CTRL-Z: CODE AGENT TRIAL ROBOT ACTIONS AND REASON WITH LATENT WORLD MODEL $\mathcal{Z}$

Anonymous authors
Paper under double-blind review

ABSTRACT

Code-as-policy approaches translate language instructions into executable programs, enabling compositional task planning and reuse of robot skills. However, generating a valid program don't establish whether a candidate action will produce the desired physical change. Action-conditioned world models offer a way to anticipate such changes, but their latent predictions are not directly expressed in the language of executable programs generated by code agent. To address these questions, we propose a closed-loop system in which a code agent can read, reason, make prediction and update object physical parameters based on those predictions, and generate actions accordingly. To make it readable for the agent, we build Structured Object Program, a compositional program describing object primitive geometry, physical parameters, interaction properties and rendering object programs into geometric views. For the prediction system, we propose a dual-stream world model that accepts RGB images and rendered geometric views as input while predicting future visual and geometric latent variables. Physical parameters regarding the dynamic parts of the object are decoded from these predicted latents and fed back to the agent, enabling it to understand how its current actions affect the object's future properties. Experiments across manipulation tasks and object generalization settings show that our framework improves task success, cross-object generalization, and agent interaction efficiency

1 INTRODUCTION

"Code as Policies" utilizes language models to generate composable and executable programs that invoke perception and control interfaces, enabling the integration of task logic, geometric computation, and feedback control within a unified code representation for general-purpose robot manipulation (Liang et al., 2022). Although execution feedback enables iterative correction, program executability alone does not guarantee the expected physical effects. If an agent does not know how the programs it generates will alter the physical world, it can only continue to grope blindly without manipulation guidance.

Research on language-driven robotic control has explored value function guidance based on task-skill relevance (Ahn et al., 2022), feedback from scene descriptions (Huang et al., 2022), and the mapping of language-inferred affordances onto 3D maps (Huang et al., 2023). However, these mechanisms only provide post-action recovery and certain manipulation knowledge for the robot, rather than establish what physical outcomes those guidance will induce. Skill-value grounding assesses skill feasibility but does not explicitly represent skill-resulting object state changes. While feedback-based planning adapts to observed outcomes but often ignore whether feedback can correct failures caused by previous actions. Spatial value maps guide motion planning but remain limited by geometric perception accuracy and the generalization of object contact dynamics.

Moving from post-execution correction toward pre-execution evaluation requires anticipating the physical consequences of candidate actions. Action-conditioned world models provide how environments changes under actions, enabling the comparison of alternative behaviors (Ebert et al., 2018; Zhou et al., 2024; Assran et al., 2025). However, these predictions are typically expressed as images or latent features, rather than code-program-accessible object physical states. Therefore, a shared interface is needed to connect code agents with world models, making predicted physical outcomes

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

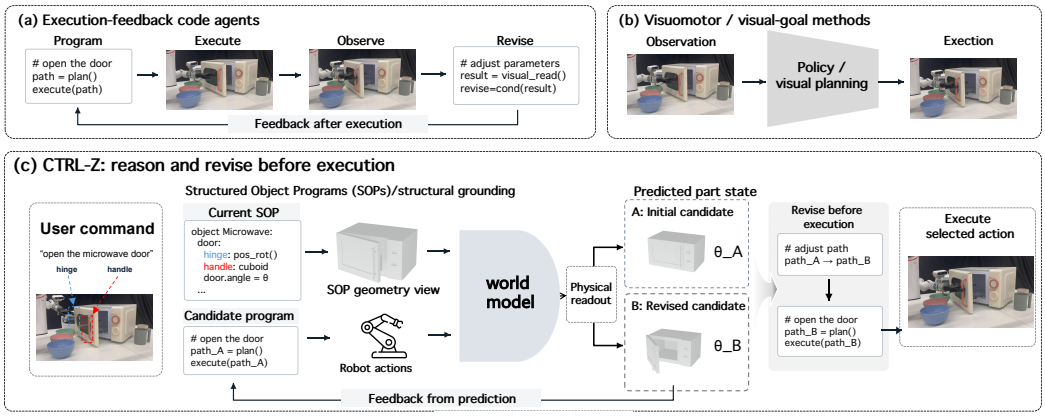


Figure 1: **Predictive program revision before execution.** (a) Execution-feedback code agents revise programs based on observed execution outcomes. (b) Visuomotor policies and visual-goal planners derive actions from visual observations. (c) Our framework, CTRL-Z, connects code agents and world models through Structured Object Programs (SOPs). RGB observations and SOP-rendered geometry jointly support dual-stream prediction, while a physical readout converts predicted latents into object-part states accessible to the agent. These predictions enable the agent to evaluate candidate interactions and revise action programs before executing the selected action.

accessible to programmatic reasoning and enabling action evaluation before execution rather than relying solely on correction afterward.

Existing code programs and scene languages provide compositional representations for geometry synthesis and scene editing (Jones et al., 2020; Zhang et al., 2024). However, their emphasis on generating or reconstructing objects leaves less explicit support for describing how objects can be interacted with, providing manipulation knowledge, and updating dynamic object states based on action executions. To fill these gap, we build Structured Object Programs (SOPs) by drawing on and combining complementary ideas from prior work (Jones et al., 2020; Zhang et al., 2024; Wei et al., 2025; Sun et al., 2025) that associate semantic parts and parameterized geometry with affordances and manipulation templates, enabling the agent to compute grasp poses and motion directions from object geometry, while the renderer program uses that geometry and the current object configuration to generate geometric views that serve as inputs to the world model. Besides, Object poses and part configurations are represented numerically, allowing predicted outcomes to update SOP instances and be compared with task goals. These properties allow code agents to reference object parts, formulate candidate actions, and inspect predicted state changes within a shared object representation.

On this basis, we propose a closed-loop code agent framework that connects programmatic reasoning, physical prediction, and execution through Structured Object Programs. Our framework employs a dual-stream action-conditioned world model that takes RGB observations and rendered geometric views as input to predict future visual and geometric latents. At runtime, the code agent proposes candidate actions, and a physical readout decodes the latent predictions into parameters of dynamic object parts. These parameters are returned to the agent through the object instance program in SOPs, enabling it to evaluate anticipated action effects and refine its decisions before execution. Our contributions are threefold:

- We propose a closed-loop code agent framework that extend code agents from post-execution correction to pre-execution evaluation, enabling them to assess predicted physical outcomes and revise control programs before acting.
- We establish a bidirectional connection between code and physical prediction through Structured Object Programs: rendering geometric views for a dual-stream world model, while decoded predictions update object states accessible to the agent's programs.
- Systematic experiments show higher success than code-based and visuomotor baselines, better generalization and interaction efficiency, and further gains from world-model-guided offline training and skill refinement.

2 Related Work

Language- and Code-Based Robot Control. Code-as-policy methods aim to enable robots to perform diverse, multi-step manipulation tasks that require spatial reasoning and flexible skill composition. These systems typically combine high-level language reasoning with perception modules and reusable control primitives. Within this framework, SayCan grounds skill selection through value functions (Ahn et al., 2022), while Code as Policies leverages language models to select skills, invoke APIs, or synthesize executable programs for robot control (Liang et al., 2022). To support adaptation during execution, Inner Monologue incorporates feedback from scene descriptions into planning (Huang et al., 2022). VoxPoser further connects language-derived spatial objectives with model-based planning and online dynamics learning (Huang et al., 2023). More recently, CaP-X advances embodied coding agents through iterative execution, visual feedback, and skill synthesis (Fu et al., 2026). However, these systems primarily emphasize action selection and adaptation to execution feedback, while using predicted object-state changes to identify potential failures and revise robot programs before execution remains underexplored.

