_c \ell_c)]\,. \tag{6}$$

Here, ℓ_v, ℓ_a are flow-matching losses, and ℓ_c is masked-token cross-entropy; bars denote clipped weights normalized to unit batch mean. We use $\alpha = 0.5$ and leave code-mask sampling unchanged.

2.3 Agentic Coding in WAM

Automatic Annotation Pipeline. We propose an automatic annotation pipeline (Figure 3) that annotates the training data with skill planning code in three stages. We first sample one episode per task and use Gemini 3.1 Pro (Google DeepMind, 2026) to write subtask-level skill programs against predefined declaration-only APIs, while Claude Code running Claude Opus 4.8 (Anthropic, 2026) acts as a coordinator that distills recurring, task-general skills. We then freeze the skill library and re-annotate the sampled episodes with the finalized vocabulary, forming a cache of reusable subtask-to-program mappings. To annotate the remaining corpus, we reuse a cached program when the subtask's contact and motion pattern matches an existing entry; otherwise, the coding agent generates a new annotation and adds it to the cache. See Appendix J for further pipeline details.

Code Usage in Model. Annotated programs define the interface through which the code expert plans. During training, the code stream concatenates a conditioning prompt and the active slice, a short block of primitive or skill calls. Only the slice is corrupted by the absorbing-mask process of Eq. 4 and supervised by the masked-token loss. At each replanning point, the policy rebuilds the prompt and appends a canvas of G mask tokens. The canvas is decoded jointly with the video and

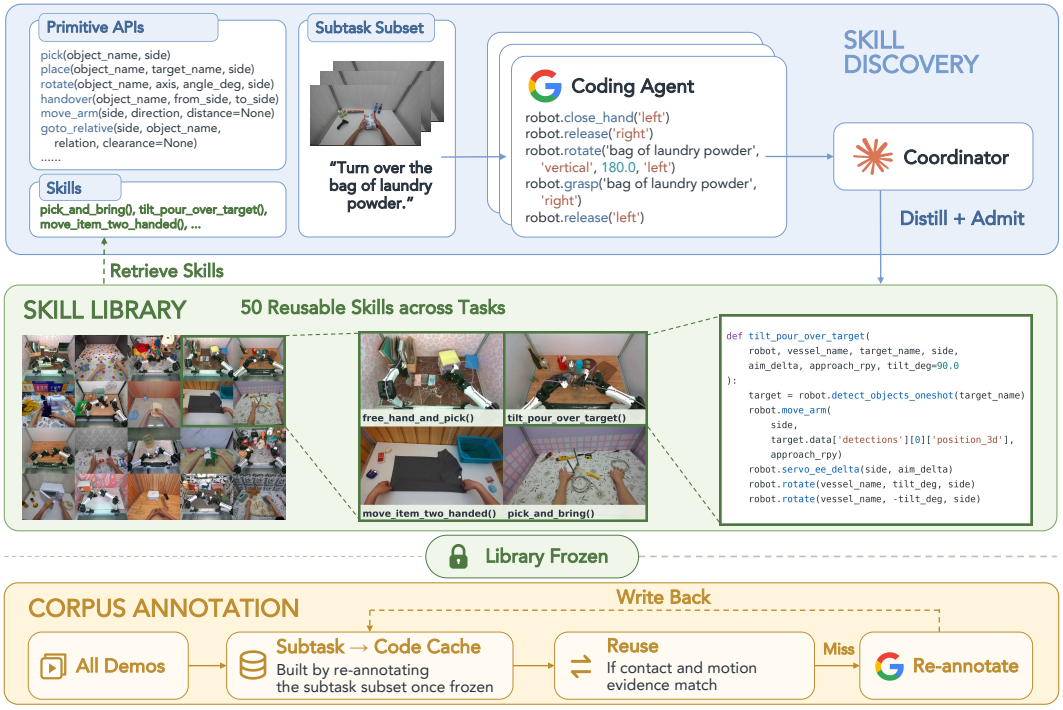


Figure 3: **Automatic Annotation Pipeline.** Coding agents generate subtask-level programs using predefined primitive APIs and available skills, while a coordinator distills recurring patterns into a library of reusable skills across tasks. With the library frozen, we annotate the corpus by reusing cached code when contact and motion evidence match and generating new labels otherwise.

action states over the T denoising steps, committing the G/T most confident masked positions per step. By default, the decoded slice is not executed. It serves as a subtask plan that conditions the action expert through joint denoising, while the predicted action chunk drives the robot.

Memory Cache. Since a single observation is often insufficient to reveal task progress, the code expert receives a compact cache of the slices generated so far. The history slices are compressed into K learnable memory slots and read back as prefix tokens before the prompt in the code stream. During training, the buffer holds the preceding annotated slices, replaced with probability p by the model's own predictions. At inference, it is reset per episode and grows with each decoded slice.

Hybrid Actions. We additionally explore an inference mode in which the decoded code slice is executed in the manner of Code-as-Policies (CaP) against a library of perception and control primitives, and the executed actions are integrated with the direct actions of the action expert. At deployment, the two channels are arbitrated at every replanning point: the decoded slice takes over the current stage only if it passes a set of validity and progress checks; otherwise the action chunk from the same forward pass drives the robot until the next replan. See Appendix H for the full algorithm.

2.4 Training Recipe

We first pretrain AW-0 on 2,000 hours of YAM robot data and 1,000 hours of egocentric human manipulation data to learn a general unified generative prior. We apply the automatic annotation pipeline in Section 2.3 to extract a reusable skill library including 50 skills and annotate demonstrations with temporally aligned program slices. The annotated data support joint pretraining of code, video, and action generation, connecting structured skill plans with visual dynamics and continuous control across embodiments and scenarios. We further include the details of dataset statistics, preprocessing, and state-action representations in Appendix B. For each downstream task, we reuse the same annotation pipeline under the frozen skill library to label task-specific demonstrations and jointly fine-tune AW-0 on the resulting code, video, and action targets using the objective in Eq. 6.

Table 1: **RoboTwin 2.0 Results under *Clean* and *Random* Settings.** Success rate (%) averaged over 50 tasks with 50 rollouts each. Per-task numbers are reported in Table 11.

Method	Clean	Random	Average
$\pi_{0.5}$	54.9	27.7	41.3
DreamZero	33.6	17.6	25.6
EgoScale	26.2	2.9	14.5
WLA	64.8	3.6	34.2
AW-0 (ours)	**80.8**	20.6	**50.7**
AW-0 (hybrid)	63.1	**29.2**	46.2

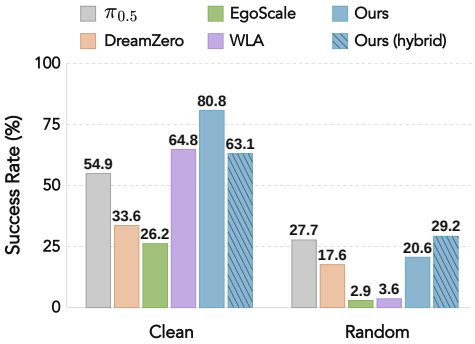


Figure 4: **Average Scores from Table 1.**

3 Experiments

We evaluate AW-0 on RoboTwin 2.0 (Chen et al., 2025) and 8 real-world tasks spanning parallel-gripper and dexterous-hand embodiments, assessing language following, semantic reasoning, and skill execution, as well as the robustness benefits of hybrid execution in variant scenarios.

Baselines. We compare four baselines: (1) $\pi_{0.5}$ (Black et al., 2025), a state-of-the-art VLA with strong semantic reasoning; (2) EgoScale (Zheng et al., 2026), a VLA pretrained on large-scale egocentric human data; (3) DreamZero (Ye et al., 2026), a WAM jointly generating future video and actions; and (4) WLA (Yang et al., 2026), an autoregressive model jointly predicting textual subtasks, subgoal images, and actions. All methods use the same task-specific demonstrations for post-training and the same evaluation protocol. Implementation details are provided in Appendix C.

3.1 Simulation Experiment

Benchmarks. We evaluate AW-0 on RoboTwin 2.0. Following the standard protocol, we train on 50 clean demonstrations per task (2,500 episodes in total) and evaluate under the benchmark's two settings: *clean*, with a clean tabletop, plain background, fixed lighting, and no distractors; and *random*, with domain randomization over background and tabletop textures, lighting, tabletop height, and distractor objects. We report the mean success rate over 50 rollouts per task.

Hybrid Evaluation. We additionally evaluate AW-0 in the hybrid mode, in which the training demonstrations are collected by executing CaP programs written against a set of CaP-X-style (Fu et al., 2026) primitive APIs that we develop for RoboTwin 2.0. The programs are written automatically by Claude Opus 4.8, following the agentic framework of ASPIRE (Lu et al., 2026).

Main Results. Table 1 and Figure 4 report the results on RoboTwin 2.0. The two inference modes exhibit complementary strengths. Overall, AW-0 achieves the highest average success rate of 50.7% across both settings, outperforming all baselines including models designed for broad generalization. The advantage comes mostly from the clean setting, where AW-0 reaches 80.8%, exceeding the strongest baseline WLA (64.8%) by 16.0 points. Under visual randomization, performance drops to 20.6% due to pixel-level O.O.D. prediction (Yan et al., 2026), but still outperforms the strongest WAM baseline DreamZero (17.6%) and remains far ahead of EgoScale (2.9%) and WLA (3.6%).

We further observe that hybrid execution improves robustness under distribution shift. In the random setting, it raises the success rate from 20.6% to 29.2%, outperforming all baselines including the strong VLA baseline $\pi_{0.5}$ (27.7%). Relative to its clean-setting performance, the hybrid mode retains 46% of its success rate under randomization, compared with 25% for the default mode. This robustness comes at a cost in the clean setting, where the hybrid mode reaches 63.1%, on par with the strongest baseline, but 17.7 points below the default mode. Further analysis refer to Appendix I.

3.2 Real-world Experiment

Robot Platforms and Action Space. We conduct real-world experiments on two platforms: the YAM station, a fixed-base bimanual setup with two 6-DoF arms and parallel grippers, and the

Table 2: **Real-world Results on 8 Tasks across 2 Embodiments.** Average rubric score (%) over 3×16 rollouts/task (Appendix E).

Method	*YAM (parallel grippers)*				*Tianji–Sharpa (dexterous hands)*				Avg.
	Arrange Fruits	Pack Backpack	Replace Lightbulb	Fold Towels	Fruits w/ Tong	Deal Poker	Flip Scoreboard	Solve Cube	
$\pi_{0.5}$	70.8	49.7	63.8	81.7	7.3	13.1	26.3	36.1	43.6
DreamZero	53.3	36.7	67.4	79.8	22.6	37.1	28.4	42.5	46.0
EgoScale	37.5	27.3	60.2	53.3	17.6	25.6	40.8	24.9	35.9
WLA	39.6	29.6	56.0	74.6	14.2	22.9	29.7	52.3	39.9
AW-0 (ours)	**91.7**	**64.6**	**75.0**	**84.4**	**54.2**	**58.3**	**68.8**	**72.1**	**71.1**

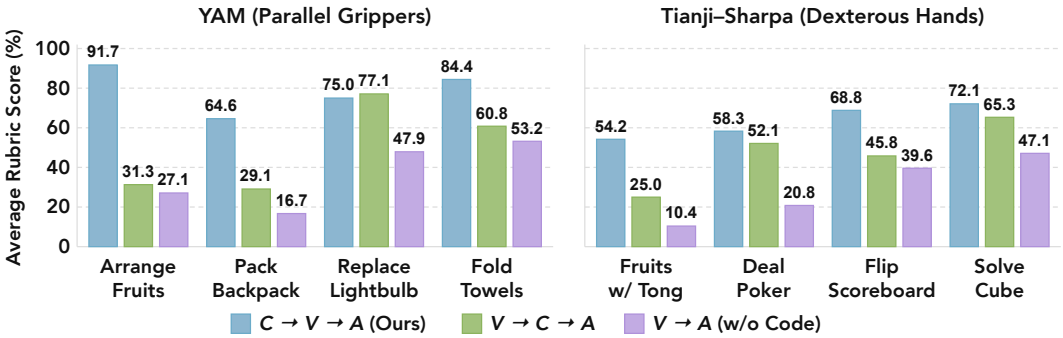


Figure 5: **Impact of Code Expert and Attention Structure.** Avg. rubric score (%) over 3×16 rollouts/task. C, V, and A denote the code, video, and action streams.

Tianji–Sharpa platform, a bimanual robot with two 22-DoF Sharpa dexterous hands. Both platforms observe three RGB views and act in the unified 64-D state–action space defined in Appendix A. Further details on the robot setups are provided in Appendix D.

Evaluation Protocol and Metrics. We evaluate on the 8 real-world tasks listed in Table 2, 4 on each platform, with 3×16 rollouts per task under randomized initial object poses. Task definitions, evaluation settings, and per-task scoring rubrics are provided in Appendix E.

Quantitative and Qualitative Analysis. As shown in Table 2, AW-0 achieves the highest average rubric score across both embodiments, outperforming the strongest baseline by 25.1 points. Two observations emerge. **Language following and semantic reasoning:** AW-0 outperforms the baselines strongly on *Arrange Fruits* and *Flip Scoreboard*, which require instruction-specific object selection and counting, respectively. The VLA baselines, particularly $\pi_{0.5}$, also show relatively strong semantic understanding, while the WAM baselines are less reliable on these properties. These results suggest that code-based planning strengthens the language-following and semantic reasoning of world action models. **Robust skill execution:** On contact-rich and dexterous tasks, DreamZero outperforms $\pi_{0.5}$ on *Replace Lightbulb* and *Deal Poker*, where predicting future states provides stable physical-dynamics features for action prediction. AW-0 further outperforms these baselines by combining guidance from generated skill programs with predictive representations of physical dynamics, aligning skill-level planning with continuous control. See Appendix G for further analysis.

Hybrid Execution in the Real World. We further evaluate hybrid execution on *Fold and Stack Towels* with the YAM platform based on the hybrid algorithm H, comparing it against action-chunk execution under the same post-trained model and evaluation conditions. Under the same task rubric, hybrid execution achieves **90.6%**, compared with **84.4%** for action-chunk execution, while the direct execution steps of code further enhance the model's reasoning capabilities in distinguishing stacking order and action stability. Further visualizations are provided in Appendix G.3.

3.3 Ablation Study

We organize our ablation study mainly around three research questions: 1) how to integrate a visually grounded code expert into a WAM framework, 2) which components of the planning representation

Table 4: **Impact of Code Generation Paradigm.** Avg. rubric score (%) over 3 × 16 rollouts/task.

Task	**Joint denoise (ours)**	AR
Arrange Fruits	**91.7**	81.3
Pack Backpack	**64.6**	62.5
Replace Lightbulb	75.0	**79.2**
Fold Towels	**84.4**	53.2
Fruits w/ Tong	**54.2**	20.8
Deal Poker	**58.3**	44.1
Flip Scoreboard	**68.8**	39.3
Solve Cube	**72.1**	55.8
Avg.	**71.1**	54.5

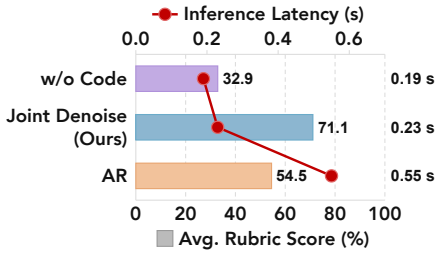


Figure 6: **Performance and Inference Latency.** Average rubric score over the 8 real-world tasks (bars) and inference latency on an NVIDIA RTX 5090 (red line).

contribute to task performance, and 3) how joint denoising affects task performance and inference efficiency. All the variants are post-trained and evaluated under the protocol of Section 3.2.

Q1: How should the code expert be integrated into a WAM? We first compare three blockwise causal attention orders during joint denoising: $V \to A$ (Vanilla WAM), $V \to C \to A$, and $C \to V \to A$ (Figure 5). Adding code through $V \to C \to A$ improves performance by grounding the predicted dynamics to action, but restricts its role to reasoning over video predictions without guiding or correcting them. With $C \to V \to A$, both video and action generation attend to code, allowing semantic task plans to shape future prediction throughout denoising. The performance gain of this ordering supports introducing semantic reasoning directly into video generation, which improves language following and provides more reliable guidance for stable action generation.

Table 3: **Visual Conditioning for Code Expert.** Avg. rubric score (%) over 3 × 16 rollouts/task.

Variant	Arrange Fruits	Pack Backpack	Replace Lightbulb	Fold Towels	Fruits w/ Tong	Deal Poker	Flip Scoreboard	Solve Cube	Avg.
ViT (ours)	**91.7**	**64.6**	**75.0**	**84.4**	**54.2**	**58.3**	**68.8**	**72.1**	**71.1**
VAE	83.3	39.6	66.7	61.3	41.7	47.9	50.0	53.9	55.6

We then compare a dedicated ViT against reused Wan2.2 VAE features for conditioning the code expert (Table 3). ViT conditioning improves performance on all 8 tasks, with substantial gains on *Pack Backpack* (64.6% vs. 39.6%) and *Solve Rubik's Cube* (72.1% vs. 53.9%), which require fine-grained semantic reasoning about object attributes and task states. VAE features are optimized for pixel-level reconstruction (Yan et al., 2026) and thus poorly support the semantic distinctions needed to select and compose skills, whereas ViT feature enables the code expert to translate visual observations into more accurate skill plans.

Q2: What makes code an effective planning representation? To assess the planning content produced by our annotation pipeline, we compare *(1) full annotated programs*, *(2) code without comments*, *(3) code without skills*, and *(4) natural-language subgoals* in Figure 7. Full programs outperform comment-free code on all 8 shared tasks, suggesting that explanatory language helps connect program structure with task intent. Removing skill calls also reduces performance on all 8 evaluated tasks, especially on skill-required tasks such as *Deal Poker* (58.3% vs. 42.0%). Since this variant still uses structured code, the drop supports the value of reusable skill abstractions beyond primitive-level API composition. Full programs also beat language-only subgoals on 7 of 8 shared tasks, showing that structured semantic guidance can provide more action-centric representation as condition. Together, these comparisons support combining explanatory language, reusable skills, and explicit program structure to provide effective planning guidance.