World Models for Robotic Manipulation. World models support robotic planning by anticipating how environments change under candidate actions. Early work such as Visual Foresight combines action-conditioned video prediction with model-predictive control for object manipulation (Ebert et al., 2018). Moving to latent-space prediction, DINO-WM predicts pretrained spatial features and optimizes action sequences toward visual goals (Zhou et al., 2024). Along this direction, V-JEPA 2-AC learns visual latent dynamics from robot interaction data for image-goal planning (Assran et al., 2025), while V-JEPA 2.1 further improves spatial detail and temporal consistency in the representations used for prediction (Mur-Labadia et al., 2026). For policy generation, DreamZero jointly models future video and actions to learn robot policies that generalize across tasks and environments (Ye et al., 2026). Together, these works demonstrate how predictive models support planning and policy generation. However, translating their predictions into explicit dynamic part parameters that robots can reference and inspect for policy evaluation and revision remains underexplored.

Structured Representations for Manipulation. Structured representations support robotic manipulation by making object relationships, geometric requirements, and interaction constraints explicit for reasoning and control. These methods organize task and scene information as symbolic descriptions, relational graphs, or parameterized programs. At the planning level, PDDLStream connects symbolic planning with continuous sampling procedures (Garrett et al., 2020), while PRoC3S combines generated programs with continuous constraint satisfaction (Curtis et al., 2024) and ReKep expresses manipulation requirements as optimizable programs over 3D keypoints (Huang et al., 2024). At the scene level, ConceptGraphs organizes objects and their semantic spatial relationships in 3D scene graphs (Gu et al., 2023). However, these methods primarily organize task constraints or scene relationships, with less emphasis on expressing object geometry and interaction properties through compositional, executable templates. Analytic Concepts provide such a representation by linking semantic parts to parameterized geometry and manipulation knowledge (Wei et al., 2025; Sun et al., 2025). Drawing on the complementary strengths of these approaches, we construct Structured Object Programs (SOPs) that bring together compositional geometry, manipulation knowledge, and explicit dynamic states, providing a shared interface through which code agents query object structure and world models receive rendered geometry and return predicted state updates.

3 Preliminaries

Code-as-policy manipulation. We consider language-conditioned robotic manipulation, where an agent receives an instruction L and observations $o_t = (I_t, x_t)$, comprising visual observations I_t and robot proprioceptive states x_t. Let $\mathcal{H}_t = (o_{0:t}, a_{0:t-1})$ denote the interaction history. In the code-as-policy formulation, a language model generates an executable program that organizes task decomposition, geometric computation, and calls to perception and control interfaces (Liang et al., 2022). At a program-generation step, we write

$$P_t = \mathcal{G}_\phi(L, \mathcal{H}_t), \qquad a_t = \mathrm{Exec}(P_t, \mathcal{H}_t),$$

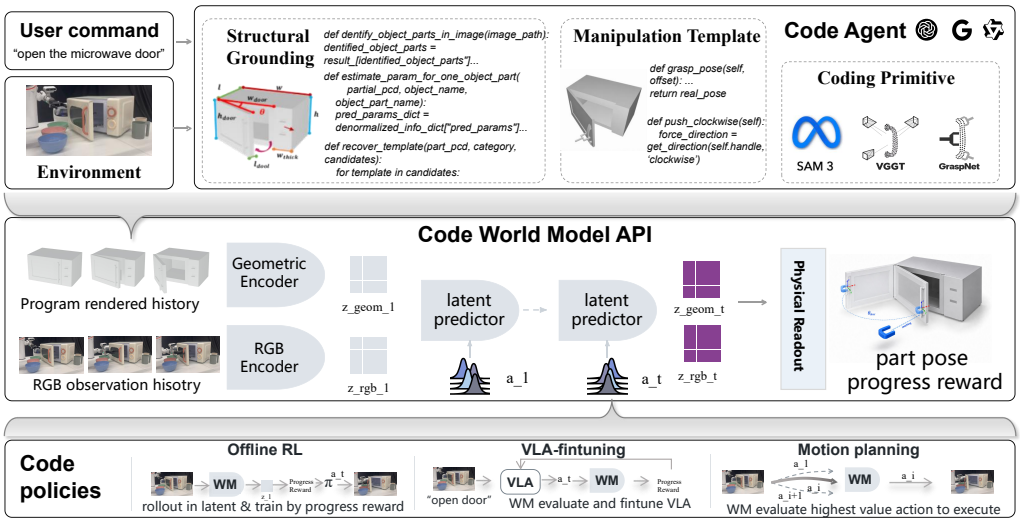


Figure 2: Overview of closed-loop framework. The code agent combines perception tools and Structured Object Program manipulation priors to generate candidate actions. A dual-stream world model predicts future RGB and geometric latents, which a physical readout converts into dynamic object parameters for Program-based progress evaluation. The code agent can leverage this world model and physical readout to perform offline reinforcement learning, fine-tuning, and motion planning on action programs or the visuomotor policies being invoked, thereby enhancing performance.

where $\mathcal{G}_\phi$ denotes the code agent and Exec advances program execution to produce a robot command, such as an end-effector target or controller input. The program may contain reusable functions, conditional branches, and feedback loops, allowing to refer observed execution outcomes.

Action-conditioned world models. A world model predicts how the environment evolves under candidate robot actions (Ebert et al., 2018; Zhou et al., 2024; Assran et al., 2025). In a latent formulation, an encoder E_ψ maps visual observations to representations $z_t = E_\psi(I_t)$. Given an observation history of length h, the current robot TCP Pose state x_t, and a candidate action sequence $\mathbf{a}_t = (a_t, \ldots, a_{t+H-1})$, a predictor models future latent states over a horizon H:

$$\hat{z}_{t+1:t+H} = F_\theta(z_{t-h+1:t}, x_t, \mathbf{a}_t)\,.$$

Here, F_θ learns action-conditioned transitions from interaction data. The resulting predictions support the evaluation of candidate actions against task goals. However, latent states do not inherently expose named object parts or explicit physical state variables for programmatic reasoning.

4 Methodology

We propose a closed-loop framework that connects a code agent with an action-conditioned world model through Structured Object Programs. Fig. 2 summarizes this process. The code agent combines language instructions, perception tools, and Structured Object Program derived physical priors to generate executable control programs or invoke policy APIs, including vision-language-action models and world action models. The world model evaluates candidate actions by predicting their physical effects, which are decoded into dynamic object parameters to update Structured Object Program instances and compute task-progress rewards. This feedback enables the code agent to reason about physical outcomes before execution and optimize action programs or invoked policies offline, grounding its decisions in predicted interactions with the physical world. The complete pseudocode of our framework is provided in Appendix A.5.

4.1 Structural Grounding

Structured Object Programs represent physical objects through reusable, parameterized programs that describe semantic parts, geometric structure, kinematic relationships, part poses, and manipu-

lation knowledge. Each part is associated with a template that constructs its geometry from basic primitives, such as cuboids and cylinders, together with geometric parameters, affordances, and manipulation templates. By invoking structural grounding, the agent instantiates and combines these components according to the observed geometry and part relationships. An unfamiliar object can therefore be represented through familiar components, even when their overall composition or object category was absent from training dataset. Formally, an structured object program, its constituent parts, and its rendered view are represented as

$$\mathcal{O}_t = (c, \{P_i\}_{i=1}^{N_p}, \mathcal{K}, s_t), \qquad P_i = (\tau_i, \Theta_i, A_i, F_{\tau_i}), \qquad G_t = \mathcal{R}(\mathcal{O}_t).$$

Here, c denotes the object category, N_p is the number of semantic parts, and $\mathcal{K}$ specifies their kinematic relationships, including joint types, axes, and limits. For each semantic part P_i, τ_i identifies its reusable template type, Θ_i specifies its estimated geometric parameters, A_i denotes its affordances, and F_{τ_i} contains the associated manipulation templates. The dynamic state s_t comprises the object pose and part configurations, from which individual part poses can be computed. The renderer $\mathcal{R}$ maps the instantiated object to a geometric view G_t. Template types encode reusable structure and manipulation knowledge, while geometric parameters and dynamic states instantiate scene-specific objects. Updating s_t changes the object state and rendered geometry while preserving part identities. This mechanism allows the agent to maintain an up-to-date representation of the object state and query the world model to anticipate the physical effects of its actions. See Appendix A.4 for details on Structured Object Programs, the object and interaction priors they provide to code agents, and their use in our world-model-guided framework.