Q3: How does joint denoising affect performance and efficiency? As shown in Table 4, joint denoising outperforms AR on 7 of 8 tasks. Under teacher-forced AR training, early errors accumulate and propagate into video prediction through the conditioning path. Joint denoising instead reconstructs masked code alongside video and actions, providing evolving semantic guidance throughout generation. This offers a mechanism to mitigate error propagation, consistent with the observed performance gains. We also test the inference latency of the two paradigms on NVIDIA RTX 5090 and

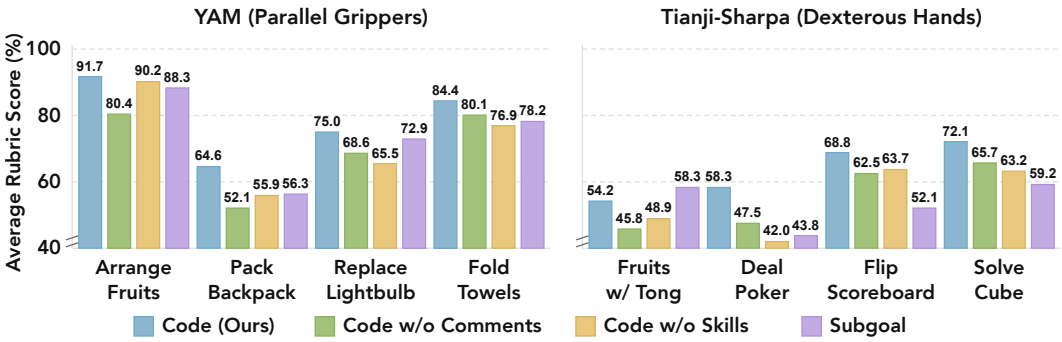


Figure 7: **Impact of Code Label Content.** Average rubric score (%) over 3×16 rollouts per task.

other inference settings in Appendix A. Joint denoising takes 0.23 s versus 0.55 s for AR, a 2.39× speedup (Figure 6). Compared with the WAM without code (0.19 s), it adds 0.04 s (21.1%) while improving performance on all 8 shared tasks (Figure 5).

4 Related Work

World Action Models (WAMs). World models predict future observations and have recently emerged as a powerful paradigm for robot manipulation, serving as learned simulators and data generators for policy learning (NVIDIA, 2025a;b; Gao et al., 2026; Yang et al., 2024; Jang et al., 2025). Building on this progress, World Action Models (WAMs) unify future prediction and action generation, through joint video-action diffusion (Ye et al., 2026; Zhu et al., 2025), expert-based architectures (Bi et al., 2025; Cai et al., 2026), or direct adaptation of pretrained video models into policies (Kim et al., 2026). Recent WAMs further couple autoregressive sub-task prediction with future-state modeling and action synthesis (Yang et al., 2026) or dynamically update subtask instructions to condition video and action generation (Tang et al., 2026). However, existing WAMs lack structured semantic guidance and explicit skill invocation, limiting their ability to perform complex semantic planning and structure predicted dynamics into actionable behavioral abstractions.

Agentic Robotics. Early work composed perception and control APIs into executable closed-loop behaviors without policy training (Ahn et al., 2022; Liang et al., 2023; Singh et al., 2023). Later systems generate full Python programs over perception APIs (Huang et al., 2023; Mu et al., 2024; Goldberg et al., 2025), and executable code has proven a stronger action space for LLM agents than text (Wang et al., 2024). With the rise of coding agents, program synthesis has become explicitly agentic, with recent systems iteratively writing, executing, and repairing control programs (Lu et al., 2026; Fu et al., 2026). Frontier models now also act as controllers themselves, reasoning over instructions and feedback and driving robots directly through native tool calls and learned primitives (Shi et al., 2025; Li et al., 2026; Gemini Robotics Team, 2025a;b). In all of these systems, the agent controls the robot from outside the policy and is bounded by the primitives it can call. We instead bring code inside an end-to-end model as a high-level planning interface and further explore how the generated code can be executed to complement the model's own actions at test time.

5 Conclusion

We present AW-0, an agentic world action model that integrates code-based skill planning with predictive dynamics and continuous control. Through joint denoising, the code expert provides structured semantic guidance for video and action generation. An automated annotation pipeline enables learning this planning interface from large-scale robot and human demonstrations. Experiments on RoboTwin 2.0 and real-world manipulation with YAM and dexterous hands demonstrate improvements in language following, semantic reasoning, and skill execution, with hybrid execution further enhancing robustness. These results establish a unified foundation for future work on skill reuse and iterative interaction between programmatic planning, world prediction, and execution, providing a potential agentic system foundation that continually interacts with and learns from environments.

AI USE STATEMENT

We used Gemini 3.1 Pro and Claude Code with Claude Opus 4.8 in our automatic annotation pipeline to generate demonstration-aligned program labels and distill reusable skill libraries (Section 2.3). Claude Opus 4.8 was also used to generate executable programs for collecting simulation demonstrations (Section 3.1). Additionally, we used GPT-6 Astra for language polishing and stylistic refinement, as well as part of figure preparation.

ETHICS STATEMENT

This work studies code-guided world action models for robotic manipulation using robot and egocentric human demonstrations. All pretraining datasets are used in accordance with their respective licenses and data-use terms. All post-training demonstrations are collected by our team for the tasks evaluated in this work.

REPRODUCIBILITY STATEMENT

We provide detailed information in Appendix A regarding the model architecture design, parameter settings, and computational resources. Regarding training data, Appendix B details the composition of the pre-training dataset. Section 2.3 introduces the complete data annotation pipeline. While prompts for the agentic framework and performance analysis of the pre-trained model are provided in Appendix J.2 and Appendix F.1. For downstream tasks, we also present standardized data generation criteria and methods to facilitate extension to any further tasks.

REFERENCES

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, 2022. arXiv:2204.01691.

Anthropic. Claude Opus 4.8 System Card, 2026. URL https://www-cdn.anthropic.com/0b4915911bb0d19eca5b5ee635c80fef830a37ea.pdf.

Mido Assran, Adrien Bardes, David Fan, Quentin Garrido, Russell Howes, Mojtaba, Komeili, Matthew Muckley, Ammar Rizvi, Claire Roberts, Koustuv Sinha, Artem Zholus, Sergio Arnaud, Abha Gejji, Ada Martin, Francois Robert Hogan, Daniel Dugas, Piotr Bojanowski, Vasil Khalidov, Patrick Labatut, Francisco Massa, Marc Szafraniec, Kapil Krishnakumar, Yong Li, Xiaodong Ma, Sarath Chandar, Franziska Meier, Yann LeCun, Michael Rabbat, and Nicolas Ballas. V-jepa 2: Self-supervised video models enable understanding, prediction and planning, 2025. URL https://arxiv.org/abs/2506.09985.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibo Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Hongzhe Bi, Hengkai Tan, Shenghao Xie, Zeyuan Wang, Shuhe Huang, Haitian Liu, Ruowen Zhao, Yao Feng, Chendong Xiang, Yinze Rong, Hongyan Zhao, Hanyu Liu, Zhizhong Su, Lei Ma, Hang Su, and Jun Zhu. Motus: A unified latent action world model, 2025. URL https://arxiv.org/abs/2512.13030.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024. URL https://arxiv.org/abs/2410.24164.

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

Junhao Cai, Zetao Cai, Jiafei Cao, Yilun Chen, Zeyu He, Lei Jiang, Hang Li, Hengjie Li, Yang Li, Yufei Liu, Yanan Lu, Qi Lv, Haoxiang Ma, Jiangmiao Pang, Yu Qiao, Zherui Qiu, Yanqing Shen, Xu Shi, Yang Tian, Bolun Wang, Hanqing Wang, Jiaheng Wang, Tai Wang, Xueyuan Wei, Chao Wu, Yiman Xie, Boyang Xing, Yuqiang Yang, Yuyin Yang, Qiaojun Yu, Feng Yuan, Jia Zeng, Jingjing Zhang, Shenghan Zhang, Shi Zhang, Zhuoma Zhaxi, Bowen Zhou, Yuanzhen Zhou, Yunsong Zhou, Hongrui Zhu, Yangkun Zhu, and Yuchen Zhu. Internvla-a1: Unifying understanding, generation and action for robotic manipulation, 2026. URL https://arxiv.org/abs/2601.02456.

Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Zixuan Li, Qiwei Liang, Xianliang Lin, Yiheng Ge, Zhenyu Gu, Weiliang Deng, Yubin Guo, Tian Nian, Xuanbing Xie, Qiangyu Chen, Kailun Su, Tianling Xu, Guodong Liu, Mengkang Hu, Huan ang Gao, Kaixuan Wang, Zhixuan Liang, Yusen Qin, Xiaokang Yang, Ping Luo, and Yao Mu. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation, 2025. URL https://arxiv.org/abs/2506.18088.

Shuang Cheng, Yihan Bian, Dawei Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, and Bowen Zhou. Sdar: A synergistic diffusion-autoregression paradigm for scalable sequence generation, 2025. URL https://arxiv.org/abs/2510.06303.

Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction, 2016. URL https://arxiv.org/abs/1605.07157.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Dantong Niu, Fei-Fei Li, Guanya Shi, Jiajun Wu, Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi Fan. Cap-x: A framework for benchmarking and improving coding agents for robot manipulation. In *ICML*, 2026. arXiv:2603.22435.

Shenyuan Gao, William Liang, Kaiyuan Zheng, Ayaan Malik, Seonghyeon Ye, Sihyun Yu, Wei-Cheng Tseng, Yuzhu Dong, Kaichun Mo, Chen-Hsuan Lin, Qianli Ma, Seungjun Nah, Loic Magne, Jiannan Xiang, Yuqi Xie, Ruijie Zheng, Dantong Niu, You Liang Tan, K.R. Zentner, George Kurian, Suneel Indupuru, Pooya Jannaty, Jinwei Gu, Jun Zhang, Jitendra Malik, Pieter Abbeel, Ming-Yu Liu, Yuke Zhu, Joel Jang, and Linxi "Jim" Fan. Dreamdojo: A generalist robot world model from large-scale human videos. *arXiv preprint arXiv:2602.06949*, 2026.

Gemini Robotics Team. Gemini robotics: Bringing AI into the physical world. *arXiv preprint arXiv:2503.20020*, 2025a.

Gemini Robotics Team. Gemini robotics 1.5: Pushing the frontier of generalist robots with advanced embodied reasoning, thinking, and motion transfer. *arXiv preprint arXiv:2510.03342*, 2025b.

Andrew Goldberg, Kavish Kondap, Tianshuang Qiu, Zehan Ma, Letian Fu, Justin Kerr, Huang Huang, Kaiyuan Chen, Kuan Fang, and Ken Goldberg. Blox-Net: Generative design-for-robot-assembly using VLM supervision, physics simulation, and a robot with reset. In *IEEE International Conference on Robotics and Automation*, 2025. arXiv:2409.17126.

Google DeepMind. Gemini 3.1 Pro Model Card, 2026. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf.

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners, 2021. URL https://arxiv.org/abs/2111.06377.

Siyuan Huang, Zhengkai Jiang, Hao Dong, Yu Qiao, Peng Gao, and Hongsheng Li. Instruct2Act: Mapping multi-modality instructions to robotic actions with large language model. *arXiv preprint arXiv:2305.11176*, 2023.

Joel Jang et al. Dreamgen: Unlocking generalization in robot learning through video world models, 2025. URL https://arxiv.org/abs/2505.12705.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. Openvla: An open-source vision-language-action model, 2024. URL https://arxiv.org/abs/2406.09246.

Moo Jin Kim et al. Cosmos policy: Fine-tuning video models for visuomotor control and planning, 2026. URL https://arxiv.org/abs/2601.16163.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies, 2016. URL https://arxiv.org/abs/1504.00702.

Ruiying Li, Yunlang Zhou, YuYao Zhu, Kylin Chen, Jingyuan Wang, Sukai Wang, Kongtao Hu, Minhui Yu, Bowen Jiang, Zhan Su, Jiayao Ma, Xin He, Yongjian Shen, Yang Yang, Guanghui Ren, Maoqing Yao, Wenhao Wang, and Yao Mu. Roboclaw: An agentic framework for scalable long-horizon robotic tasks. *arXiv preprint arXiv:2603.11558*, 2026.

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 *ICRA*, 2023.

Weixin Liang, Lili Yu, Liang Luo, Srinivasan Iyer, Ning Dong, Chunting Zhou, Gargi Ghosh, Mike Lewis, Wen tau Yih, Luke Zettlemoyer, and Xi Victoria Lin. Mixture-of-transformers: A sparse and scalable architecture for multi-modal foundation models, 2025. URL https://arxiv.org/abs/2411.04996.

Runyu Lu, Yubo Wu, Ethan Kou, Letian Fu, Wenli Xiao, Ajay Mandlekar, Yinzhen Xu, Guanya Shi, Ken Goldberg, Ang Chen, Mosharaf Chowdhury, Yuke Zhu, Linxi Fan, and Guanzhi Wang. ASPIRE: Agentic skills discovery for robotics. *arXiv preprint arXiv:2607.00272*, 2026.

Yao Mu, Junting Chen, Qinglong Zhang, et al. Robocodex: Multimodal code generation for robotic behavior synthesis. In *ICML*, 2024.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models, 2025. URL https://arxiv.org/abs/2502.09992.

Dantong Niu, Yuvan Sharma, Giscard Biamby, Jerome Quenum, Yutong Bai, Baifeng Shi, Trevor Darrell, and Roei Herzig. Llarva: Vision-action instruction tuning enhances robot learning, 2024. URL https://arxiv.org/abs/2406.11815.

Dantong Niu, Yuvan Sharma, Haoru Xue, Giscard Biamby, Junyi Zhang, Ziteng Ji, Trevor Darrell, and Roei Herzig. Pre-training auto-regressive robotic models with 4d representations. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=2FDsh5D2Th.

NVIDIA. Cosmos world foundation model platform for physical ai, 2025a. URL https://arxiv.org/abs/2501.03575.

NVIDIA. World simulation with video foundation models for physical ai, 2025b. URL https://arxiv.org/abs/2511.00062.

Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting, 2016. URL https://arxiv.org/abs/1604.07379.

Lucy Xiaoyang Shi, Brian Ichter, Michael Equi, Liyiming Ke, Karl Pertsch, Quan Vuong, James Tanner, Anna Walling, Haohuan Wang, Niccolo Fusai, Adrian Li-Bell, Danny Driess, Lachy Groom, Sergey Levine, and Chelsea Finn. Hi Robot: Open-ended instruction following with hierarchical vision-language-action models. In *International Conference on Machine Learning*, 2025. arXiv:2502.19417.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *ICRA*, 2023.

Yinzhou Tang, Jingbo Xu, Yu Shang, Zihao Song, Chen Gao, Wei Wu, and Yong Li. Dreaming when necessary: Advancing world action models with adaptive multi-modal reasoning, 2026. URL https://arxiv.org/abs/2606.07089.

Zhan Tong, Yibing Song, Jue Wang, and Limin Wang. Videomae: Masked autoencoders are data-efficient learners for self-supervised video pre-training, 2022. URL https://arxiv.org/abs/2203.12602.

Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, Jianyuan Zeng, Jiayu Wang, Jingfeng Zhang, Jingren Zhou, Jinkai Wang, Jixuan Chen, Kai Zhu, Kang Zhao, Keyu Yan, Lianghua Huang, Mengyang Feng, Ningyi Zhang, Pandeng Li, Pingyu Wu, Ruihang Chu, Ruili Feng, Shiwei Zhang, Siyang Sun, Tao Fang, Tianxing Wang, Tianyi Gui, Tingyu Weng, Tong Shen, Wei Lin, Wei Wang, Wei Wang, Wenmeng Zhou, Wente Wang, Wenting Shen, Wenyuan Yu, Xianzhong Shi, Xiaoming Huang, Xin Xu, Yan Kou, Yangyu Lv, Yifei Li, Yijing Liu, Yiming Wang, Yingya Zhang, Yitong Huang, Yong Li, You Wu, Yu Liu, Yulin Pan, Yun Zheng, Yuntao Hong, Yupeng Shi, Yutong Feng, Zeyinzi Jiang, Zhen Han, Zhi-Fan Wu, and Ziyu Liu. Wan: Open and advanced large-scale video generative models, 2025. URL https://arxiv.org/abs/2503.20314.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Peng, Hao Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning*, 2024. arXiv:2402.01030.

Wenli Xiao, Jia Xie, Tonghe Zhang, Haotian Lin, Letian "Max" Fu, Haoru Xue, Jalen Lu, Yi Yang, Cunxi Dai, Zi Wang, Jimmy Wu, Guanzhi Wang, S. Shankar Sastry, Ken Goldberg, Linxi "Jim" Fan, Yuke Zhu, and Guanya Shi. Enpire: Agentic robot policy self-improvement in the real world, 2026. URL https://arxiv.org/abs/2606.19980.

Haodong Yan, Junfeng Li, Junjie He, Zhide Zhong, MingMing Yu, Wenxuan Song, Jiaguan Zhu, Yangyang Zheng, Yuqiao Du, Jiadi You, Yingjie Cai, Xu Yan, Guanyi Zhao, Bingbing Liu, and Haoang Li. Robust-wam: Bridging generative pretraining and semantic foresight in world-action models, 2026. URL https://arxiv.org/abs/2608.05903.