4.2 Predictive Evaluation and Program Revision

To predict geometric and visual changes in objects, the world model combines the available RGB history with geometric views rendered from the corresponding object Program instances. At an observed time step,

$$G_t = \mathcal{R}(\mathcal{O}_t), \qquad z_t = E_\psi(I_t), \qquad z_t^{\text{geom}} = E_{\text{geom}}(G_t), \tag{1}$$

where E_ψ and E_{geom} encode the RGB and geometric streams, respectively. RGB observations capture appearance and interaction context, while the rendered geometric views expose estimated part shapes and configurations without appearance textures. The predictor conditions on the RGB latent history $z_{t-h+1:t}$, the geometric latent history $z^{\text{geom}}_{t-h+1:t}$, robot state x_t, and the candidate actions:

$$\left(\hat{z}_{t+1:t+H}, \hat{z}^{\text{geom}}_{t+1:t+H}\right) = F_\theta\left(z_{t-h+1:t}, z^{\text{geom}}_{t-h+1:t}, x_t, \mathbf{a}_t\right). \tag{2}$$

For each candidate actions, the predictor starts from a separate copy of the available histories and recursively appends the predicted latents to forecast subsequent steps.

To make these predictions usable by the program, the physical readout decodes the dynamic object state and inserts it into the corresponding SOP instance:

$$\hat{\mathcal{O}}_{t+k} = \left(c, \{P_i\}_{i=1}^{N_p}, \mathcal{K}, \underbrace{D_\rho\left(\hat{z}_{t+k}, \hat{z}^{\text{geom}}_{t+k}; \mathcal{O}_t\right)}_{\hat{s}_{t+k}}\right). \tag{3}$$

Here, the readout output is denoted by $\hat{s}_{t+k}$ for $k = 1, \ldots, H$. The updated SOP instance is re-rendered into a geometric view, allowing the agent to read the predicted object parameters and visually inspect the corresponding part motions.

The progress reward API evaluates the predicted trajectory using

$$J_t(\mathbf{a}_t) = \sum_{k=1}^{H} r_L(a_{t+k-1}, \hat{s}_{t+k-1}, \hat{s}_{t+k}), \qquad \hat{s}_t = s_t, \tag{4}$$

where r_L scores task progress using the candidate robot actions and corresponding predicted object state. The progress reward supports comparison between candidates, the agent can use these queries to change control parameters or revise the relevant action program, and reevaluate resulting actions again. The detailed computations are provided in SOP-based reward computation in Appendix A.4.

Training. To support multi-step prediction and state-based evaluation, we jointly train the predictor and physical readout with

$$\mathcal{L} = \lambda_{\mathrm{TF}}\mathcal{L}_{\mathrm{TF}} + \lambda_{\mathrm{AR}}\mathcal{L}_{\mathrm{AR}} + \lambda_{\mathrm{phys}}\mathcal{L}_{\mathrm{phys}}. \tag{5}$$

We extend teacher-forced and autoregressive latent prediction objectives to both streams. Details of the world-model architecture and training are provided in Appendix A.7. Teacher-forced training uses encoded ground-truth histories, while autoregressive training recursively uses predicted latents to match inference-time rollouts. The physical loss $\mathcal{L}_{\mathrm{phys}}$ compares decoded object poses and part configurations with ground-truth labels, supervising the object state estimates used for object program instance updates and task-progress evaluation.

4.3 World-Model-Guided Code Agent

Given instruction L and observation I_0, structural grounding instantiates the SOP $\mathcal{O}_0$. The resulting SOP provides geometric and manipulation priors for action generation, with the grounding procedure detailed in Appendix A.6. At step t, the agent uses $\mathcal{O}_t$, interaction history, tool feedback, and retrieved experience to generate or revise a control program P_t. The program invokes F_{τ_i} to compute grasp poses and motion directions, and produces candidate actions $\mathbf{a}_t$ either directly or through policy APIs, including vision-language-action and world action models.

Both control routes use the world model to obtain predicted SOP instances $\hat{\mathcal{O}}_{t+1:t+H}$ and returns $J_t(\mathbf{a}_t)$. The agent inspects numerical state changes and rendered geometric views, compares predicted outcomes with the task goal, and revises action parameters or program operations. Revised candidates are evaluated again before selection.

The selected action advances the interaction in one of two modes:

$$\mathcal{O}_{t+1} \leftarrow \begin{cases} \mathrm{StructuralGrounding}(I_{t+1}), & \text{planning}, \\ \hat{\mathcal{O}}_{t+1}, & \text{offline}. \end{cases} \tag{6}$$

In planning mode, the action is executed in the environment, and new observations refresh the SOP and latent histories. Execution traces, visual changes, and discrepancies between predicted and observed SOP parameters guide failure diagnosis, program repair,and world model optimization. In offline mode, predicted states and latents advance the interaction without new environment observations, feeding the predicted latent into the latent history to perform recursive prediction. world-model-guided policy optimization is described in Appendix A.8..

In addition, the agent distills successful execution trajectories into reusable skill programs, stored with their corresponding SOP instances and application conditions. When encountering an object with similar geometric structure, it retrieves a relevant skill and adapts its grasp poses and motion parameters to the current object's geometry. The adapted skill can also be rolled out in the world model to evaluate its task progress. Using the resulting progress rewards, the agent revises the skill's control parameters or programs and evaluates it again before environment execution. This process allows successful experience to be reused across objects while refining each skill for the geometry and configuration of the current object. The complete procedure is provided in Appendix A.5.

5 Experiments

5.1 Overall Performance and Policy Post-Training

This section answers: *How does our framework compare with existing manipulation methods, and can world-model-based visuomotor policy API like VLA training further improve performance?*

Table 1 compares performance across articulated-object opening, pick-and-place, and precision manipulation. Our framework achieves success rates of 46.1%, 53.4%, and 49.8%, exceeding the strongest baseline in each task group by 7.9, 5.0, and 8.2 percentage points, respectively. Compared with CaP-Agent0 using the same code-agent backbone, the improvements are 8.0, 5.9, and 8.2 percentage points. We further validate our framework on real-world, results in Appendix A.1.

The framework also improves performance when invoking an existing visuomotor policy. Integrating $\pi_{0.5}$ without additional offline RL training achieves 50.6%, 61.3%, and 54.2% success, improv-

Table 1: Task success rates (%, ↑) across three manipulation task groups. CaP-Agent0 and our framework use the same GPT-5.6-Sol (medium) code-agent backbone.

Method	Open articulated objects	Pick and place	Precision operation
	Success (%) ↑	Success (%) ↑	Success (%) ↑
Diffusion Policy	15.4 ± 1.7	24.6 ± 0.6	18.8 ± 1.1
Online PPO	23.9 ± 1.2	31.3 ± 0.8	25.7 ± 1.3
Online TD-MPC2	25.5 ± 1.0	34.6 ± 0.6	29.1 ± 0.9
DreamerV3	30.6 ± 1.1	39.0 ± 0.5	32.9 ± 1.4
$\pi_{0.5}$	38.2 ± 1.3	47.1 ± 0.9	39.4 ± 1.2
Fast-WAM	37.8 ± 0.8	48.4 ± 0.7	39.2 ± 1.3
DreamZero	34.1 ± 1.4	43.2 ± 1.0	33.5 ± 1.5
CaP-Agent0	38.1 ± 0.9	47.5 ± 0.4	41.6 ± 0.8
Ours (code-generated actions)	46.1 ± 1.5	53.4 ± 1.0	49.8 ± 1.4
Ours ($\pi_{0.5}$, no additional training)	50.6 ± 0.9	61.3 ± 0.5	54.2 ± 1.7
Ours ($\pi_{0.5}$ + WM-RL)	55.0 ± 0.4	65.8 ± 0.5	57.9 ± 1.1

ing over standalone $\pi_{0.5}$ by 12.4, 14.2, and 14.8 percentage points. This comparison demonstrates the benefit of combining a visuomotor policy with SOP-guided reasoning and predictive evaluation.

Reinforcement learning with world-model rollouts further increases success to 55.0%, 65.8%, and 57.9%. Relative to the same framework without offline RL training, these correspond to gains of 4.4, 4.5, and 3.7 percentage points. WM-RL updates only the invoked visuomotor policy, while the code-agent backbone remains frozen. The additional gains support using predicted interactions and SOP progress rewards for test-time-training alongside inference-time predictive evaluation.

5.2 Effect of Predictive Evaluation

This section answers: *Does pre-execution physical prediction improve code-agent manipulation beyond current SOP priors?*

Table 2 compares three agent backbones on 17 objects, with 10 episodes per object per seed. The Current SOP, RGB WM, and Dual-stream WM configurations use the same structural-grounding procedure to obtain SOP priors from observations. Both world-model variants decode future object states and insert them into SOP instances for task-progress evaluation, using the same SOP update and reward computation procedures. The oracle simulator provides ground-truth object states and rewards as a privileged-information reference.

Current SOP priors alone yield limited improvements, whereas world-model feedback increases success rates across all three backbones without updating code-agent weights. For GPT-5.6-Sol, the dual-stream configuration raises success from 5.8% with current SOP priors to 45.2% and reduces average environment attempts from 29.8 to 20.5. The dual-stream model also achieves higher success rates than the RGB-only model for every backbone, supporting the additional benefit of SOP-rendered geometric inputs. These results support predictive evaluation beyond current structural priors, with the largest reduction in environment attempts observed for GPT-5.6-Sol.

5.3 Generalization across Object Structures

This section answers: *Can compositional object representations support world-model-guided manipulation of unseen objects and structures?*

We evaluate five settings defined relative to *Seen instance* in world-model training data. *Unseen instance* introduces new objects within seen categories and familiar part compositions. *Unseen composition* introduces previously unseen numbers, arrangements, or assemblies of familiar part templates within seen categories. *Unseen category* evaluates object categories absent from training but sharing geometric primitives and joint types with training objects. Finally, *Geometry and kine-*

Table 2: Manipulation performance without code-agent post-training. Success denotes task success rate (%, ↑); Attempts denotes mean environment attempts (↓). Metrics are aggregated over objects and episodes within each seed, then reported as mean ± sample standard deviation across 10 seeds.

Agent backbone	**Metric**	Without prediction		Predictive evaluation		Oracle Simulator object state
		Direct Agent	Current SOP	RGB WM	**Dual-stream WM**	
GPT-5.6-Sol (medium)	Success	5.1 ± 0.6	5.8 ± 0.7	26.5 ± 1.9	45.2 ± 1.4	46.4 ± 1.2
	Attempts	29.4 ± 0.3	29.8 ± 0.4	25.1 ± 2.7	20.5 ± 2.2	18.6 ± 1.3
Gemini 3 Pro	Success	0.0 ± 0.0	0.0 ± 0.0	5.5 ± 1.3	8.3 ± 1.7	12.7 ± 2.5
	Attempts	30.0 ± 0.0	30.0 ± 0.0	29.8 ± 0.6	29.5 ± 1.1	29.6 ± 1.2
Qwen 3 Coder 30B	Success	0.0 ± 0.0	0.0 ± 0.0	3.4 ± 1.8	7.2 ± 2.0	10.8 ± 1.6
	Attempts	30.0 ± 0.0	30.0 ± 0.0	29.9 ± 0.3	29.9 ± 0.3	29.8 ± 0.5

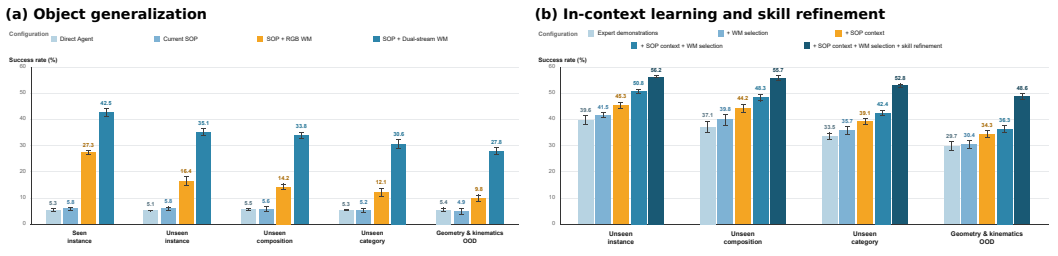


Figure 3: Generalization and experience reuse with same GPT-5.6-Sol (medium). (a) Manipulation performance across object generalization settings. (b) Effects of WM-based demonstration selection, SOP context, and WM-guided skill refinement. All configurations in (b) use expert demonstrations; SOP context provides the objects' structural information. Bars show mean success rates, and error bars indicate standard deviations.

matics OOD evaluates generalization to geometric configurations and motion patterns beyond those represented in the training data.

As shown in Fig. 3(a), the dual-stream configuration achieves the highest success rate in all five settings. It reaches 33.8% on unseen compositions and 30.6% on unseen categories, compared with 14.2% and 12.1% for the RGB-only model. Under geometry and kinematics shifts, it maintains a success rate of 27.8%, exceeding RGB-only prediction by 18.0 percentage points. These results demonstrate that the benefit of SOP-rendered geometry extends beyond familiar object instances to new part compositions, categories, and motion patterns. The consistent improvements over RGB-only prediction support the value of geometric inputs for generalization within the same code-agent framework. For the world model, SOP-rendered views expose relative part configurations and their changes under candidate actions. These geometric relationships provide shared cues across objects with different appearances and part compositions so that guarantee the predication quality.

5.4 Physical Prediction Accuracy

This section answers: *Does SOP-rendered geometry improve multi-step physical state prediction?*

We compare RGB-only, duplicated-RGB, and SOP-geometry inputs over $H = 52$ autoregressive steps, using shared initial states and action sequences and the same physical-readout architecture. SOP geometry is initialized through structural grounding, and decoded states update the candidate SOP instances and their rendered views, while predicted latents advance the rollout without ground-truth corrections. The oracle reference instead receives ground-truth geometry at every step, while its RGB history remains autoregressive.

As shown in Figure 4, SOP geometry maintains lower position and rotation errors than both RGB-based variants across the reported steps. Duplicating RGB does not recover these gains, supporting

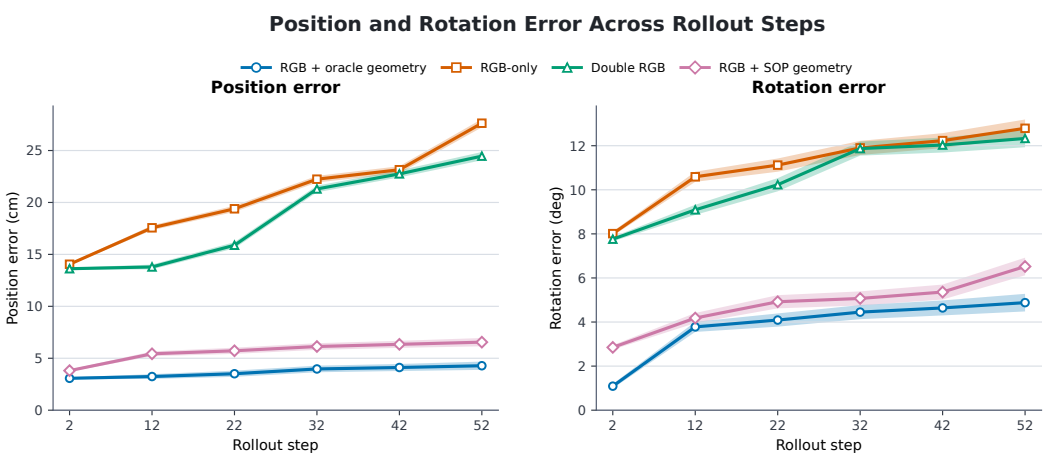


Figure 4: Moving-part position (left) and rotation (right) errors across rollout steps. Lines show mean errors, and shaded bands indicate standard deviations.

the contribution of geometric information beyond an additional RGB stream. The oracle achieves the lowest errors under continuous ground-truth geometric feedback. These results demonstrate that SOP-rendered geometry improves multi-step state estimation, providing more accurate part configurations through the object-program interface.