Sherry Yang, Yilun Du, Seyed Kamyar Seyed Ghasemipour, Jonathan Tompson, Leslie Kaelbling, Dale Schuurmans, and Pieter Abbeel. Learning interactive real-world simulators. In *International Conference on Learning Representations*, 2024.

Yi Yang, Zhihong Liu, Siqi Kou, Yiyang Chen, Yanzhe Hu, Jianbo Zhou, Boyuan Zhao, Zhijie Wei, Xiao Xia, Xueqi Li, Pengfei Liu, and Zhijie Deng. World-language-action model for unified world modeling, language reasoning, and action synthesis, 2026. URL https://arxiv.org/abs/2606.05979.

Seonghyeon Ye, Yunhao Ge, Kaiyuan Zheng, Shenyuan Gao, Sihyun Yu, George Kurian, Suneel Indupuru, You Liang Tan, Chuning Zhu, Jiannan Xiang, Ayaan Malik, Kyungmin Lee, William Liang, Nadun Ranawaka, Jiasheng Gu, Yinzhen Xu, Guanzhi Wang, Fengyuan Hu, Avnish

Narayan, Johan Bjorck, Jing Wang, Gwanghyun Kim, Dantong Niu, Ruijie Zheng, Yuqi Xie, Jimmy Wu, Qi Wang, Ryan Julian, Danfei Xu, Yilun Du, Yevgen Chebotar, Scott Reed, Jan Kautz, Yuke Zhu, Linxi "Jim" Fan, and Joel Jang. World action models are zero-shot policies, 2026. URL https://arxiv.org/abs/2602.15922.

Tianyuan Yuan et al. Fast-wam: Do world action models need test-time future imagination?, 2026. URL https://arxiv.org/abs/2603.16666.

Junyi Zhang, Jiaxin Ge, Hanjun Yoo, Letian Fu, Zihan Yang, Yaowei Liu, Raj Saravanan, Shaofeng Yin, Justin Yu, Dantong Niu, Zirui Wang, Roei Herzig, Ken Goldberg, Yutong Bai, David M. Chan, Ion Stoica, Angjoo Kanazawa, Jiahui Lei, Haiwen Feng, and Trevor Darrell. Playful agentic robot learning, 2026. URL https://arxiv.org/abs/2606.19419.

Ruijie Zheng, Dantong Niu, Yuqi Xie, Jing Wang, Mengda Xu, Yunfan Jiang, Fernando Castañeda, Fengyuan Hu, You Liang Tan, Letian Fu, Trevor Darrell, Furong Huang, Yuke Zhu, Danfei Xu, and Linxi Fan. Egoscale: Scaling dexterous manipulation with diverse egocentric human data, 2026. URL https://arxiv.org/abs/2602.16710.

Chuning Zhu, Raymond Yu, Siyuan Feng, Benjamin Burchfiel, Paarth Shah, and Abhishek Gupta. Unified world models: Coupling video and action diffusion for pretraining on large robotic datasets, 2025. URL https://arxiv.org/abs/2504.02792.

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 CONTENTS

A Model and Training Details **16**

B Analysis on Pretraining Dataset **17**

B.1 Pretraining Dataset Statistics 17

B.2 Skill Library Definitions 18

B.3 Visualizations of YAM Skills 20

B.4 Visualizations of Egocentric Human Skills 21

B.5 Skill Distribution in the Embedding Space 24

C Baseline Implementation Details **26**

D Real-world Experimental Setup **27**

E Real-world Task Progress **28**

E.1 YAM Tasks 28

E.2 Tianji–Sharpa Tasks 31

F Analysis on Code Generation Quality **34**

F.1 Pre-trained Code Generation Quality 34

F.2 Post-trained Code Generation Quality 36

G Additional Analysis on Real-world Results **38**

G.1 Impact of Code Expert on Language Following Capability 38

G.2 Impact of Code Expert on Skill Robustness 38

G.3 Hybrid Actions Evaluation Details 38

H Hybrid Actions Algorithm **39**

I Additional Simulation Experiment Details **40**

I.1 RoboTwin Per-Task Breakdown 40

I.2 Hybrid Actions Evaluation Details 42

J Agentic Annotation Details **42**

J.1 Low-level Primitives 42

J.2 Annotation Agent Prompts and Coordinator Protocol 42

K Limitations and Future Work **45**

A Model and Training Details

We implement the AW-0 architecture using three modality-specific experts coupled by masked joint attention. All experts have 28 transformer layers and project their queries, keys, and values into a common 2,048-dimensional attention space. The code expert receives current visual observations and language instructions, the video expert attends to the code stream, and the action expert attends to both. Tables 5 and 6 summarize the architecture and the pretraining configuration.

State and Action Representation. Both YAM robot data and egocentric human data are mapped into a shared 64-dimensional state–action space. Each side contains a 9-dimensional end-effector pose (3D of x, y, z position and a 6D rotation representation), 22 dexterous-hand joint angles, and one parallel gripper width, giving $2 \times (9 + 22 + 1) = 64$ dimensions. States are all absolute; while action chunks use end-effector pose deltas relative to the first frame of the chunk, with absolute gripper targets for YAM and absolute hand-joint targets for human demonstrations/dexterous hand. Dimensions unused by an embodiment—the hand-joint blocks for YAM and the gripper dimensions for human demonstrations—are set to zero and left unnormalized. All remaining dimensions are normalized using per-dimension 1st and 99th percentiles.

Initialization and Trainable Components. The video expert is initialized from a pretrained Cosmos-Predict2.5-2B (NVIDIA, 2025a) video model, whereas the action transformer is initialized from scratch before joint pretraining. The code expert reuses the token embeddings, layer normalizations, feed-forward networks, and vocabulary head of SDAR-1.7B (Cheng et al., 2025). These pretrained components remain frozen, while its MoT attention projections and modulation parameters are trainable. The separate Qwen3-VL (Bai et al., 2025) vision tower is frozen, with a trainable merger that aligns its features to the code embedding space. The frozen Wan2.2 (Wan et al., 2025) VAE has 48 latent channels, and the video transformer uses $1 \times 2 \times 2$ latent patches with multi-view 3D RoPE. Language conditioning for the video and action experts uses precomputed Cosmos-Reason1 embeddings. During downstream adaptation, shape-compatible parameters are transferred from the jointly pretrained model, and the action and state projections are reinitialized when the embodiment requires different dimensions.

Table 5: **Expert Architectures and Shared Visual Inputs.**

Configuration	Code	Video	Action
Backbone	SDAR-1.7B	Cosmos-Predict2.5-2B	Action DiT
Residual width	2,048	2,048	1,024
FFN width	6,144	8,192	4,096
Attention	Bidirectional	Bidirectional	Bidirectional
Transformer layers	28		
Query / KV heads	16 / 16		
Head / attention width	128 / 2,048		
ViT encoder	Qwen3-VL-2B tower		
Vision depth / width	24 / 1,024		
Spatial patch / merge	16×16 / 2×2		
Visual tokens / width	210 / 2,048		
Views / resolution	3 / 224×320		
Video frames / stride	9 / 4		
State / action width	64 / 64 (Real-world); 14 / 14 (RoboTwin)		
Action horizon	32		

Observation and Program Representations. Images are sampled at timestep offsets $0, 4, \ldots, 32$, giving nine frames per view; only the first frame is used as observation conditioning. The code expert's visual encoder also receives only this first frame from each view. At 224×320 resolution, spatial merging produces 70 visual tokens per view. The code expert uses the Qwen3-family tokenizer of the SDAR checkpoint. Its 512-token training budget applies to the instruction and program together, with the 210-token visual prefix added separately. At inference, the default program

canvas contains 128 tokens. Visual-prefix, instruction, and padding positions are excluded from the standard masked-program loss.

Table 6: **Generation and Pretraining Settings.** Downstream overrides are given in the text.

Generation		**Optimization**	
Video / Action loss	Flow matching	Optimizer	AdamW
Code loss	Masked CE	Adam coefficients	$(0.9, 0.95)$
Video mixture α	0.5	Peak learning rate	10^{-4}
Sigma shifts (v, a)	$(5, 5)$	LR schedule	Cosine
Action u_a / code r	$U(0, 1)$	Linear warmup	5%
Loss weights (v, c, a)	$(1, 0.5, 1)$	Weight decay	0
Importance clip	4.0	Gradient clip	1.0
Weight normalization	Unit batch mean	Precision / ZeRO	bf16 / Stage 1
Mixture sampling	Stratified	Checkpointing	Enabled
Code / video dropout	0.15 / 0	Seed	42
Video / action solver	UniPC / Euler	Nodes / GPUs	6 / 48
Code order	Confidence	Batch per GPU	6
Reveal ρ / warmup	1 / 0	Accumulation	9
Timestep scale	1,000	Effective batch	2,592
Code canvas	128 tokens	Training budget	1 epoch

Joint Training and Decoding. Video noise resampling follows Section 2.2. The pretrained variable is $u_{\text{pre}} = 0.999b$, where $b \sim \text{Beta}(7, 1)$; training samples are drawn from $q = 0.5p_{\text{pre}} + 0.5U(0, 1)$. The video objective uses the pretrained-density weight, while action and code losses use the uniform-reference weight. Action noise levels and code-mask ratios are sampled independently of the video variable. At least one program token is masked in every nonempty training target. The pretraining recipe additionally replaces the instruction and program inputs with mask tokens for 15% of training batches, retaining the visual prefix. The default decoder progressively commits high-confidence code predictions across the full joint trajectory. Early completion is controlled by $\rho < 1$, a separate decoding-time choice that does not change the training mask distribution.

Downstream Optimization. Downstream runs use the same optimizer, warmup schedule, and mixed-precision settings as pretraining, with task-specific batch sizes and training budgets. The YAM configurations use a per-GPU batch size of four, one gradient accumulation step, a learning rate of 10^{-4}, and weight decay of 10^{-2}. The RoboTwin configuration uses two 8-GPU nodes with a per-GPU batch size of 6 and one accumulation step, giving an effective batch size of 96, a learning rate of 2×10^{-4}, weight decay of 10^{-2}, and a budget of 100,000 optimizer steps. These settings are distinct from the larger pretraining batch in Table 6.

B Analysis on Pretraining Dataset

B.1 Pretraining Dataset Statistics

We pretrain AW-0 on the union of two real-world manipulation datasets: YAM and Egocentric Human data. The YAM corpus contains 73,101 episodes across 279 tasks, totaling 1,919.3 hours of bimanual robot teleoperation. Demonstrations are recorded at 30 Hz with left-wrist, right-wrist, and top-view RGB cameras. The egocentric human corpus contains 175,541 episodes across 327 tasks, totaling 993.9 hours, with a single head-mounted RGB camera and tracked hand motion retargeted to 22-DoF dexterous-hand joint targets per hand. Together, the two sources comprise 248,642 episodes and 2,913.2 hours. The human camera view is replicated across the three input slots; each training window is paired with its episode-level instruction and aligned program slice. Table 7 reports detailed statistics.

Table 7: **Pretraining Corpus Statistics.**

	YAM	Egocentric Human	Total
Episodes	73,101	175,541	248,642
Durations (h)	1,919.30	993.91	2,913.22
Tasks	279	327	606
Total subtask slices	986,314	1,089,059	2,075,373
Mean subtask slices per episode	13.5	6.2	8.3

B.2 Skill Library Definitions

A skill is a reusable motion that the model can invoke by name. Each skill is a Python function written over the primitive API described in Appendix J.1, taking the objects it acts on as arguments, so that a subtask recurring across the corpus becomes a single call rather than being spelled out primitive by primitive. Following the annotation pipeline of Section 2.3, we generate a skill library for each dataset: 23 skills for YAM and 27 skills for the egocentric human data. Tables 8 and 9 list them, each with a one-line description of the motion it implements.

Table 8: **The Frozen YAM Skill Library** (23 skills).

Skill	Definition
`adjust_in_place`	Grip an object where it lies, slide it a few centimetres without lifting, and let go.
`alternating_smoothing_pats`	Press the fabric flat with one arm and come off, then the other — the alternating pats of a smoothing slice.
`approach_best_grasp`	Sample grasp candidates on an object, pick the best one reachable by this arm, and drive the end-effector onto it.
`brace_and_pull_open`	Brace the body with one arm, take its lid, flap or handle with the other, draw it open, then release the brace.
`brace_and_push_in`	Grip the same part twice, low to keep it aligned and high to push, and drive it home into its receptacle.
`flip_object_over`	Lift a held flat object clear, roll the wrist a half turn, release it inverted, then grasp it again to pin it down.
`fold_edge_over`	Both arms pinch a cloth edge, carry it across the item, and release — one fold.
`free_hand_and_pick`	Open the idle arm, then pick the object up with the working arm.
`hand_over_hand_extract`	Two grip–pull–release bites that walk a long object out of a bag, sleeve or slot.
`lower_onto_target`	Hover the carried object above a detected target at the current wrist orientation and lower it in; the caller releases.
`pull_part_free`	Grip a part where it sits, drag it straight off along one bounded delta, and release.
`push_closure_shut`	Grip a drawer, lid, door or cover and drive it along its travel until shut, then let go.
`ratchet_turn_fastener`	Two ratchet strokes on a threaded fastener; the release in between is the wrist unwinding, not a drop.
`ratchet_twist_off`	Two ratchet bites on a threaded part; the sign of the angle sets whether it tightens or loosens.
`relocate_object`	One-arm transport inside a single slice: pick the object up and place it at the named target.
`roll_up_one_bite`	One reach-limited bite of rolling: pinch, push the roll forward, release, and travel back to the next pinch point.
`run_zipper_along_seam`	Pin the case with one arm, take the zipper pull with the other, and draw it one run along the seam.
`scrub_back_and_forth`	Bring an already-held cloth to a detected surface and scrub it once back and forth.
`spread_taut_and_release`	With both arms already gripping, drag them apart to pull the item taut, then release both.
`stir_braced_vessel`	Stir inside a braced vessel, re-seating the brace once the tool tip starts loading the wall.

Skill	Definition
tear_off_piece	Pin the parent sheet, roll or bag with one arm, pinch the piece with the other, and pull along the tear line.
tilt_pour_over_target	Carry a held vessel over a detected receptacle, clear the rim, tip once, and return to upright.
wrap_tension_pull	Two hand-over-hand bites on a strand being wound: regrasp further along and pull the slack taut, twice.

Table 9: **The Frozen Egocentric Human Skill Library** (27 skills).

Skill	Definition
align_and_seat_braced	One hand pins the target so it cannot shift while the other pushes the part home and lets go.
brace_and_pick	Hold the source still with one hand and pick the object off or out of it with the other.
brace_and_take_tool	Set up the two-handed working stance: the brace hand closes on the workpiece, the working hand on the tool.
close_closure	Swing a closure shut with both hands on it and come off it; nothing is held afterwards.
flatten_by_pulling_taut	Pin a slack flat item with one hand, drag it taut with the other, re-bite further along, and drag again.
flip_over_between_hands	Turn an object face-over by passing it between the hands through a half turn.
fold_over_and_crease	Brace a flat item, carry its free edge across in two steps, set it down, and press the crease — one fold.
hang_item_on_support	Bring a held item over a hook, rack or branch from above and let it down onto it.
insert_into_receptacle	Seat the held object into a receptacle, let go, and withdraw the now-empty hand.
move_item_two_handed	Take a bulky item with both hands where it sits, carry it along a semantic direction, and release with both.
open_closure	Swing a hinged closure open with one hand while the other braces the container body, then come off it.
pick_and_bring	Pick an object up and carry it to a semantic relation on where it is going next; the hand keeps hold of it.
place_at_target	Set the carried object down at a target with the working hand and free the second hand as well.
pour_into_target	Bring a held vessel's mouth over a target, tip it until it empties, and return it upright; the grip never changes.
press_onto_target	Press a target with both hands and come off it — seating a lid or case home, or working a control.
pull_open_and_hold	Draw a handle, leaf or free end open along a semantic direction and keep hold of it.
pull_part_free	Brace an assembly with one hand and pull a part off or out of it with the other.
rotate_grasped_object	Take an object where it sits and turn it about a semantic axis; the hand ends still on the object.
scoop_and_carry	Take a load out of a source container with a held utensil and bring it over the destination.
scrub_inside_vessel	One hand holds the vessel still while the other puts a brush down inside it and works a stroke pair.
stir_contents	Drop the already-held tool inside the vessel and sweep it round.
thread_through_and_pull_out	Push a lace, cord or pin through a hole with one hand and catch the emerging end with the other to draw the slack through.
tighten_with_ratchet_regrasps	Take up the last of a thread on an already-seated cap: turn, release the bite, re-bite, and turn again.
turn_over_object	Flip a held object face-over and bring the new face to a semantic relation on its destination.
unscrew_cap	Twist a threaded part, let the wrist unwind and re-close, twist again, and lift the freed part clear.

Skill	Definition
`wipe_surface_back_and_forth`	Park a cloth or sponge already in one hand on an open surface and run two out-and-back strokes.
`wrap_around_target`	Lead a held wrapper — ribbon, foil, film, gauze — once around a target in a closed circuit.

B.3 Visualizations of YAM Skills

We visualize four of the 23 YAM skills. For each, we select one representative slice and show a strip of key frames, the corresponding task and subgoal annotations, the program slice, and the skill function. In the listings, calls in **bold** are library skills; all others are primitive calls.

Skill I: `fold_edge_over`
Task: "wrap blanket roll with ribbon"
Sub-goal: "Fold the blanket at the center of the board"



Figure 8: Key frames of Skill I, `fold_edge_over`.