5.5 In-Context Learning and Skill Refinement

This section answers: *Can SOP-enriched demonstrations and world-model evaluation improve experience transfer to unseen objects?*

Fig. 3(b) evaluates demonstration context, selection, and skill refinement with the code-agent weights fixed. All configurations share the current-object SOP and the same base planning procedure. SOP context additionally describes the demonstration objects' parts, geometry, and kinematic relationships. WM selection evaluates candidate skills adapted to the target object and selects their corresponding demonstrations using predicted progress rewards.

Adding SOP context improves success over the original demonstrations by 4.6–7.1 percentage points across the four settings. This supports the value of explicit object structure for adapting demonstrated interactions to new objects. WM selection provides further gains of 2.0–5.5 percentage points when SOP context is available, indicating that predicted outcomes help identify experience suitable for the target geometry and motion constraints.

The final configuration additionally uses world-model rollouts and progress rewards to revise the adapted skills' control parameters or code before execution. This refinement improves success in every setting, with the largest gains on unseen categories and geometry–kinematics OOD. Success increases from 42.4% to 52.8% on unseen categories and from 36.3% to 48.6% under OOD shifts. These results support combining structural context with predictive selection and skill refinement: SOPs help the agent interpret and adapt prior experience, while world-model feedback guides its selection and further adjustment without updating the code-agent parameters.

6 Conclusion

We presented CTRL-Z, a closed-loop framework that enables code agents to evaluate and revise action programs before execution. Structured Object Programs connect programmatic reasoning with a dual-stream world model through rendered geometry and decoded object states. World-model rollouts further guide skill reuse and refinement and provide object-manipulation guidance for agent. Experiments demonstrate improved manipulation success, generalization, and self-improvement. Future work will explore longer-horizon task, more diverse manipulation tasks, and more effective skill transfer across objects and environments.

AI USE STATEMENT

In this work, we used generative AI tools for no tasks with required disclosure. We have not used generative AI tools to generate synthetic datasets, develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in writing proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, or interpret results, and translation, dataset cleaning and reformatting, and qualitative or thematic data analysis are not applicable to this work. Our use of generative AI tools was limited to polishing the manuscript's language, including grammar, wording, and readability. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI

REFERENCES

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. *arXiv preprint arXiv:2204.01691*, 2022. URL https://arxiv.org/abs/2204.01691.

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. *arXiv preprint arXiv:2506.09985*, 2025. URL https://arxiv.org/abs/2506.09985.

Aidan Curtis, Nishanth Kumar, Jing Cao, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Trust the PRoC3S: Solving Long-Horizon Robotics Problems with LLMs and Constraint Satisfaction. *arXiv preprint arXiv:2406.05572*, 2024. URL https://arxiv.org/abs/2406.05572.

Frederik Ebert, Chelsea Finn, Sudeep Dasari, Annie Xie, Alex Lee, and Sergey Levine. Visual Foresight: Model-Based Deep Reinforcement Learning for Vision-Based Robotic Control. *arXiv preprint arXiv:1812.00568*, 2018. URL https://arxiv.org/abs/1812.00568.

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 "Jim" Fan. CaP-X: A Framework for Benchmarking and Improving Coding Agents for Robot Manipulation. *arXiv preprint arXiv:2603.22435*, 2026. URL https://arxiv.org/abs/2603.22435.

Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. PDDLStream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1):440–448, 2020. doi: 10.1609/icaps.v30i1.6739. URL https://ojs.aaai.org/index.php/ICAPS/article/view/6739.

Qiao Gu, Alihusein Kuwajerwala, Sacha Morin, Krishna Murthy Jatavallabhula, Bipasha Sen, Aditya Agarwal, Corban Rivera, William Paul, Kirsty Ellis, Rama Chellappa, Chuang Gan, Celso Miguel de Melo, Joshua B. Tenenbaum, Antonio Torralba, Florian Shkurti, and Liam Paull. ConceptGraphs: Open-Vocabulary 3D Scene Graphs for Perception and Planning. *arXiv preprint arXiv:2309.16650*, 2023. URL https://arxiv.org/abs/2309.16650.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner Monologue: Embodied Reasoning through Planning with Language Models. *arXiv preprint arXiv:2207.05608*, 2022. URL https://arxiv.org/abs/2207.05608.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Fei-Fei Li. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models. *arXiv preprint arXiv:2307.05973*, 2023. URL https://arxiv.org/abs/2307.05973.

Wenlong Huang, Chen Wang, Yunzhu Li, Ruohan Zhang, and Fei-Fei Li. ReKep: Spatio-Temporal Reasoning of Relational Keypoint Constraints for Robotic Manipulation. *arXiv preprint arXiv:2409.01652*, 2024. URL https://arxiv.org/abs/2409.01652.

R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. ShapeAssembly: Learning to generate programs for 3D shape structure synthesis. *ACM Transactions on Graphics*, 39(6), 2020. URL https://rkjones4.github.io/shapeAssembly.html.

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. *arXiv preprint arXiv:2209.07753*, 2022. URL https://arxiv.org/abs/2209.07753.

Lorenzo Mur-Labadia, Matthew Muckley, Amir Bar, Mido Assran, Koustuv Sinha, Mike Rabbat, Yann LeCun, Nicolas Ballas, and Adrien Bardes. V-JEPA 2.1: Unlocking Dense Features in Video Self-Supervised Learning. *arXiv preprint arXiv:2603.14482*, 2026. URL https://arxiv.org/abs/2603.14482.

Mingyang Sun, Jiude Wei, Qichen He, Donglin Wang, Cewu Lu, and Jianhua Sun. Executable Analytic Concepts as the Missing Link Between VLM Insight and Precise Manipulation. *arXiv preprint arXiv:2510.07975*, 2025. URL https://arxiv.org/abs/2510.07975.

Jiude Wei, Yuxuan Li, Cewu Lu, and Jianhua Sun. Physically Ground Commonsense Knowledge for Articulated Object Manipulation with Analytic Concepts. *arXiv preprint arXiv:2503.23348*, 2025. URL https://arxiv.org/abs/2503.23348.

Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, and Hao Su. SAPIEN: A SimulAted Part-based Interactive ENvironment. *arXiv preprint arXiv:2003.08515*, 2020. URL https://arxiv.org/abs/2003.08515.

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. *arXiv preprint arXiv:2602.15922*, 2026. URL https://arxiv.org/abs/2602.15922.

Yunzhi Zhang, Zizhang Li, Matt Zhou, Shangzhe Wu, and Jiajun Wu. The Scene Language: Representing Scenes with Programs, Words, and Embeddings. *arXiv preprint arXiv:2410.16770*, 2024. URL https://arxiv.org/abs/2410.16770.

Gaoyue Zhou, Hengkai Pan, Yann LeCun, and Lerrel Pinto. DINO-WM: World Models on Pre-trained Visual Features enable Zero-shot Planning. *arXiv preprint arXiv:2411.04983*, 2024. URL https://arxiv.org/abs/2411.04983.

Table 3: Real-world task success rates (%, ↑). Each method is evaluated over 100 trials per task: 10 initial pose configurations with 10 trials each.

Method	Block stacking	Drawer opening
$\pi_{0.5}$	56	54
Fast-WAM	47	42
CTRL-Z (ours)	**69**	**63**

A Appendix

A.1 Real-World Manipulation

Tasks and evaluation. We evaluate CTRL-Z on two real-world manipulation tasks: block stacking and drawer opening. The stacking task requires placing the green block on the red block, followed by the blue block on top. The drawer-opening task requires grasping the upper handle and pulling the drawer outward. We compare our framework with $\pi_{0.5}$ and Fast-WAM. For each method and task, we test 10 initial pose configurations with 10 trials per configuration, resulting in 100 trials. We report success rates aggregated over these trials.

Results. As shown in Table 3, CTRL-Z achieves the highest success rate on both tasks. It reaches 69% on block stacking and 63% on drawer opening, exceeding $\pi_{0.5}$ by 13 and 9 percentage points and Fast-WAM by 22 and 21 percentage points, respectively. These results support the applicability of our framework to physical manipulation across sequential object placement and constrained articulated motion.

Qualitative examples. Figures 5 and 6 pair recorded robot observations with illustrative agent dialogue and API pseudocode. Structural grounding provides object geometry and interaction targets, while world-model rollouts and the physical readout provide predicted object states for SOP-based action evaluation. After execution, new observations update the SOP and guide subsequent actions. In the stacking example, the updated configuration provides the placement target for the next block. In the drawer example, the observed drawer motion updates the handle pose and informs the next pulling action. The pseudocode abbreviates history management and candidate selection; `z_hat` denotes the predicted RGB and geometric latents, and P_t denotes the current action program.

A.2 Simulation Platform and Task Setup

We build a robotic manipulation platform using SAPIEN and object assets from PartNet-Mobility (Xiang et al., 2020). The platform integrates physical simulation, visual observations, robot control, and object-state recording with our SOP construction and annotation pipeline. PartNet-Mobility supplies object geometry and mobility annotations, while SOPs associate these assets with compositional geometric descriptions and executable manipulation knowledge.

Our overall evaluation covers three task groups: articulated-object opening, pick-and-place, and precision manipulation. Opening tasks require changing a selected part's configuration, such as rotating a door or translating a drawer. Pick-and-place tasks require transporting an object to a specified destination. Precision tasks require satisfying task-specific alignment or placement conditions. Task success is evaluated from the environment state using the corresponding goal conditions, rather than from world-model predictions.

The notation follows the main text: I_t denotes the RGB observation, q_t the robot joint positions, x_t the robot state, and s_t the dynamic object state. For the code-generated control interface in Algorithm 1, a_t specifies a joint delta-position command executed through the PD controller. Structural grounding constructs the current SOP from observations, as detailed in Appendix A.6.

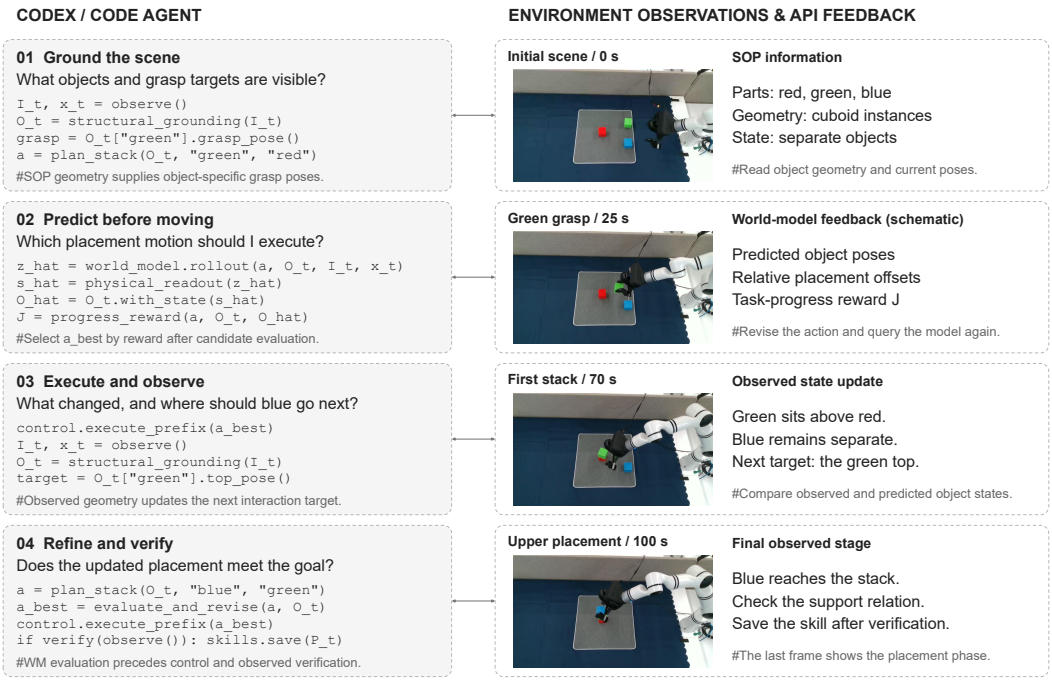


Figure 5: Real-world block stacking: green on red, followed by blue on top. Recorded keyframes and illustrative API pseudocode show structural grounding, predictive evaluation, execution, and observation-based updates.

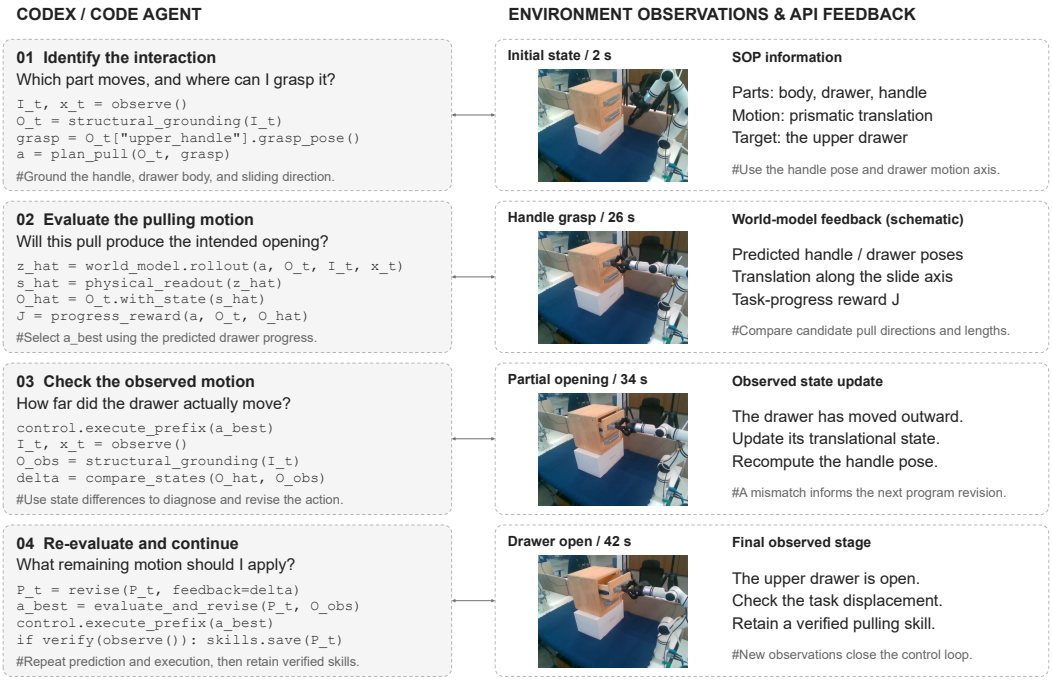


Figure 6: Real-world drawer opening. Recorded keyframes and illustrative API pseudocode show handle grounding, predictive evaluation of pulling actions, and updates from observed motion. World-model feedback is schematic.

A.3 SOP Data Collection and Annotation

We construct a data collection and annotation pipeline on this platform to associate object structure, visual observations, robot actions, and physical state changes within a shared SOP representation. The pipeline separates object-level structural annotations from time-varying interaction records.

Object structure and template annotation. For each object, the SOP annotation identifies its semantic parts and associates each part P_i with a reusable template τ_i, geometric parameters Θ_i, affordances A_i, and manipulation procedures F_{τ_i}. Templates compose geometric primitives, including cuboids and cylinders, to describe part geometry. The kinematic description $\mathcal{K}$ records part relationships and the applicable joint types, axes, and limits. These annotations determine the static structure of $\mathcal{O}_t$, while s_t specifies its current pose and configuration.

The annotations also associate SOP parts with their corresponding simulated components. This correspondence allows recorded physical states to update the appropriate SOP parameters while preserving part identities throughout an interaction. An object with multiple doors, drawers, or handles can reuse the same templates through different instance parameters and part compositions.