Skill definition:

```
def fold_edge_over(robot, item_name, left_delta, right_delta):
    # pinch the edge with both arms
    robot.grasp(item_name, 'left')
    robot.grasp(item_name, 'right')
    # carry the pinched edge across the item
    robot.servo_ee_delta('left', left_delta)
    robot.servo_ee_delta('right', right_delta)
    # let the fold lie
    robot.release('left')
    robot.release('right')
```

Program slice for this episode:

```
fold_edge_over(robot, 'blanket', [0.15, 0.0, 0.0], [0.15, 0.0, 0.0])
```

Skill II: `alternating_smoothing_pats`
Task: "roll towels"
Sub-goal: "Flatten the towel"

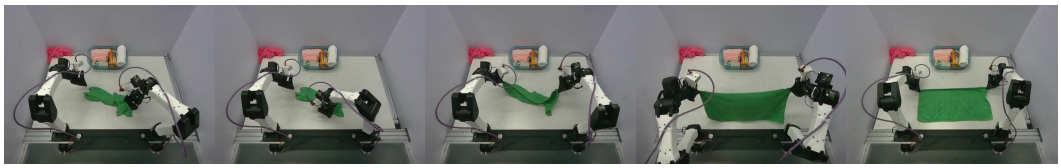


Figure 9: Key frames of Skill II, `alternating_smoothing_pats`.

Skill definition:

```
def alternating_smoothing_pats(robot, fabric_name, first_side, second_side):
    # pat 1 -- press the fabric flat, then come off
    robot.grasp(fabric_name, first_side)
    robot.release(first_side)
    # pat 2 -- the other hand, on the other half of the sheet
    robot.grasp(fabric_name, second_side)
    robot.release(second_side)
```

Program slice for this episode:

```
# Smooth out the towel with alternating pats
alternating_smoothing_pats(robot, 'towel', 'left', 'right')
```

Skill III: `flip_object_over`
Task: "fold napkin into a case and place utensils in it"
Sub-goal: "Turn over the spoon at the center of the board"



Figure 10: Key frames of Skill III, `flip_object_over`.

Skill definition:

```
def flip_object_over(robot, object_name, side, lift_delta, turn_deg=180.0):
    # lift just clear so the object can swing
    robot.servo_ee_delta(side, lift_delta)
    # roll the wrist -- this is the flip
    robot.rotate(object_name, turn_deg, side)
    # let it fall the last centimetre, landing inverted
    robot.release(side)
    # pin the flipped object down again for the next slice
    robot.grasp(object_name, side)
```

Program slice for this episode:

```
# Lift, roll the wrist to flip the spoon, drop it, and pin it
flip_object_over(robot, 'spoon', 'left', [0.0, 0.0, 0.1])
```

Skill IV: `adjust_in_place`
Task: "place bookmark into book"
Sub-goal: "Adjust the bookmark in the book"

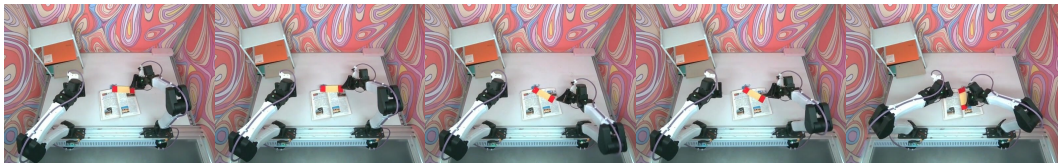


Figure 11: Key frames of Skill IV, `adjust_in_place`.

Skill definition:

```
def adjust_in_place(robot, object_name, side, nudge_delta):
    # take hold where it sits -- no lift
    robot.grasp(object_name, side)
    # slide it by a small bounded amount
    robot.servo_ee_delta(side, nudge_delta)
    # let it settle
    robot.release(side)
```

Program slice for this episode:

```
adjust_in_place(robot, 'bookmark', 'right', [0.05, 0.0, 0.0])
```

B.4 Visualizations of Egocentric Human Skills

We visualize six of the 27 egocentric human skills. For each, we select one representative slice and show a strip of key frames, the corresponding task and sub-goal annotations, the program slice, and the skill function. In the listings, calls in **bold** are library skills; all others are primitive calls.

Skill I: `open_closure`
Task: "assemble photo album"
Sub-goal: "Open the photo album"



Figure 12: Key frames of Skill I, `open_closure`.

Skill definition:

```
def open_closure(robot, container_name, closure_name, side, brace_side, axis, angle_deg):
    # Hold the container body down so the closure, not the container, moves.
    robot.grasp(container_name, brace_side)
    # Take the closure where it sits.
    robot.grasp(closure_name, side)
    # Swing it open about its hinge.
    robot.rotate(closure_name, axis, angle_deg, side)
    # Come off the closure; it stays open.
    robot.release(side)
```

Program slice for this episode:

```
open_closure(robot, 'photo album', 'photo album cover', 'left', 'right', 'object_long',
    180.0)
```

Skill II: `pour_into_target`
Task: "serve rice into a bowl"
Sub-goal: "Pour a spoonful of rice into the bowl"



Figure 13: Key frames of Skill II, `pour_into_target`.

Skill definition:

```
def pour_into_target(robot, vessel_name, target_name, side, axis, angle_deg):
    # Bring the held vessel's mouth over the receiving target.
    robot.goto_relative(side, target_name, 'above')
    # Tip it until it pours.
    robot.rotate(vessel_name, axis, angle_deg, side)
    # Bring it back upright; the pour ends.
    robot.rotate(vessel_name, axis, -angle_deg, side)
```

Program slice for this episode:

```
pour_into_target(robot, 'spoon', 'bowl', 'right', 'object_long', 110.0)
```

Skill III: `stir_contents`
Task: "mixing solutions with stirring rod"
Sub-goal: "Stir the solution in the beaker"

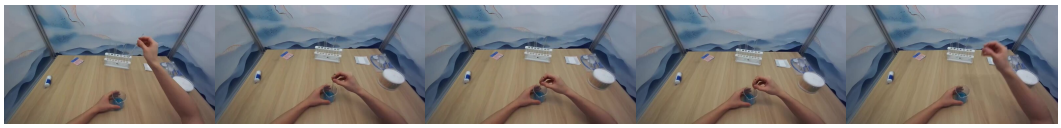


Figure 14: Key frames of Skill III, `stir_contents`.

Skill definition:

```
def stir_contents(robot, tool_name, vessel_name, side):
    # Bring the held tool down inside the vessel, among the contents.
    robot.goto_relative(side, vessel_name, 'inside')
    # Sweep it round the vessel; the grip never changes, nothing is let go.
    robot.rotate(tool_name, 'vertical', 360.0, side)
    robot.rotate(tool_name, 'vertical', 360.0, side)
```

Program slice for this episode:

```
brace_and_take_tool(robot, 'beaker', 'left', 'stirring rod', 'right')
stir_contents(robot, 'stirring rod', 'beaker', 'right')
```

Skill IV: `wipe_surface_back_and_forth`
Task: "use lint roller on pants"
Sub-goal: "Remove the lint from the pair of pants"

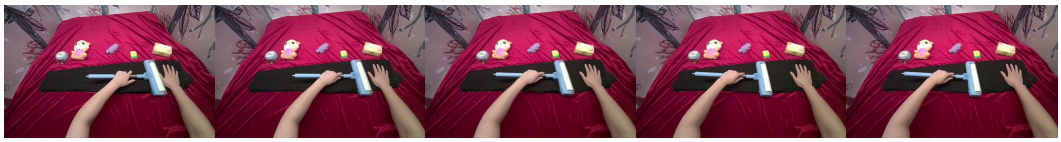


Figure 15: Key frames of Skill IV, `wipe_surface_back_and_forth`.

Skill definition:

```
def wipe_surface_back_and_forth(robot, surface_name, relation, stroke_direction,
     return_direction, side):
    # Park the cloth on the surface.
    robot.goto_relative(side, surface_name, relation)
    # Out and back.
    robot.move_arm(side, stroke_direction)
    robot.move_arm(side, return_direction)
    # And again.
    robot.move_arm(side, stroke_direction)
    robot.move_arm(side, return_direction)
```

Program slice for this episode:

```
# Roll the lint roller back and forth over the pants (4 pairs of strokes)
for _ in range(2):
    wipe_surface_back_and_forth(robot, 'pants', 'against', 'right', 'left', 'left')
```

Skill V: `fold_over_and_crease`
Task: "fold long pants"
Sub-goal: "Fold a pair of pants"



Figure 16: Key frames of Skill V, `fold_over_and_crease`.

Skill definition:

```
def fold_over_and_crease(robot, item_name, side, brace_side, direction, fold_direction):
    # Hold the item down so the fold, not the whole item, travels.
    robot.grasp(item_name, brace_side)
    # Take the free edge with the working hand.
    robot.grasp(item_name, side)
    # Carry the edge across the item ...
    robot.move_arm(side, direction)
    # ... and land it on the far side.
    robot.move_arm(side, fold_direction)
    # Let the folded edge down.
    robot.release(side)
    # Press the crease flat.
    robot.press(item_name, side)
```

Program slice for this episode:

```
fold_over_and_crease(robot, 'pants', 'left', 'right', 'up', 'right')
```

Skill VI: `wrap_around_target`
Task: "use a curling iron to curl hair"
Sub-goal: "Wrap a section of hair around the curling iron"

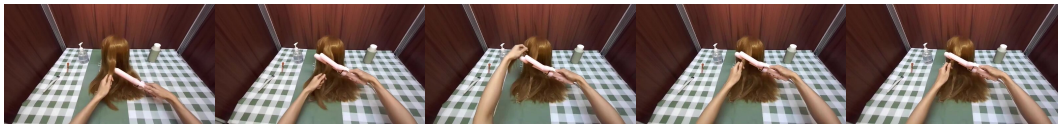


Figure 17: Key frames of Skill VI, `wrap_around_target`.

Skill definition:

```
def wrap_around_target(robot, target_name, side):
    # Start over the top of the target.
    robot.goto_relative(side, target_name, 'above')
    # Over the far side.
    robot.move_arm(side, 'away_from_body')
    # Down its far face.
    robot.move_arm(side, 'down')
    # Back underneath, toward the body.
    robot.move_arm(side, 'toward_body')
    # And up to close the circuit.
    robot.move_arm(side, 'up')
```

Program slice for this episode:

```
# Wrap the hair around the curling iron twice.
for _ in range(2):
    wrap_around_target(robot, 'curling iron', 'left')
```

B.5 Skill Distribution in the Embedding Space

In Figure 18 and 19, we visualize the skill distribution of variant skills in the pretrained code expert's embedding space. Each point represents one episode slice, placed by a two-dimensional t-SNE projection of its embedding and colored by the skill assigned to it. Dashed outlines mark the region covered by each skill, and the frames along the edges show a representative short span from an

episode for selected skills. Figure 18 covers the 18 most common skills in the YAM library and Figure 19 the 16 most common skills in the egocentric human library. In both, slices assigned to different skills occupy largely separate regions, suggesting that the skill label is predictive of the motion in a slice and can serve as a conditioning signal for action generation.

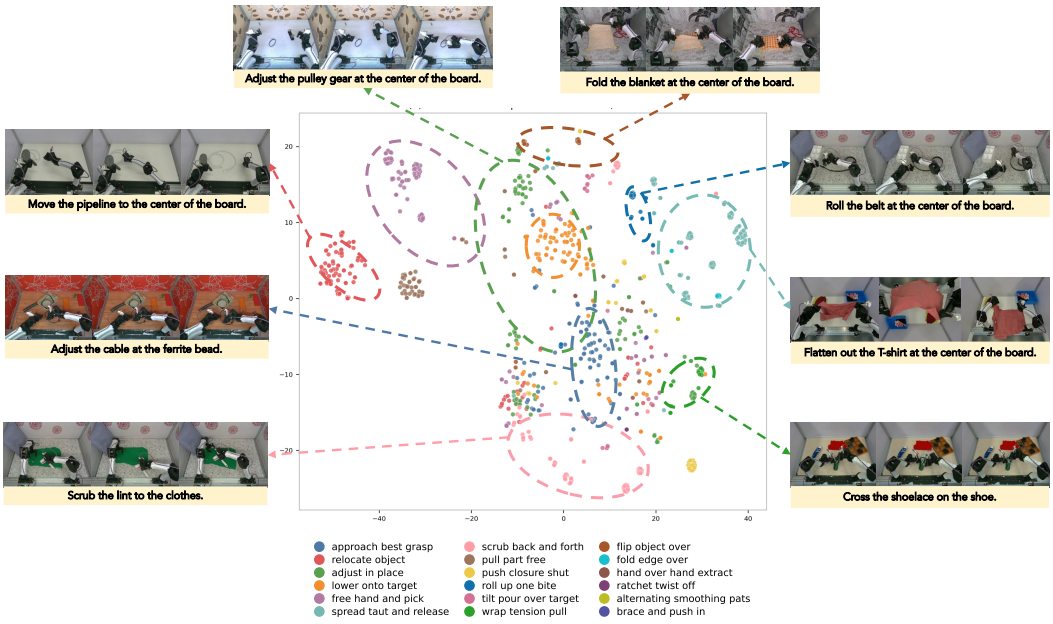


Figure 18: **Skills of the YAM Skill Library in the Embedding Space.**

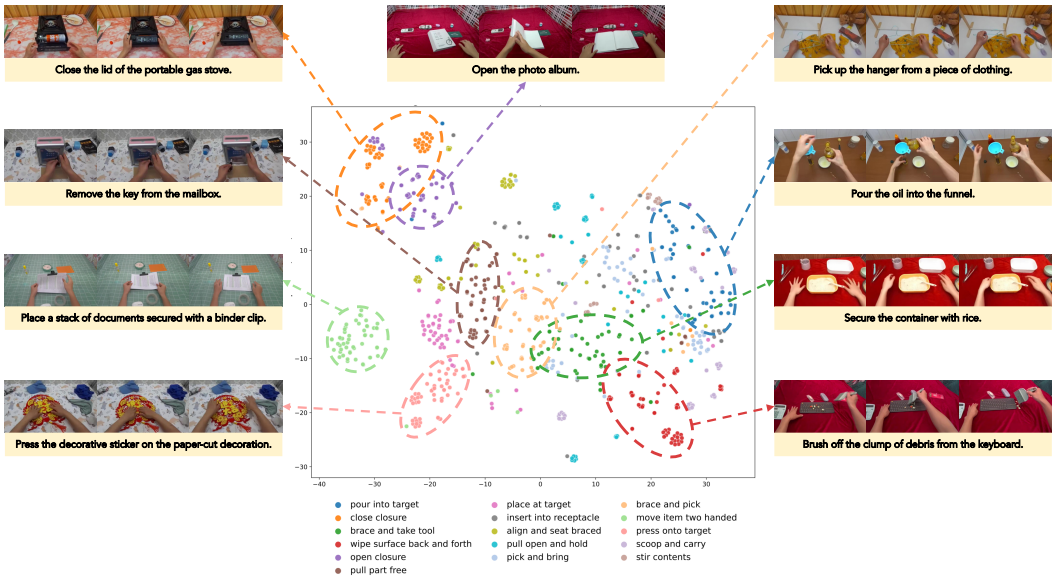


Figure 19: **Skills of the Egocentric Human Skill Library in the Embedding Space.**

C BASELINE IMPLEMENTATION DETAILS

In Sections 3.1 and 3.2, we compare AW-0 with four baselines: $\pi_{0.5}$, DreamZero, EgoScale, and WLA. Here, we describe how each is reproduced and adapted. On RoboTwin 2.0, we train a single multi-task policy per baseline on 2,500 `demo_clean` demonstrations (50 tasks $\times$ 50 demonstrations). For each of the 8 real-world tasks, we train a separate policy per baseline using task-specific demonstrations sampled at 30 Hz. Every baseline receives a language instruction, the robot state, and three RGB views (one head and two wrist cameras). All baselines follow the same evaluation protocol as AW-0. All real-world rollouts are scored using the task-specific rubrics in Appendix E.

$\pi_{0.5}$ (Black et al., 2025) is a vision-language-action model that combines a pretrained vision-language backbone with a flow-matching action expert. We follow the official OpenPI implementation[1] and initialize all policies from the released $\pi_{0.5}$ pretrained checkpoint. For each real-world task, we fine-tune a separate policy for 50 epochs on the task-specific demonstrations. We use 14-dimensional joint-space actions on RoboTwin and YAM, and 62-dimensional actions on Tianji–Sharpa consisting of a 9-dimensional end-effector pose and 22 hand joints per side. Arm actions are relative to the current state, while gripper and hand actions are absolute targets. We reinitialize the action input and output projections to accommodate these dimensions and fine-tune all model parameters. Following the official implementation, we use an action horizon of 50 for RoboTwin and 32 for the real robots, together with quantile normalization and standard image augmentation. All models are trained with AdamW on eight H100 GPUs using FSDP and a global batch size of 256. We use a peak learning rate of 5×10^{-5} with cosine decay to 5×10^{-6} and 2,000 warmup steps.

DreamZero (Ye et al., 2026) is a world-action model that jointly generates future video and actions using a pretrained video diffusion backbone. We follow the official implementation[2] and initialize from the DreamZero-AgiBot checkpoint. We reproduce DreamZero on RoboTwin and all 8 real-world tasks, training a separate policy for one epoch on each real-world task. We fully fine-tune the diffusion transformer and state/action modules, while keeping the text encoder, image encoder, and video VAE frozen. Following the original design, the three camera views are tiled into one frame, and each training sample contains up to four action chunks of 24 steps. Actions are predicted relative to the state at the beginning of each chunk, with absolute hand-joint targets on Tianji–Sharpa. The state and action projections are adapted to the target embodiment, and the video and action losses are weighted equally. All policies are trained with AdamW on 32 H100 GPUs using a global batch size of 32. We use the same training hyperparameters across all tasks, with a peak learning rate of 2×10^{-5} and 5% linear warmup followed by cosine decay.

EgoScale (Zheng et al., 2026) studies how pretraining on egocentric human demonstrations scales to dexterous robot manipulation. We reproduce this baseline using the GR00T N1.7 implementation[3] and initialize from the pretrained `nvidia/GR00T-N1.7-3B` checkpoint. For each real-world task, we fine-tune a separate policy for 50 epochs while keeping the vision-language backbone frozen and training the projection layers and diffusion action head. We use relative joint actions for the arms on RoboTwin and YAM, relative end-effector actions on Tianji–Sharpa, and absolute targets for the grippers and 22-DoF dexterous hands. Following the official fine-tuning recipe, we apply state dropout with probability 0.2, random image cropping, and color jitter. We use an action horizon of 20 steps and train with AdamW on eight H100 GPUs with a global batch size of 640. The peak learning rate is 10^{-4}, with 5% linear warmup followed by cosine decay and a weight decay of 10^{-5}.

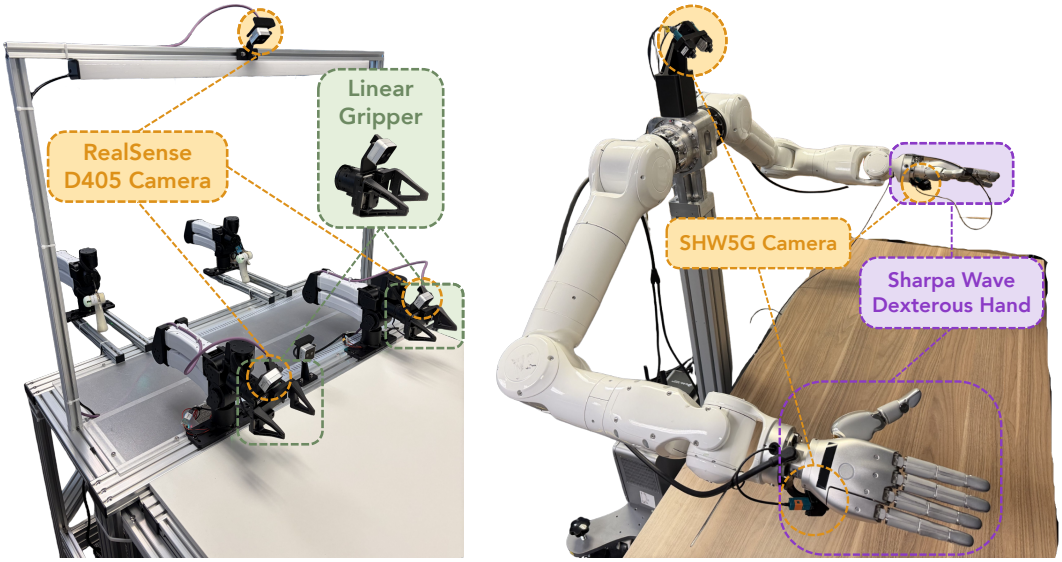
WLA (Yang et al., 2026) combines a vision-language backbone, a world expert, and an action expert to jointly model visual observations, language, and robot actions. We follow the official implementation[4] and initialize the backbone and world expert from pretrained RynnBrain-2B and Sana-600M weights, respectively. The connector and action expert are randomly initialized. We train a separate policy for 50 epochs on each real-world task, updating all modules except the image VAE. Following the original design, the policy observes the current head and wrist views together with the head view from 8 steps earlier. We use 64 meta-queries and a 32-step action chunk, with

[1] `https://github.com/Physical-Intelligence/openpi`
[2] `https://github.com/dreamzero0/dreamzero`
[3] `https://github.com/NVIDIA/Isaac-GR00T`
[4] `https://github.com/SJTU-DENG-Lab/WLA`

absolute action targets normalized to $[-1, 1]$. The world expert predicts the head-camera image at the end of the chunk, and its flow-matching loss is weighted by 0.1 relative to the action loss. All models are trained with AdamW on eight H100 GPUs using a global batch size of 256. We use a peak learning rate of 5×10^{-5} with cosine decay to 5×10^{-6} and 1,000 warmup steps.

D Real-world Experimental Setup

We conduct data collection and policy evaluation on two robot embodiments: the YAM station with parallel grippers and the Tianji–Sharpa station with dexterous hands. This section describes the robot hardware, visual observations, and teleoperation interfaces used in our experiments. Figure 20 gives an overview of the two stations.



(a) YAM Station (b) Tianji–Sharpa Station

Figure 20: **Real-world Robot Setups.** (a) The YAM station: two arms with linear grippers, with a RealSense D405 camera overhead and one on each wrist. The bottom camera is not used. (b) The Tianji–Sharpa station: a Tianji torso with two Sharpa Wave dexterous hands, observed by an SHW5G camera on the head and one on each wrist.

Robot Hardware and Control. The YAM station is a fixed-base bimanual system with two 6-DoF arms, each equipped with a parallel gripper. The Tianji–Sharpa station consists of a Tianji humanoid torso with two arms and two 22-DoF Sharpa dexterous hands. For AW-0, both embodiments use the 64-dimensional state–action representation described in Appendix A. The policy predicts relative end-effector poses together with absolute gripper targets on YAM or absolute finger-joint targets on Tianji–Sharpa.

Perception System. Both stations provide three RGB views: a head or overhead camera observes the tabletop workspace, and a camera at each wrist provides a local view of the gripper or hand and the manipulated objects. Each policy receives these images together with the task instruction and robot state. For AW-0, images are resized to 224×320 pixels, and the state contains the end-effector poses and the gripper positions or hand-joint angles. The same camera streams are provided to all baselines, with method-specific preprocessing described in Appendix C.

Teleoperation and Data Collection. We collect YAM demonstrations using the standard YAM station configuration, with two master arms and two puppet arms. The operator moves the master arms, whose measured joint angles are sent as joint-position targets to the corresponding puppet arms. The triggers on the teaching handles control the parallel gripper openings, yielding six arm-joint targets and one gripper target per side. For Tianji–Sharpa, we use Manus gloves with PICO motion trackers attached to the gloves, together with an Apple Vision Pro worn around the operator's neck. The tracked wrist positions and orientations define poses in $\mathrm{SE}(3)$, which are mapped to end-effector

targets for the two Tianji arms. In parallel, finger-motion measurements from the Manus gloves are retargeted to 22-dimensional joint-position targets for each Sharpa hand. At each teleoperation step, the arm-pose and hand-joint targets are passed to the corresponding robot controllers to coordinate arm and finger motion. During collection, we record the three RGB camera streams, measured robot states, and commanded action targets. We use their timestamps to pair visual observations with robot states and action commands, and associate each demonstration with a task instruction for post-training.

E REAL-WORLD TASK PROGRESS

We evaluate 8 real-world tasks that span two robot platforms. *Arrange Fruits*, *Pack Backpack*, *Replace Lightbulb*, and *Fold and Stack Towels* run on the YAM bimanual setup (two 6-DoF arms with parallel grippers); *Arrange Fruits with Tongs*, *Deal Poker*, *Flip Scoreboard*, and *Solve Rubik's Cube* run on the Tianji humanoid torso with two Sharpa dexterous hands. For each task we show the progress with the key frames of one training episode together with its code label; every color-coded block of the program is linked to the range of frames it explains. Each task is graded with an additive rubric that awards independent partial credit for each completed sub-step.

E.1 YAM TASKS

Task I: Arrange Fruits.

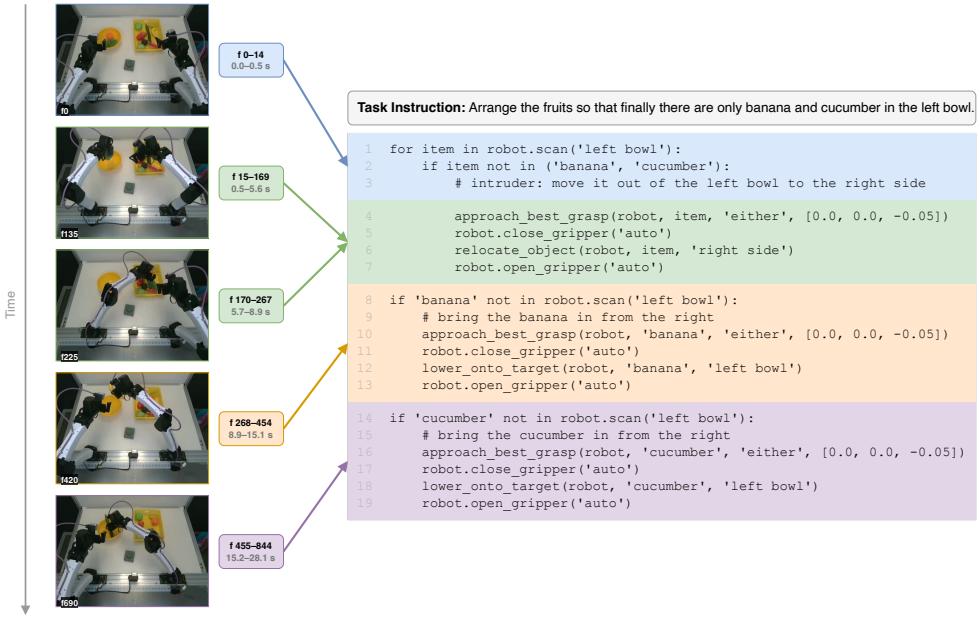


Figure 21: Key frames and code label of *Arrange Fruits*.

Grading rubric (additive):

- $+0.3$: **(a)** Moves each fruit not named in the instruction from the left bowl into the right basket (or none to remove).
- $+0.3$: **(b)** Moves each named fruit not yet in the left bowl from the right basket into the bowl (or none to fetch).
- $+0.4$: **(c)** Final state matches the instruction and no fruit is dropped.

Task II: Pack Backpack.

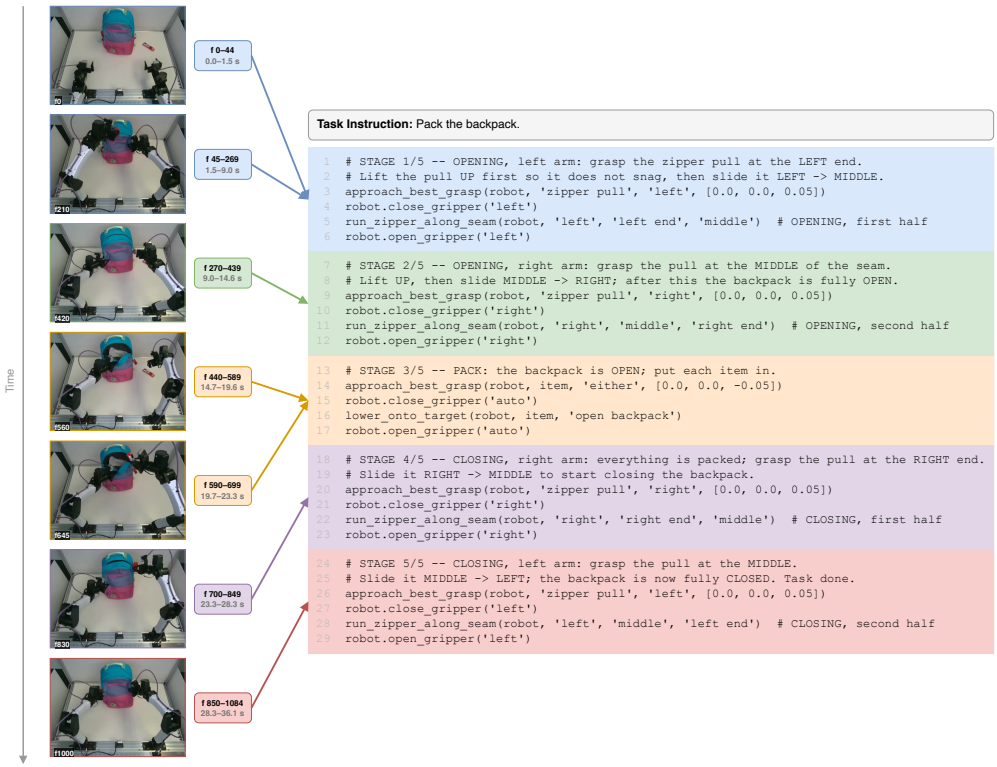


Figure 22: Key frames and code label of *Pack Backpack*.

Grading rubric (additive):

- $+0.15$: **(a)** Left gripper grasps the zipper pull at the left end and slides it to the middle.
- $+0.15$: **(b)** Right gripper re-grasps the pull at the middle and slides it to the right end.
- $+0.2$: **(c)** Open the backpack with the left gripper.
- $+0.2$: **(d)** Picks up the item from the table and places it inside the open backpack.
- $+0.15$: **(e)** Right gripper grasps the pull at the right end and slides it back to the middle.
- $+0.15$: **(f)** Left gripper slides the pull from the middle to the left end; the backpack is fully closed with the item inside.

Task III: Replace Lightbulb.

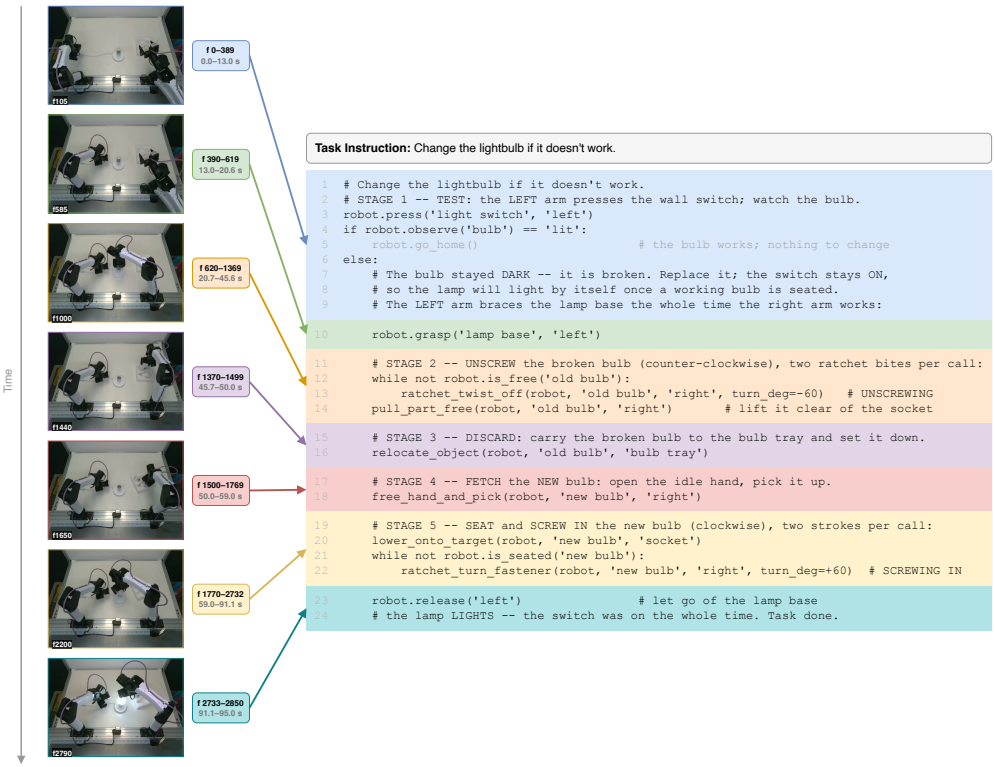


Figure 23: Key frames and code label of *Replace Lightbulb*.

Condition A – dead bulb, replacement required. Grading rubric (additive):

- $+0.1$: **(a)** Left gripper presses the inline switch and the bulb stays dark.
- $+0.1$: **(b)** Left gripper braces the lamp base.
- $+0.2$: **(c)** Right gripper unscrews the dead bulb and lifts it out of the socket.
- $+0.1$: **(d)** Dead bulb is set down in the bulb tray.
- $+0.1$: **(e)** Right gripper picks up the spare bulb.
- $+0.3$: **(f)** Spare bulb is seated in the socket and screwed in clockwise until lit.
- $+0.1$: **(g)** When the bulb is lit, release both arms to their home positions.

Condition B – working bulb, no replacement needed. Grading rubric (additive):

- $+0.3$: Left gripper presses the inline switch and the bulb lights up.
- $+0.7$: The lit bulb shows the lamp still works, so both arms move back to their initial pose and stay still until the episode ends.

Task IV: Fold and Stack Towels.