Interaction recording and geometric rendering. During data collection, the platform records RGB observations I_t, robot actions a_t, robot states, and the corresponding dynamic object states. The SOP renderer uses the instantiated geometry and its current configuration to produce $G_t = \mathcal{R}(\mathcal{O}_t)$. The RGB and geometric streams are associated with the same interaction time steps, providing paired visual inputs for action-conditioned prediction.

Physical annotations include part positions $\boldsymbol{p}_{t,i}$, unit quaternions $\boldsymbol{u}_{t,i}$, part-validity labels $v_{t,i}$, and target-part labels $y_{t,i}$, following Appendix A.7. These labels connect the visual sequences to the part-level quantities supervised by D_ρ. The resulting records support both latent prediction and physical-state supervision.

Training annotations and inference inputs. Reference physical states provide supervision for world-model training and ground truth for evaluation. During standard agent evaluation, the current SOP is estimated through structural grounding. Future candidate SOP states are obtained from F_θ and D_ρ. Ground-truth feedback is exposed only in the explicitly designated oracle configurations.

A.4 Structured Object Programs as Compact Object and Interaction Priors

Fig. 7 illustrates this representation using a microwave, showing its structural templates (left), the assembled object and its constituent semantic parts (center), and manipulation templates (right).

Structured Object Programs describe objects through reusable, parameterized programs. Each Program associates a semantic part with a template for constructing its geometry and functions for computing structure-dependent interaction information. Object program factory demonstrates how objects can be composed from parameterized program templates and how knowledge defined on a template can be transferred to its instances. Manipulation program further associate these structures with grasp poses, force directions, and executable manipulation procedures. The renderer program can render objects into geometric views based on their instances and physical parameters. We adopt this representation as a shared interface for code agents and our world model.

The microwave illustration provides a concrete example. The object is assembled from a body, a `Sunken_door`, and a `Trifold_handle`. Their structural templates specify dimensions, offsets, and how primitive shapes are assembled. The kinematic description specifies the door's hinge and the handle's rigid attachment to the door. Once the geometric parameters and current object state are specified, the instantiated templates generate the geometry for rendering and determine the part poses used to compute grasp poses and interaction directions. A manipulation template such as `grasp_pose(offset)` converts a grasp location in the handle's local frame into a world-frame gripper pose. For example,

$$T_{WG} = T_{WB}\, T_{BD}(\theta_{\text{door}})\, T_{DH}\, T_{HG}(\boldsymbol{\xi}), \tag{7}$$

where W, B, D, H, and G denote the world, body, door, handle, and gripper frames, respectively, T_{AB} maps coordinates from frame B to frame A, and $\boldsymbol{\xi}$ parameterizes the local grasp. The hinge

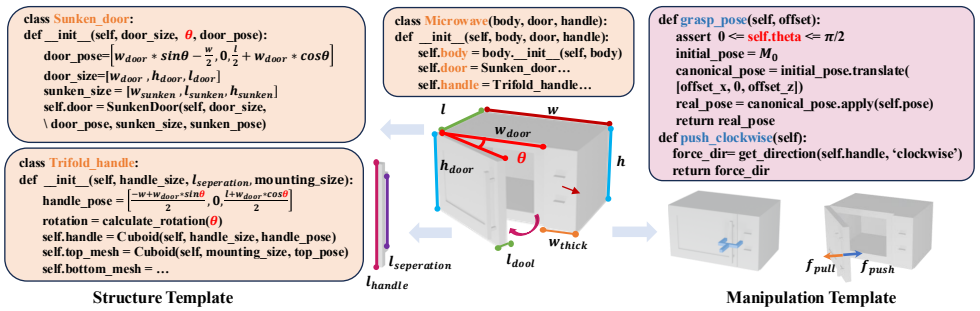


Figure 7: Structured Object Programs for articulated manipulation. Left shows two object program templates. The structure template defines the object geometry and kinematic parameters, such as the door angle θ and key size parameters. The manipulation template turns the object instance into executable actions by computing a grasp pose and choosing pull or push forces.

geometry determines T_{BD}, while T_{DH} remains fixed for a rigidly attached handle. Changing the door angle therefore updates the handle and grasp poses consistently. Push/pull routines similarly compute interaction directions from the instantiated geometry and joint configuration.

For a code-as-policy agent, these templates supply semantic priors about which parts can be referenced, geometric priors about their shapes and poses, kinematic priors about their connections and allowed motion, and interaction priors about where and how to act. The agent can select a part and an interaction routine, vary its parameters, and use the returned parameters in perception and control calls. It can also express a task goal as a numerical condition, such as whether the door angle reaches a target within tolerance. These calculations require grounded instance parameters and consistent coordinate frames.

Code-Listing 1: Shared part templates with different instance compositions. Original template code constructs both doors and handles from cuboids. Selected official annotations describe a microwave with one door and one handle, and a refrigerator with two doors and two handles. The same templates are instantiated with different dimensions and positions. The original grasp computation uses handle geometry and pose, supporting reuse across compatible instances of the same handle template. Code and instance annotations are from Structured Object Program. Only selected source lines and annotation fields are shown.

Shared templates: cuboidal doors and handles

```
class Cuboidal_Door(ConceptTemplate):
    def __init__(self, size, position = [0, 0, 0], rotation = [0, 0, 0]):
        self.mesh = Cuboid(size[1], size[0], size[2],
                           position = mesh_position)
class Cuboidal_Handle(ConceptTemplate):
    def __init__(self, size, position = [0, 0, 0], rotation = [0, 0, 0]):
                               position = mesh_position)
        vertices_list.append(self.mesh.vertices)
```

The same door and handle template implementations also appear in the micorwave and refrigerator library. A door or handle instance supplies its dimensions and pose; the template generates the corresponding geometry.

(a) Microwave: one door and one handle

```
# Official instance
# Selected records from the object object program list.
[
    {"template": "Cuboidal_Door", "parameters": {
        "size": [1.59, 0.72, 0.005],
        "position": [0.018, 0.0, 0.545],
        "rotation": [0, 0, 0]}},

    {"template": "Cuboidal_Handle", "parameters": {
```