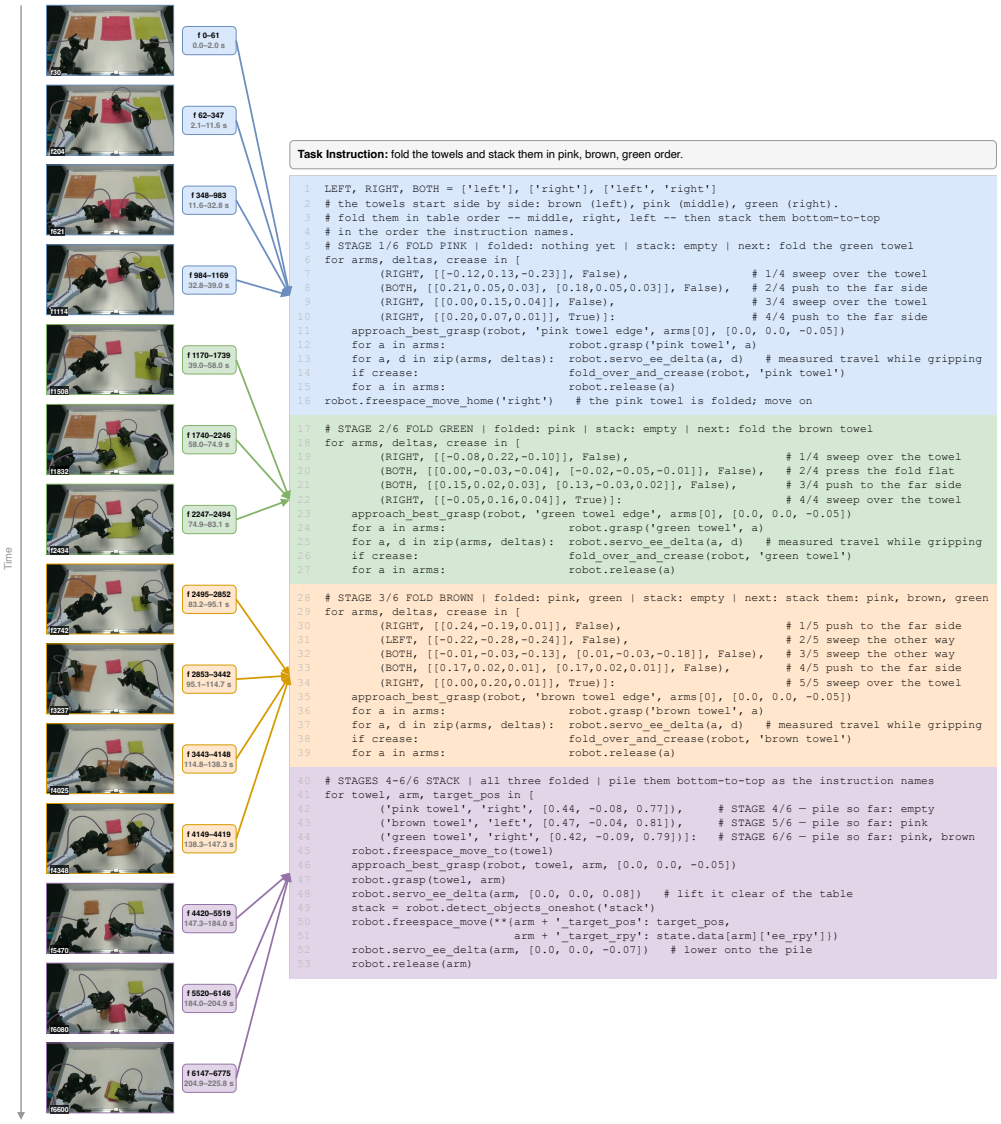


Figure 24: Key frames and code label of *Fold and Stack Towels*.

Grading rubric (additive):

- $+0.2$: **(a)** Grasps a towel edge and completes one fold.
- $+0.2$: **(b)** Each of the three towels is folded flat.
- $+0.1$: **(c)** Picks a folded towel up and lays it on the pile.
- $+0.2$: **(d)** All three folded towels end up in a single pile, none dropped or left behind.
- $+0.3$: **(e)** The pile reads, bottom to top, in the colour order the instruction names.

E.2 TIANJI–SHARPA TASKS

Task I: Arrange Fruits with Tongs.

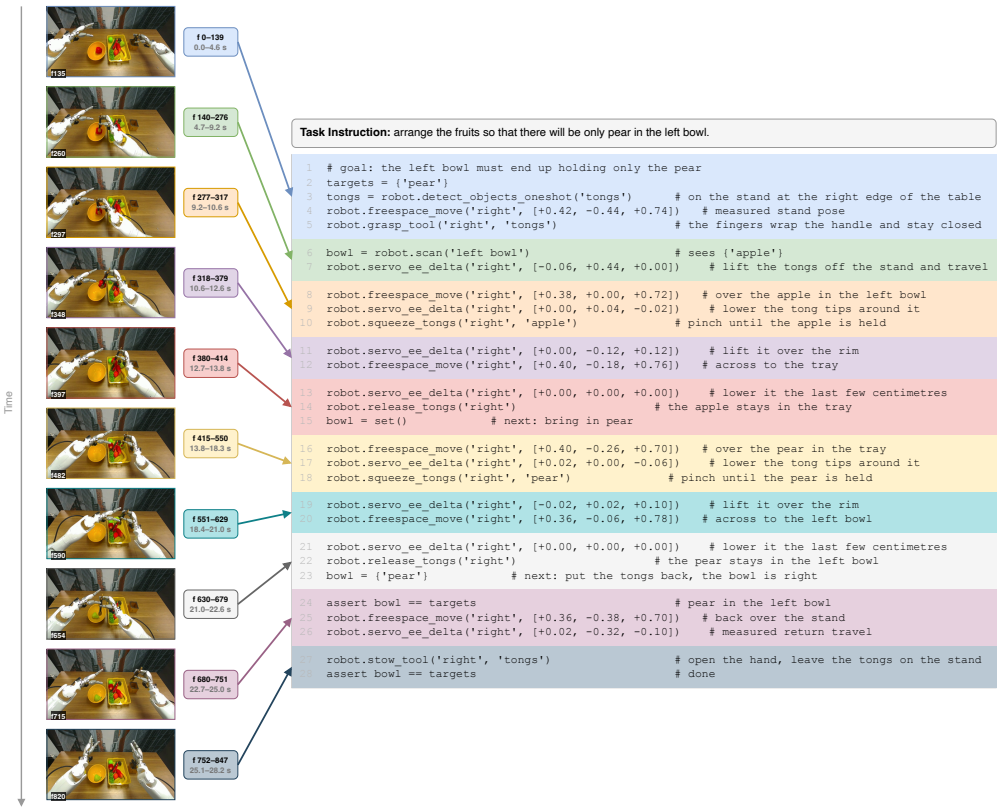


Figure 25: Key frames and code label of *Arrange Fruits with Tongs*.

Grading rubric (additive):

- $+0.2$: **(a)** Right hand takes the tongs off their stand and keeps hold of them.
- $+0.25$: **(b)** Moves each fruit not named in the instruction from the left bowl into the tray (or none to remove).
- $+0.25$: **(c)** Moves each named fruit not yet in the left bowl from the tray into the bowl (or none to fetch).
- $+0.2$: **(d)** Final state matches the instruction and no fruit is dropped.
- $+0.1$: **(e)** Tongs are put back on their stand.

Task II: Deal Poker.

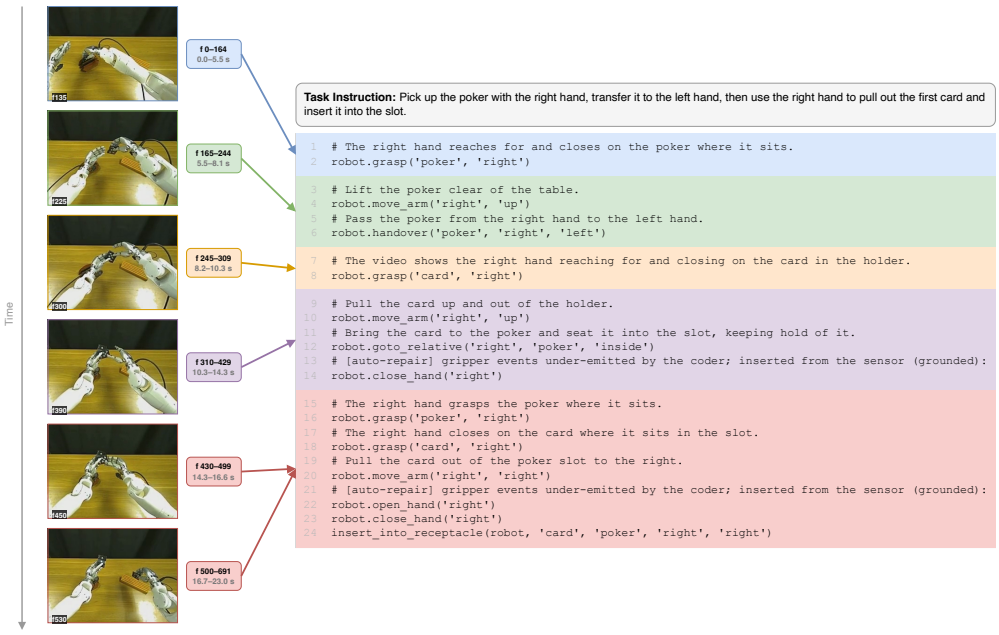


Figure 26: Key frames and code label of *Deal Poker*.

Grading rubric (additive):

- $+0.1$: **(a)** Right hand grasps the card stack from above.
- $+0.2$: **(b)** Right hand transfers the stack to the left hand (handover).
- $+0.3$: **(c)** Right thumb flicks and adjusts until exactly one card protrudes from the top.
- $+0.3$: **(d)** Right hand grasps the single protruding card.
- $+0.1$: **(e)** Right hand inserts the card vertically into the card slot.

Task III: Flip Scoreboard.

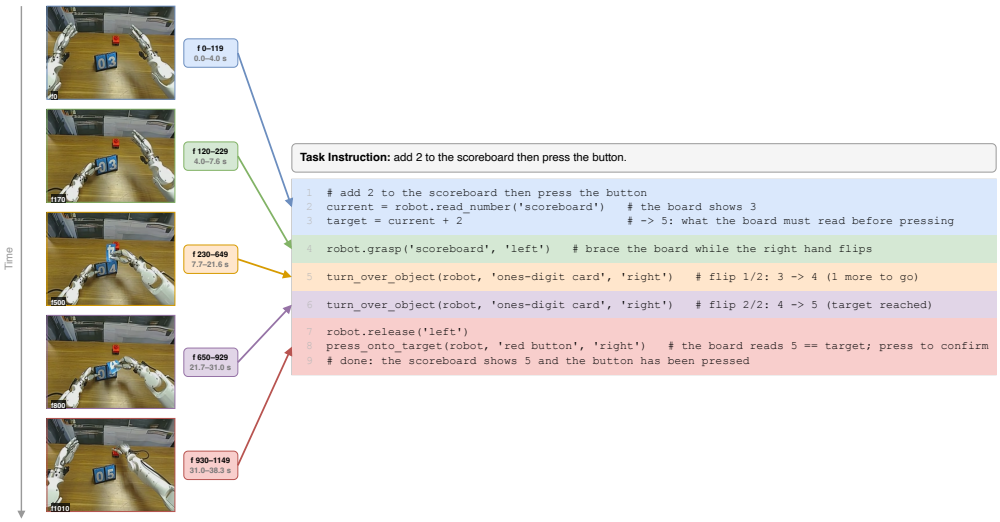


Figure 27: Key frames and code label of *Flip Scoreboard*.

Grading rubric (additive):

- $+0.1$: **(a)** Left hand holds the scoreboard steady.
- $+0.2$: **(b)** Right hand flips a digit card over (at least one complete flip).
- $+0.4$: **(c)** Flips exactly k times, so the board shows the initial score plus k.
- $+0.1$: **(d)** Left hand releases the board.
- $+0.2$: **(e)** Right hand presses the red button while the target score is showing.

Task IV: Solve Rubik's Cube.

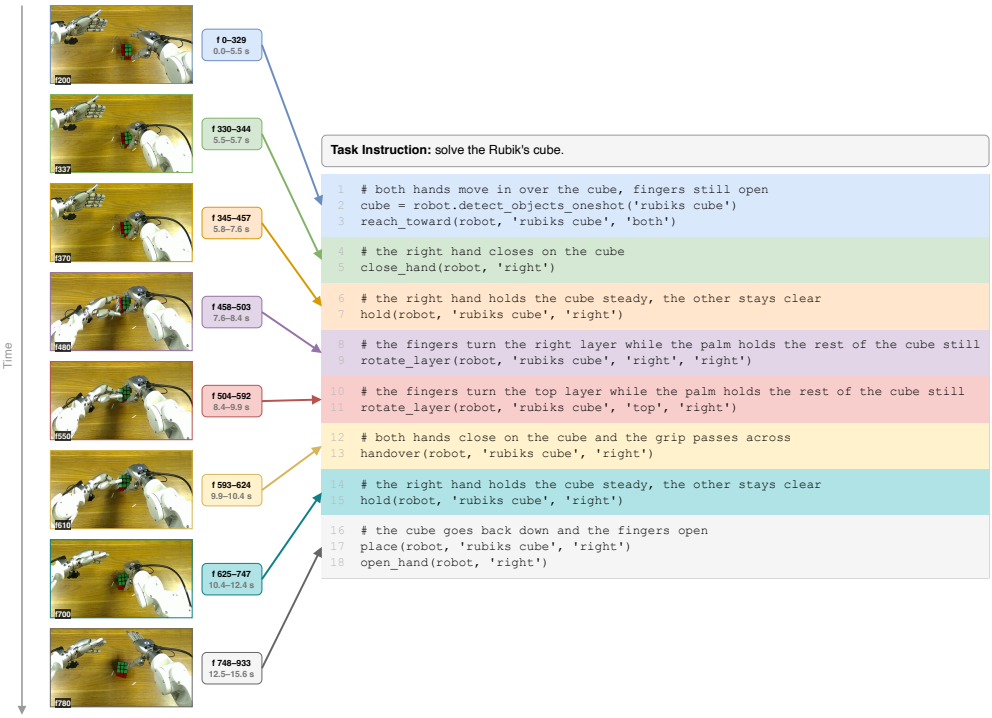


Figure 28: Key frames and code label of *Solve Rubik's Cube*.

Grading rubric (additive):

- $+0.2$: **(a)** Right hand grasps the cube and lifts it off the table.
- $+0.3$: **(b)** Fingers turn the scrambled layer a full quarter turn, leaving no layer half-turned.
- $+0.3$: **(c)** The cube is solved: every face shows a single colour.
- $+0.2$: **(d)** The cube is put back on the table and both hands let go.

F Analysis on Code Generation Quality

F.1 Pre-trained Code Generation Quality

We evaluate the pretrained code expert on validation demonstrations before task-specific fine-tuning, examining visual grounding, program structure, and agreement with the annotated programs. Table 10 summarizes the quantitative results, while Figures 29 and 30 illustrate generated programs for robot and egocentric human observations.

Table 10: **Validation Metrics for the Pretrained Code Expert.** TF denotes teacher-forced evaluation. The grounding gap is the absolute increase in code loss when visual observations are shuffled across validation samples.

TF loss	TF accuracy	Grounding gap	Parse rate	Entity F1	Primitive F1	Skill-call rate
0.535	0.897	1.774	0.875	0.534	0.365	0.292

Quantitative Assessment. The model achieves 89.7% teacher-forced token accuracy and an 87.5% parse rate under free generation. Shuffling visual observations increases the code loss by 1.774, indicating that code prediction depends on the visual context. Entity and primitive F1 are 0.534 and 0.365, respectively, measuring overlap of quoted strings and robot API names with the reference programs. These scores indicate partial semantic agreement despite the higher syntactic validity. In addition, 29.2% of generated programs contain skill-style helper calls, identified by calls that take `robot` as their first argument. Together, these results show that pretraining learns visually conditioned program generation and skill invocation, while precise semantic and argument prediction remain areas for improvement.

YAM Robot Demonstrations. Figure 29 shows a held-out YAM demonstration of *"Clean keyboard with a cloth"*. The generated program expresses a cloth handover followed by repeated scrubbing through `scrub_back_and_forth`, preserving the object arguments, arm assignment, and loop structure of the reference subtask. This illustrates how a reusable skill can encode a repeated motion compactly. The generated displacement argument nevertheless contains two extra entries, illustrating that an appropriate skill-level plan can still contain parameter errors that prevent reliable direct execution.

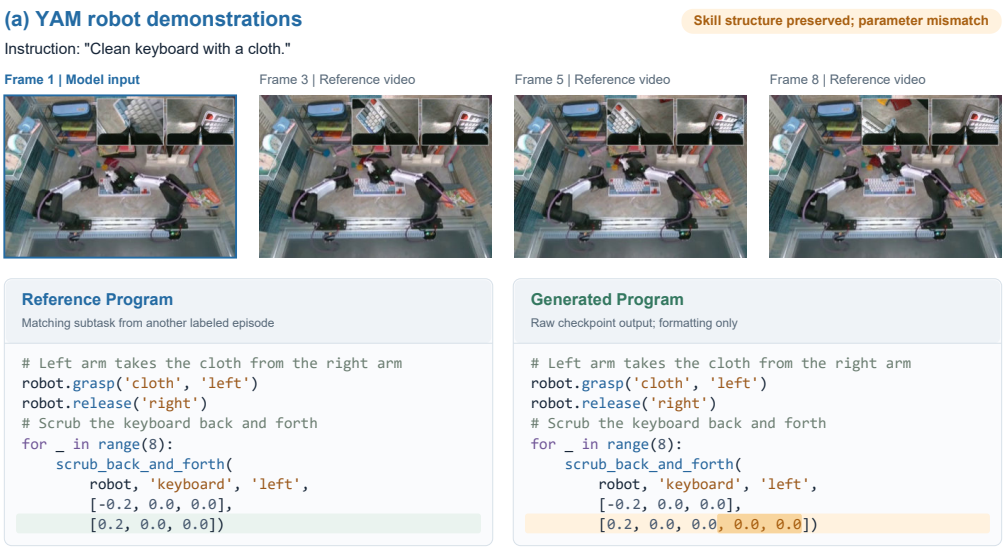


Figure 29: **Pretrained Code Generation on a YAM Demonstration.** The generated program is compared with a reference program from another labeled episode of the same task.

Egocentric Human Demonstrations. Figure 30 illustrates subtask-level planning under the instruction *"Squeeze toothpaste."* The generated program describes the preparatory subtask of removing the cap: stabilizing the tube, repeatedly twisting the cap, and lifting it away. It matches the annotated calls, arguments, and loop structure for this validation window. The two examples illustrate how the pretrained model represents manipulation through explicit object references, ordered calls, and repetition across robot and human observations, while the aggregate metrics distinguish this representational capability from uniformly correct or executable code.

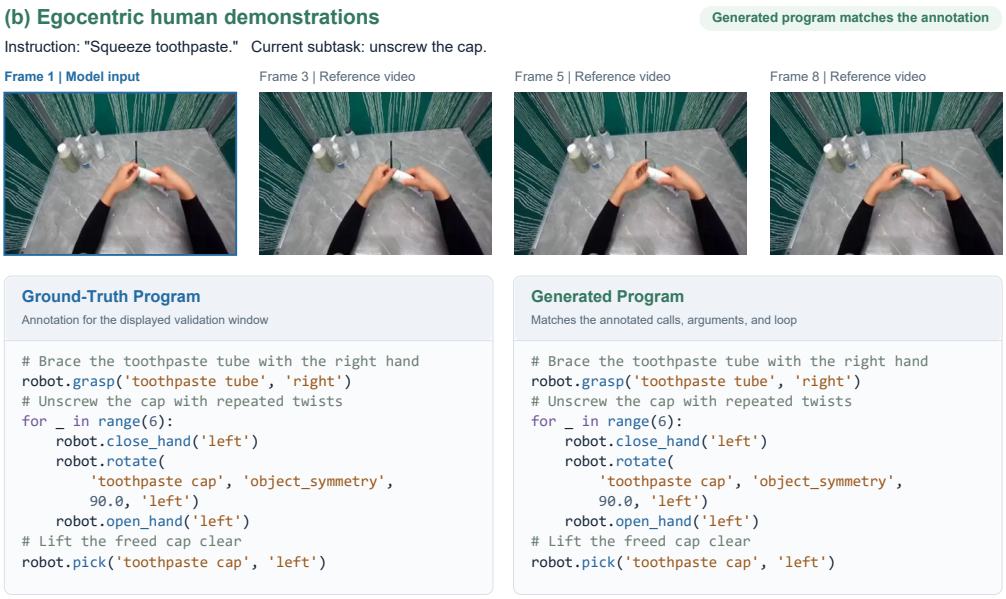


Figure 30: **Pretrained Code Generation on an Egocentric Human Demonstration.** Under the instruction *"Squeeze toothpaste,"* the model generates the cap-removal subtask, matching the annotated program for the held-out window.

F.2 Post-trained Code Generation Quality

We examine the post-trained code expert on YAM and Tianji–Sharpa demonstrations, focusing on subtask grounding and agreement with the annotated programs. Figures 31 and 32 illustrate code generated by the post-trained checkpoints for *Pack Backpack* and *Solve Rubik's Cube*, respectively during real-world evaluation. Each figure pairs the observation used for code generation with subsequent recorded frames and the annotated program for the same temporal window.

YAM Robot Demonstrations. Figure 31 shows *Pack Backpack* at the stage where the right arm grasps the zipper pull at the middle of the seam. The generated program invokes `approach_best_grasp` for the zipper pull, followed by `robot.close_gripper('right')`. It matches the annotated program exactly, preserving the stage comments, target object, arm assignment, and numerical arguments. This example illustrates how the post-trained model expresses a task-specific manipulation step through a reusable grasping skill and an explicit gripper command.

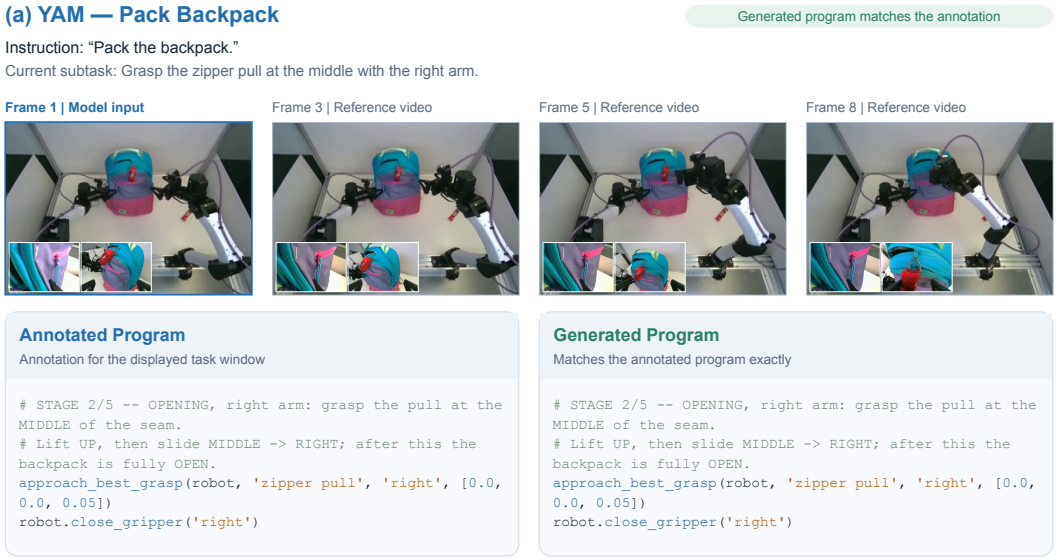


Figure 31: **Post-trained Code Generation on the Task *Pack Backpack*.**

Tianji–Sharpa Demonstrations. Figure 32 illustrates subtask-level planning for *Solve Rubik's Cube* while the right hand holds the cube steady. The generated program reproduces `hold(robot, 'rubiks cube', 'right')`, matching the annotated skill, object, and hand assignment. It nearly matches the full annotation, with the remaining differences confined to the descriptive comment: the word `cube` is omitted, and the comment continues on a new line without a comment marker. This formatting error prevents direct execution of the raw output despite the correct skill call. Together, the two examples illustrate task-specific semantic agreement across parallel grippers and dexterous hands, while distinguishing appropriate skill selection from uniformly valid executable code.

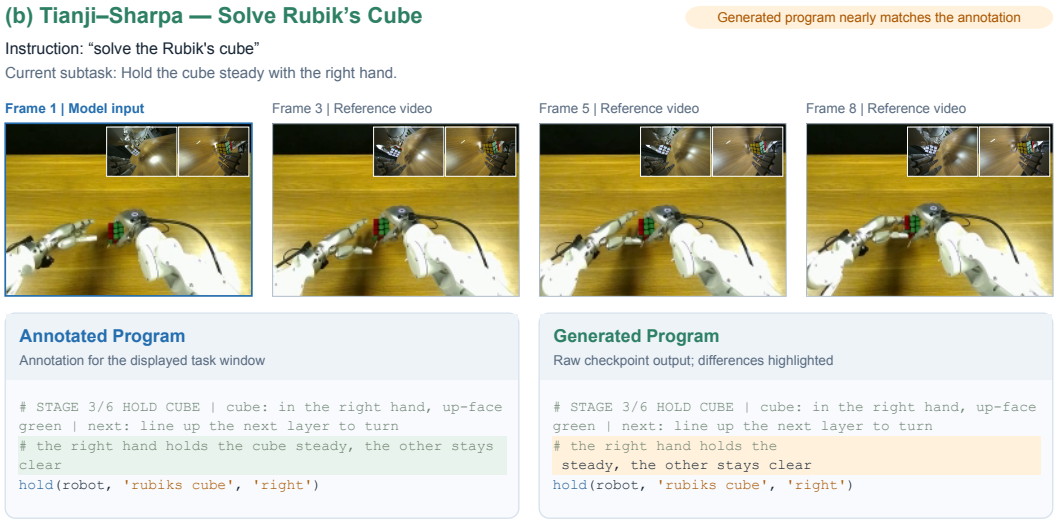


Figure 32: **Post-trained Code Generation on the Task *Solve Rubik's Cube*.**

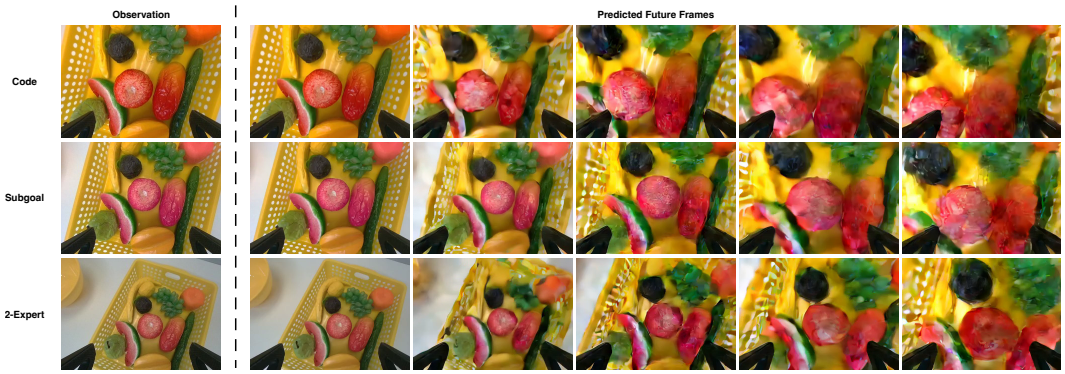


Figure 33: **Future-frame Prediction on Arrange Fruits for Three Models.**

G Additional Analysis on Real-world Results

G.1 Impact of Code Expert on Language Following Capability

We further examine whether code-based planning improves the alignment of predicted visual dynamics with task semantics. As shown in Figure 33, we compare the video generation quality of the right wrist camera of AW-0 with a natural-language subgoal variant (Subgoal) and a model without a planning stream (2-expert), under the instruction *"Arrange the fruits so that finally there is only mango in the left bowl."* This task requires identifying the intended object and grounding the desired outcome in an appropriate interaction. The three variants are evaluated under the same task and scene configuration, with similar but non-identical initial observations from separate rollouts.

The predicted futures reveal distinct differences in task understanding: AW-0 directs the interaction toward the mango, whereas Subgoal exhibits ambiguous targeting between the apple and mango, and 2-expert targets the apple. These observations suggest that structured skill calls and explicit object arguments provide a more specific planning condition for video prediction than a natural-language subgoal alone. By connecting instruction semantics to concrete interactions, the code stream helps align predicted dynamics with the intended task. Since the action expert also conditions on predictive video representations, this alignment provides a pathway for improving language following and action selection, complementing the execution results in our main experiments.

G.2 Impact of Code Expert on Skill Robustness

We further examine whether code-based planning improves the prediction of physical dynamics and skill robustness during precise manipulation. As shown in Figure 34, we compare the future frames predicted by AW-0, Subgoal, and 2-expert for the right wrist camera on *Deal Poker*, focusing on the final stage in which the right hand inserts a card into a slot. This interaction requires aligning the held card with the narrow opening and guiding it into the slot, making the relative motion of the hand, card, and receptacle critical to an accurate prediction. The three variants receive similar but non-identical initial observations sampled at nearby time steps from the same episode.

The predicted futures reveal distinct differences in the insertion motion: AW-0 depicts the card aligning with and entering the slot more accurately, whereas Subgoal and 2-expert produce less coherent insertion sequences, with blur and distortions around the hand and manipulated objects. These observations suggest that explicit skill calls and object arguments help the video expert better predict how the hand and card should move relative to the receptacle. The implementation of the code expert supports more accurate video generation, and therefore provides the action expert with more reliable predictive guidance for precise manipulation.

G.3 Hybrid Actions Evaluation Details

Figure 35 is a qualitative example of hybrid evaluation on the *"Fold and Stack Towels"* task, showing a single episode that continues from the upper to the lower row. We observe that dexterous fold-

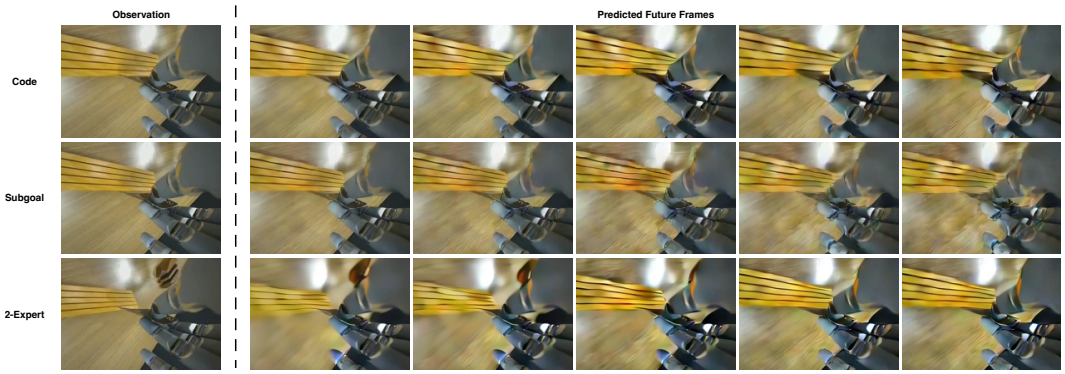


Figure 34: **Future-frame Prediction on Deal Poker for Three Models.**

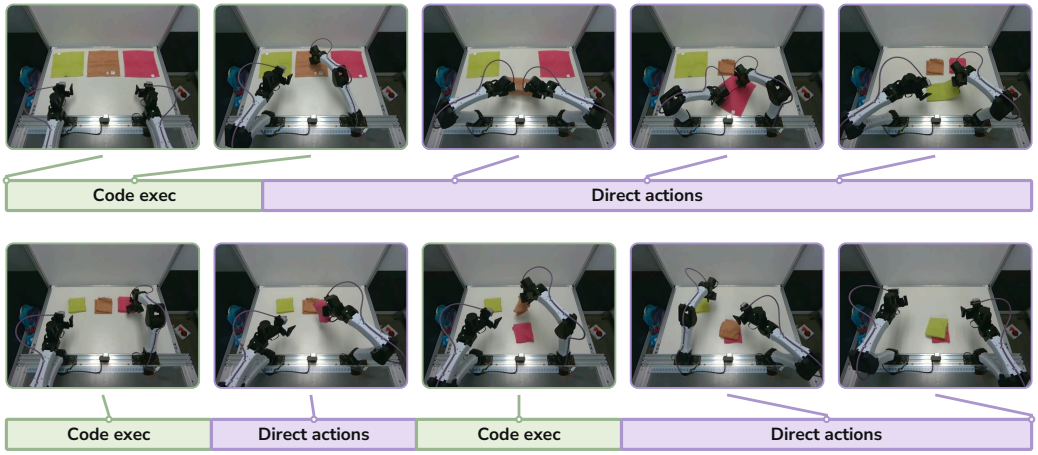


Figure 35: **Qualitative Example of Hybrid Actions for the *"Fold and Stack Towels"* Task.** A single episode continues from the upper to the lower row, with green and purple indicating code execution and direct actions, respectively.

ing motions rely primarily on direct actions from the action expert of AW-0, while code execution handles more structured motions, such as approaching and stacking the towels, which provide direct guidance on the stacking order and motion direction, improving the semantic reasoning capability for the whole system.

H Hybrid Actions Algorithm

Algorithms 1 and 2 outline the workflow of the hybrid mode. From an observation o and instruction ℓ, one forward pass of the policy π returns a direct action chunk $a_{1:H}$ over the replan horizon H and a code slice c. INIT builds, once per episode, the namespace $\mathcal{G}$ in which every slice executes: the environment's skill API, the program's helper definitions, and its setup block (episode-start perception). At each replanning point, ARBITRATE decides which of the two channels drives the robot. If the code slice is executed, the action chunk from the same forward pass is discarded; otherwise the chunk is played for H steps as in the default mode. Every replan is thus driven by exactly one channel.

A slice is executed only if it passes the following gates in order:

1. **Stage Gate**: its stage is not complete and not below a stage already attempted;
2. **Dependency Gate**: every variable it reads is defined in $\mathcal{G}$; and

Algorithm 1 Hybrid control: generated code first, direct actions otherwise

Require: policy π, environment E, frozen program P, instruction ℓ, replan horizon H
$\mathcal{G} \leftarrow \textsc{Init}(E, P)$ ▷ namespace: skill API, helpers, and the program's setup
$\mathcal{D} \leftarrow \emptyset,\ n_k \leftarrow 0\ \forall k$ ▷ stages completed by code; failed attempts per stage
while episode not terminated **do**
  $o \leftarrow E.\text{observe}()$
  $a_{1:H},\ c \leftarrow \pi(o, \ell)$ ▷ direct action chunk $a_{1:H}$ and code slice c for the next stage
  $\mathit{executed} \leftarrow \textsc{Arbitrate}(c,\ \mathcal{G},\ \mathcal{D},\ n)$ ▷ Algorithm 2: executes c on E if it passes every gate
  if not $\mathit{executed}$ **then**
    for $t = 1$ **to** H **do**
      $E.\text{step}(a_h)$ ▷ fall back to the direct actions
    end for
  end if
end while

Algorithm 2 $\textsc{Arbitrate}(c, \mathcal{G}, \mathcal{D}, n)$: execute the slice only if it passes every gate

$k \leftarrow$ stage index declared in the header of c
if $k \in \mathcal{D}$ **or** k is below a stage already attempted **or** $n_k \geq R$ **then**
  return false ▷ stage gate
end if
if c uses a variable that is not defined in $\mathcal{G}$ **then**
  return false ▷ dependency gate
end if
$\hat{c} \leftarrow \text{repair}(c)$ ▷ comment out lines that do not compile
if $\hat{c}$ does not compile **or** $\hat{c}$ calls a function not in $\mathcal{G}$ **or** $\hat{c}$ has nothing to execute **then**
  return false ▷ syntax, whitelist, and vacuity gates
end if
if $\hat{c}$ already failed on stage k and nothing has changed since **then**
  return false ▷ skip a hopeless retry
end if
execute $\hat{c}$ in $\mathcal{G}$ on the robot
if it raised no exception **then**
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{k\}$ ▷ stage k is done
else
  $n_k \leftarrow n_k + 1$
end if
return true

3. **Syntax, Whitelist, and Vacuity Gates**: it compiles without a statement being commented out due to syntax error, calls only whitelisted functions, and contains an executable statement.

I Additional Simulation Experiment Details

I.1 RoboTwin Per-Task Breakdown

Table 11 reports the per-task breakdown of the aggregate results in Table 1.

Table 11: **Per-task RoboTwin 2.0 Results under the *Clean* and *Random* Evaluation Settings (50 Tasks).** Success rate (%) over 50 rollouts per task per setting. Best result per task and setting is shown in bold.

Task	$\pi_{0.5}$		EgoScale		DreamZero		WLA		**AW-0 (ours)**		**AW-0 (hybrid)**	
	Clean	Rand.	Clean	Rand.	Clean	Rand.	Clean	Rand.	Clean	Rand.	Clean	Rand.
Adjust Bottle	62	18	26	0	0	6	**100**	0	**100**	**50**	90	48
Beat Block Hammer	48	16	22	0	26	12	82	0	**94**	**28**	46	24
Blocks Ranking RGB	74	16	0	0	0	0	54	0	**96**	32	94	**82**
Blocks Ranking Size	32	0	0	0	16	0	32	0	56	0	**64**	**50**
Click Alarmclock	44	56	88	16	**100**	24	**100**	42	96	56	**100**	**62**
Click Bell	32	40	92	36	**100**	66	**100**	48	96	18	**100**	**72**
Dump Bin Bigbin	72	**70**	32	18	0	0	**100**	12	98	20	78	14
Grab Roller	94	46	96	6	64	22	**100**	0	**100**	32	92	**50**
Handover Block	16	**0**	4	**0**	0	**0**	8	**0**	**82**	**0**	4	**0**
Handover Mic	26	4	0	0	16	0	68	0	**98**	**12**	34	2
Hanging Mug	10	**4**	0	0	0	0	8	0	**30**	0	2	0
Lift Pot	8	2	0	0	8	0	88	0	**90**	**4**	66	0
Move Can Pot	28	0	24	0	4	4	64	0	76	6	**82**	**52**
Move Pillbottle Pad	60	**44**	36	0	34	16	72	0	92	6	**96**	6
Move Playingcard Away	90	**52**	46	0	0	4	94	0	**100**	46	86	30
Move Stapler Pad	22	**6**	0	0	0	0	2	0	**38**	0	26	2
Open Laptop	68	20	20	0	66	22	**90**	0	84	38	72	**48**
Open Microwave	26	8	8	2	12	8	0	0	20	4	**46**	**24**
Pick Diverse Bottles	56	18	44	0	40	16	**76**	12	**76**	**36**	74	32
Pick Dual Bottles	82	14	38	4	54	**42**	80	0	80	28	**96**	28
Place A2B Left	60	10	0	0	22	14	48	0	84	30	**98**	**58**
Place A2B Right	58	14	4	0	24	8	54	0	80	26	**96**	**80**
Place Bread Basket	68	**44**	44	0	62	12	84	0	**92**	20	80	6
Place Bread Skillet	86	**46**	32	0	36	34	64	0	**90**	14	24	2
Place Burger Fries	90	**78**	78	0	70	62	**100**	0	94	20	66	22
Place Can Basket	40	0	6	0	26	**12**	42	6	**70**	8	26	8
Place Cans Plasticbox	94	**74**	40	0	0	12	**100**	0	98	12	80	34
Place Container Plate	96	**58**	46	0	96	44	**100**	0	94	26	86	22
Place Dual Shoes	54	**12**	8	0	20	0	32	0	**64**	**12**	2	2
Place Empty Cup	74	**56**	72	0	76	50	94	0	**98**	16	96	14
Place Fan	36	**30**	12	0	0	0	8	0	**70**	4	16	10
Place Mouse Pad	18	0	2	0	10	0	26	0	**64**	0	18	**4**
Place Object Basket	24	4	0	0	10	0	42	2	**84**	**14**	50	4
Place Object Scale	58	44	8	0	36	8	36	0	70	18	**88**	**50**
Place Object Stand	**94**	40	26	0	50	26	92	6	92	26	62	**56**
Place Phone Stand	40	4	8	0	22	0	86	0	**90**	22	82	**54**
Place Shoe	64	24	22	6	18	6	72	0	**78**	22	60	**30**
Press Stapler	42	32	**100**	20	88	44	82	0	86	8	**100**	**68**
Put Bottles Dustbin	14	2	0	0	4	6	16	0	**86**	**12**	14	0
Put Object Cabinet	22	0	12	0	4	0	**68**	0	56	**6**	66	0
Rotate QRcode	50	2	32	0	0	0	22	0	**70**	4	40	**28**
Scan Object	62	**38**	4	0	40	20	**80**	0	68	10	16	10
Shake Bottle Horizontally	98	74	72	30	**100**	**86**	**100**	26	**100**	62	**100**	26
Shake Bottle	96	72	80	6	**100**	**76**	**100**	24	**100**	58	94	30
Stack Blocks Three	68	16	0	0	10	0	12	0	**92**	18	62	**56**
Stack Blocks Two	80	56	6	0	58	46	**90**	0	70	18	80	**82**
Stack Bowls Three	38	36	0	0	14	14	**88**	0	78	**40**	44	4
Stack Bowls Two	82	44	0	0	88	24	**100**	0	98	**56**	52	44
Stamp Seal	32	2	6	0	40	**16**	40	0	**78**	2	58	4
Turn Switch	**56**	**40**	12	0	14	18	42	0	44	28	52	28
Average	54.9	27.7	26.2	2.9	33.6	17.6	64.8	3.6	**80.8**	20.6	63.1	**29.2**

Table 12: **Share of simulation steps driven by each control channel in the hybrid evaluation of RoboTwin tasks in Table 1.** Code exec denotes steps driven by executing the generated code from the code expert; Direct action denotes steps directly generated by the action expert.

		Share of sim. steps		
Setting	Episode set	Code exec	Direct action	Steps / ep.
Clean	All	8.5%	91.5%	26,720
	Successful	14.5%	85.5%	11,549
	Failed	6.3%	93.7%	52,687
Random	All	5.6%	94.4%	41,716
	Successful	35.0%	65.0%	8,400
	Failed	3.7%	96.3%	55,483

I.2 Hybrid Actions Evaluation Details

Table 12 reports the fraction of simulation steps driven by each of the two control channels during the hybrid evaluation in Table 1: execution of code produced by the code expert and direct action chunks generated by the action expert. On successful episodes, code drives 14.5% of steps in the clean setting and 35.0% in the random setting. The gap follows from Table 1. In the clean setting, the action expert alone is already strong (80.8%), so there is little for code to add. Admitting code in fact lowers success to 63.1%, which we attribute to the code channel inheriting the failure modes of the low-level perception and control primitives it calls, and the arbiter admits a code slice on validity alone, without predicting whether it will outperform the action chunk it displaces. In the random setting, where direct action degrades to 20.6%, the code channel carries a third of the physical time on the episodes that succeed and lifts success to 29.2%. Executing the code expert's generation thus complements the action expert especially where learned actions are less reliable, here under visually out-of-distribution scenes. Pooled over all episodes, code execution accounts for only 8.5% of steps in the clean setting and 5.6% in the random setting. The pooled share is dominated by failed episodes, which run to the step cap and accumulate roughly five times as many steps as successful ones (52.7k vs. 11.5k in clean, 55.5k vs. 8.4k in random), almost all under direct action.

Figure 36 shows two qualitative examples of how code execution and direct action generation interleave within an episode. The top progress bar shows an episode of `blocks_ranking_rgb`, where the three blocks must be arranged in a row in red, green, blue order. We observe that the three pick-and-place cycles are almost entirely carried out by code execution, with direct action generation taking over around a stage transition. For tasks of this kind that involve regularly shaped objects and well-structured motions, the integration is able to utilize the stable and smooth behavior yields by the CaP actions. The bottom progress bar shows an episode of `pick_dual_bottles`, where the two bottles must be picked up and held together. Here, code execution correctly localized the bottle and moves the gripper to it but fails to secure the grasp due to its less regular shape. Direct action generated from the model then takes over and completes the grasp, after which code execution resumes to finish the episode.

J Agentic Annotation Details

J.1 Low-level Primitives

Figure 37 shows the two API surfaces for the parallel-gripper (left) and the dexterous-hand (right) embodiments. They share ten semantic contact verbs but differ in their low-level primitives: end-effector poses for the gripper and relative motion for the dexterous hand. Each API call declares the contact events it produces, which is what allows a program to be grounded against a demonstration.

J.2 Annotation Agent Prompts and Coordinator Protocol

The framework has two agents: a VLM *annotator* that writes program lines for one subtask slice per call, and an agentic *coordinator* that audits annotator output between rounds and grows the skill library (Stage 1). After the library is frozen, the same annotator runs over the remaining corpus and

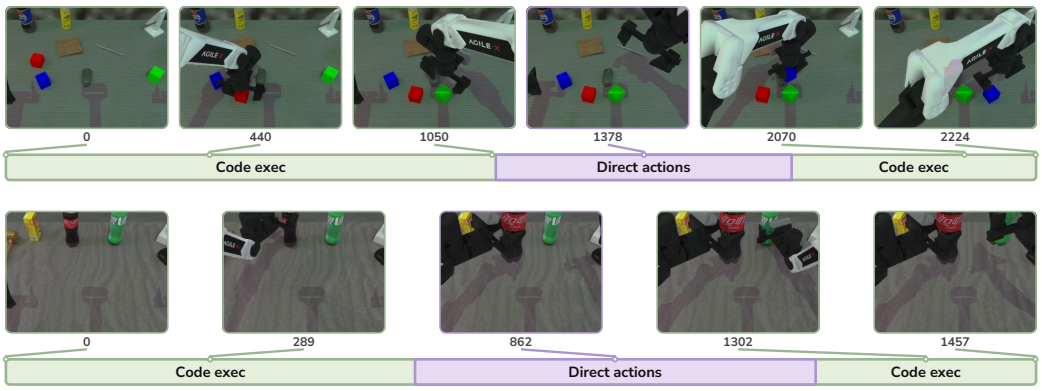


Figure 36: **Qualitative examples of hybrid actions for RoboTwin 2.0 tasks.** Each row shows one successful episode from the hybrid evaluation in Table 1. Top: `blocks_ranking_rgb`. Bottom: `pick_dual_bottles`.

```python
class ParallelGripperApi:
    # side: "left" or "right"
    # Declaration only; types omitted.

    # Semantic APIs
    def pick(self, object_name, side): ...
    def grasp(self, object_name, side): ...
    def place(self, object_name, target_name, side): ...
    def release(self, side): ...
    def regrasp(self, object_name, side): ...
    def insert(self, object_name, target_name, side): ...
    def attach(self, object_name, target_name, side): ...
    def handover(
        self, object_name, from_side, to_side): ...
    def rotate(self, object_name, angle_deg, side): ...
    def press(self, target_name, side): ...

    # Low-level primitives
    def move_arm(self, side, target_pos, target_rpy): ...
    def freespace_move(
        self, left_target_pos=None, left_target_rpy=None,
        right_target_pos=None, right_target_rpy=None): ...
    def servo_ee_delta(
        self, side, delta_pos, duration_s=1.0): ...
    def go_home(self): ...
    def close_gripper(self, side): ...
    def open_gripper(self, side): ...

    # Perception and state
    def get_robot_state(self): ...
    def sample_grasp_pose(
        self, object_name, camera='top', max_grasps=10,
        top_down_only=False): ...
    def select_best_grasp(
        self, grasp_candidates, side='right'): ...
    def detect_objects_oneshot(
        self, query, camera='top'): ...
```

```python
class DexterousHandApi:
    # side: "left" or "right"
    # Declaration only; types omitted.

    # Semantic APIs
    def pick(self, object_name, side): ...
    def grasp(self, object_name, side): ...
    def place(self, object_name, target_name, side): ...
    def release(self, side): ...
    def regrasp(self, object_name, side): ...
    def insert(self, object_name, target_name, side): ...
    def attach(self, object_name, target_name, side): ...
    def handover(
        self, object_name, from_side, to_side): ...
    def rotate(
        self, object_name, axis, angle_deg, side): ...
    def press(self, target_name, side): ...

    # Low-level primitives
    def move_arm(self, side, direction, distance=None): ...
    def goto_relative(
        self, side, object_name, relation, clearance=None): ...
    def close_hand(self, side): ...
    def open_hand(self, side): ...
```

Figure 37: **Declaration-only APIs for parallel grippers and dexterous hands.**

over downstream demonstrations, reusing cached subtask code where the embodiment's evidence verifies it. The full API docstrings and skill documentation are appended to the system prompt and omitted here.

Shared annotator system instructions

Role and output
Read the frames of one subtask and express the observed actions in Python using only the `robot` API and documented skills. Output only new code for this subtask, continuing the preceding program.

Grounding and coverage
The subtask annotation states *what* happens; the frames ground *which* objects and *how* they move. Cover the whole frame span and invent nothing that neither source supports.

Abstraction level
Prefer the single highest-level call whose contact effects match; never wrap a semantic verb in the primitives it already contains. Use bare open/close primitives only when no verb fits.

Repetition and literals
Write repeated in-place twisting as one literal loop with the count read from the frames. Hand sides and loop counts must be literals. End on a complete statement.

Static validation
Code is checked without execution: syntax in program context, existence of called functions, argument binding, and vocabulary membership of literals. On failure the annotator sees its output with the problems and rewrites the slice.

System prompt for egocentric human demonstrations (Pre-train)

Setting
Two hands, `'left'` and `'right'`, seen from a head-mounted camera. No contact-event list is given; infer every contact from the frames and the annotation, judging object motion relative to the scene.

Contact semantics
`pick`/`grasp` close a hand (pick lifts, grasp holds in place); `place`/`release`/`insert`/`attach` open it; `regrasp` opens then closes; `press` closes then opens; `handover` closes the receiver before opening the giver.

Motion and naming
Motion is purely semantic: `move_arm` along a documented direction, `goto_relative` at a documented relation to a named object, `rotate` about a documented axis. No coordinates; objects are named in plain language.

Conventions
Calls return `None`. Distances only when visibly measurable; angles as semantic constants.

System prompt for bimanual robot demonstrations, YAM (Pre-train)

Setting
Two arms seen from a top-down camera. The sensor-derived gripper-event list is authoritative: emitted code must reproduce every open/close transition on the named arm, in order, with nothing added, dropped, or reordered.

Event gaps
If the annotation implies a contact the event list lacks, write the approach and transport in full but do not emit the missing open/close call.

Contact semantics
`grasp`/`pick` close; `place`/`release` open; `regrasp` opens then closes; `insert` closes then opens; `handover` closes, closes, then opens. A repeated same-arm cycle in the list is one literal loop.

Perception and motion
Every pose used by a motion call comes from a perception result earlier in the program; nothing is hard-coded. Absolute single-arm, synchronized two-arm, and small relative motion each have their own primitive.

Per-slice user message

Overall task
[task instruction]

Current subtask
[human subtask annotation]

Frames
[ordered frames with timestamps]. On YAM, the frame before each gripper event is captioned with the event index, kind, arm, and time.

Program so far
[code written for the preceding subtasks]

Egocentric directive
Decide from the frames and annotation where each hand closes on and opens off an object, and produce exactly those contacts with the visible motion around them. A comment-only slice is correct only when the hands neither move nor touch anything.

YAM directive
Observed gripper events: *[ordered list, repeated cycles collapsed to a count]*. Reproduce this sequence exactly, with the visible motion around it; a brief blip with nothing in hand gets the minimal call.

Output
Continue the program with this subtask's code only.

Only the directive for the current dataset is included.

Annotation under the frozen library (Post-train and corpus scale)

Same prompt, frozen vocabulary
The system prompt above is used unchanged, with the skill block fixed to the frozen library. Inherited skills are called as plain functions with `robot` as the first argument.

No new vocabulary
The annotator may not invent helper names; a maneuver no verb or skill expresses is written from primitives. Extending the library is the coordinator's decision.

Verified reuse
A per-task cache of subtask code is consulted before each call. A variant is reused only when the embodiment's evidence verifies it: on YAM, the contact events its code implies match the observed events (mirrored forms accepted, motion arguments re-derived); on egocentric data, its hand-motion profile is within a fixed distance of the new slice, and loop templates are never reused. A miss triggers the ordinary call, whose output is written back.

Coordinator protocol (Stage 1, skill discovery)

Role
A coding agent under a fixed written protocol that dispatches rounds, audits their output, and owns the skill library. It never writes program code and never edits the annotator, prompts, API, or verification code.

Round lifecycle
Dispatched, gated (all episodes annotated and scored), audited, done (admissions applied and logged). The library is frozen from dispatch to done, and the next round starts only then.

Audit
Read the round summary, then per-task results, then the stored trace of each questionable slice (frames seen, code written, verification outcome). Classify each as detector limit, annotator misread, abstraction-level break, vocabulary gap, or segmentation artifact. Only vocabulary gaps feed skill proposals.

Skill admission
A candidate is a straight-line composed function over the API, distilled from observed call sequences. It is admitted if generic (episode-specific values lifted to parameters), recurrent across at least two tasks, at the right level (a maneuver, not a task recipe), and not a duplicate. Its contact contract is derived by expansion; on YAM the evidence slices must also reconcile against observed events. Admitted skills enter the annotator prompt from the next round.

K Limitations and Future Work

Program Grounding and Skill Coverage. The code stream learns from automatically annotated demonstrations and a curated skill library, making its planning quality dependent on annotation accuracy and skill coverage. Generated programs can select an appropriate skill yet contain incorrect arguments, as illustrated in Appendix F.1; syntactic validity therefore does not ensure correct execution. Future work could use execution feedback to improve program grounding and discover or refine reusable skills beyond the initial library.

Predictive Robustness and Hybrid Execution. Structured planning improves WAM performance but does not eliminate sensitivity to visual distribution shifts. Hybrid execution improves robustness under RoboTwin randomization, yet reduces clean-setting performance (Appendix I.2). This trade-off reflects both the reliability of the underlying primitives and the limitation of selecting code based on validity without estimating its advantage over the action chunk. Learning to arbitrate using execution reliability and expected task progress could better exploit the complementary strengths of the two modes.

Toward Continual Agentic Learning. Our real-world hybrid evaluation is limited to YAM platform due to the lack of code-as-policy systems for dexterous hand, and the current system does not learn online from its own interactions. The unified design nevertheless offers a basis for coupling program generation and world prediction with execution feedback and model updates. Future work could explore iterative plan revision, skill-library expansion, and autonomous learning through sustained environmental interaction, with broader evaluation across tasks and embodiments.