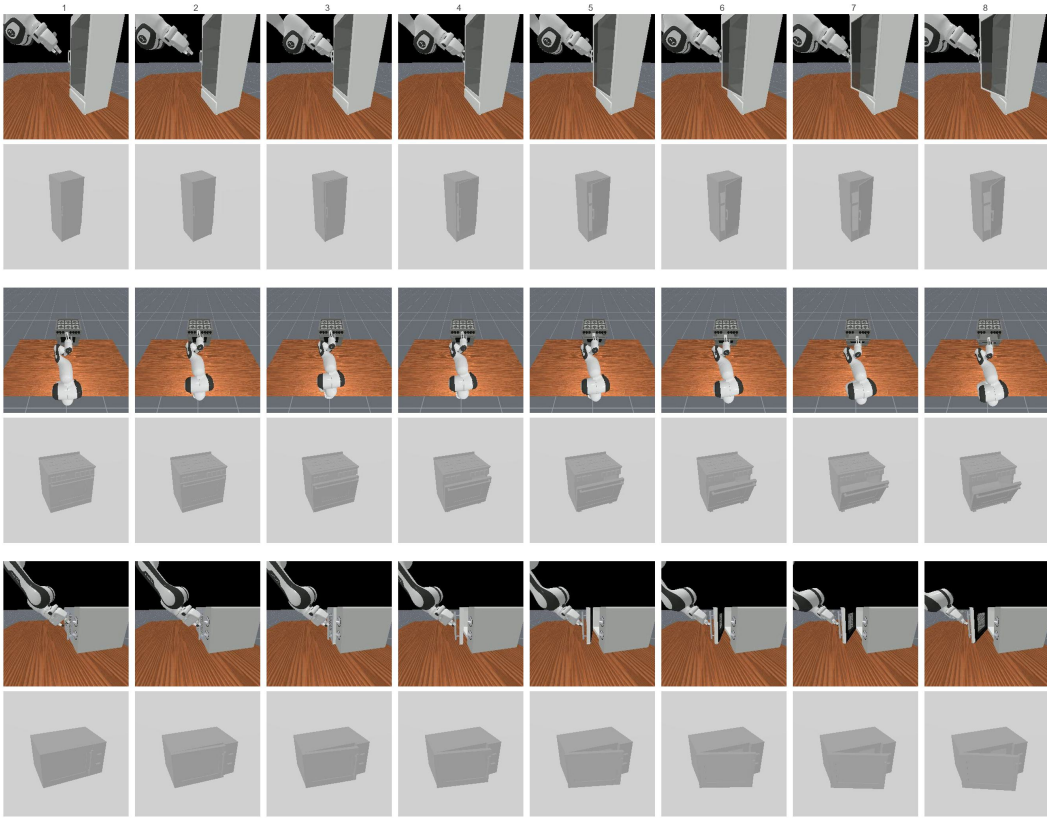


Figure 8: RGB observations and corresponding geometric views rendered from Structured Object Programs during cabinet, dishwasher, and microwave manipulation. For each object, the upper row shows RGB observations, and the lower row shows program-rendered geometry with predicted object dynamic parameters at the corresponding time steps, illustrating changes in object configuration.

```
        "size": [0.07, 0.035, 0.035],
        "position": [0.43800000000000006, 0.025, 0.55],
        "rotation": [0, 0, 0]}}
]
```

(b) Refrigerator: two doors and two handles

```
# Official instance
# Selected records from the object program list.
[
    {"template": "Cuboidal_Door", "parameters": {
        "size": [0.66, 0.91, 0.05],
        "position": [0.0, 0.48, 0.255],
        "rotation": [0, 0, 0]}},

    {"template": "Cuboidal_Handle", "parameters": {
        "size": [0.06, 0.4, 0.03],
        "position": [-0.24, 0.19000000000000003, 0.305],
        "rotation": [0, 0, 0]}},

    {"template": "Cuboidal_Door", "parameters": {
        "size": [0.66, 0.91, 0.05],
        "position": [0.0, -0.43000000000000005, 0.255],
        "rotation": [0, 0, 0]}},

    {"template": "Cuboidal_Handle", "parameters": {
        "size": [0.06, 0.4, 0.03],
        "position": [-0.24, -0.43000000000000005, 0.305],
        "rotation": [0, 0, 0]}}
]
```

The refrigerator repeats the same door and handle templates at two different vertical positions. The second door requires another parameterized instance, rather than another construction rule. The microwave uses these templates once, with different dimensions and positions. Other annotated parts are omitted here to emphasize this difference in composition.

(c) Reusable grasp computation for cuboidal handles

```
def grasp_poses(obj, num):
    grasp_poses = []

    if (isinstance(obj, Cuboidal_Handle)):
        RT = transformation_matrix(obj.position, obj.rotation)

        for height_ratio in np.random.random(num):
            init_pose = np.eye(4)
            init_pose[:3, -1] = np.array([0,
                                          (height_ratio - 0.5) * obj.size[1],
                                          obj.size[2]])
            grasp_pose = RT @ init_pose
            grasp_poses.append(grasp_pose)
```

The cuboidal-handle branch computes local grasp positions from `obj.size` and transforms them using `obj.position` and `obj.rotation`. The same geometric computation can therefore be applied to the microwave handle and to either refrigerator handle, after aligning their part-type interfaces and pose conventions. The instance parameters change the resulting grasp poses; the underlying computation remains the same.

The representation is compact because shared rules are stored in the template library, while an object instance specifies the selected templates, their independent geometric parameters, primitives relationships, and current state. Under a fixed-body, rigid-door model with known geometry and hinge parameters, updating a single door angle is sufficient to regenerate the door and handle configuration. Likewise, a manipulation routine and a few arguments describe a family of interactions. An example is provided in Code-Listing 1, where a microwave and a refrigerator reuse the same part templates with different instance counts and geometric parameters.

In our framework, the code agent uses the instantiated Structured Object Program to formulate candidate actions, while the renderer produces the geometric view $G_t = \mathcal{R}(\mathcal{O}_t)$, Fig. 8 shows examples of RGB observations and their corresponding Program-rendered geometric views across successive states of cabinet, dishwasher, and microwave manipulation. The world model receives this view together with RGB observations and candidate actions, and predicts future visual and geometric latents. A physical readout decodes these predictions into dynamic part parameters that the agent can inspect through the Structured Object Program interface and compare with the task goal. Predicted parameters describe hypothetical future configurations for action evaluation. After short-horizon execution, new observations update the current Structured Object Program state and its rendered view for the next decision. Thus, Structured Object Program specify the structure, interaction priors, and state variables available and readable to the agent program, while the world model estimates the physical consequences of candidate actions.

SOP-based reward computation. SOPs expose deterministic reward routines that the agent can invoke to evaluate candidate interactions. The world model predicts dynamic object states, and the reward routines compute geometric quantities from the corresponding SOP instances. We distinguish reaching objectives from object-motion progress objectives. The relevant semantic part P_i is selected according to the objective, such as a handle for reaching or a door for opening progress.

Reaching reward. Let $T_{Wi}(s)$ denote the world-frame pose of the selected target part, computed from its SOP geometry, kinematic relationships $\mathcal{K}$, and dynamic state s. We use $\mathrm{pos}(\cdot)$ to extract the translation component of a pose. For the joint delta-position commands used in Algorithm 1, the nominal robot configuration and commanded TCP pose are

$$\hat{q}_{t+k} = \hat{q}_{t+k-1} + a_{t+k-1}, \qquad \hat{x}_{t+k} = \mathrm{FK}(\hat{q}_{t+k}), \qquad \hat{q}_t = q_t. \tag{8}$$

Here, q_t denotes robot joint positions, a_t denotes joint delta position and x_t denotes the TCP pose, consistent with the control interface. The reaching reward is

$$\begin{aligned} r_L^{\mathrm{reach}}&(a_{t+k-1}, \hat{s}_{t+k-1}, \hat{s}_{t+k}) \\ &= -\left\|\mathrm{pos}(\hat{x}_{t+k}) - \mathrm{pos}\left(T_{Wi}(\hat{s}_{t+k})\right)\right\|_2 . \end{aligned} \tag{9}$$

Both positions are expressed in the same world frame. When the interaction requires a specific grasp location, the target part pose is replaced by the grasp pose T_{WG} computed through F_{τ_i}, as illustrated above. A higher reward therefore indicates a smaller distance to the selected interaction target. The TCP pose is derived from the candidate command; the target pose is obtained from the predicted SOP.

Object-motion progress reward. Progress is measured relative to the initial object state s_0. Let $\mathrm{angle}_i(s)$ denote the angular coordinate of the selected moving part about its specified rotation axis, and let $\mathrm{disp}_i(s)$ denote its translational coordinate along the specified motion direction. These quantities are queried from the SOP using $\mathcal{K}$ and s. The progress reward is

$$r_L^{\mathrm{progress}}(a_{t+k-1}, \hat{s}_{t+k-1}, \hat{s}_{t+k}) = \begin{cases} \mathrm{angle}_i(\hat{s}_{t+k}) - \mathrm{angle}_i(s_0), & \text{rotational objective,} \\ \mathrm{disp}_i(\hat{s}_{t+k}) - \mathrm{disp}_i(s_0), & \text{translational objective.} \end{cases} \tag{10}$$

The coordinates are oriented according to the requested motion, so movement opposite to the task direction produces negative progress. For example, door opening is evaluated through its angular change, whereas drawer opening is evaluated through its displacement along the sliding axis. The reference s_0 remains fixed throughout the task, so this reward measures progress from the initial configuration rather than only the change between adjacent prediction steps.

The agent invokes the reward routine corresponding to the current objective, giving r_L in the trajectory return $J_t(\mathbf{a}_t)$. The same computations can be applied to observed SOP states after execution.

A.5 MODEL PSEUDOCODE

Algorithm 1: World-Model-Guided Code Agent via Structured Object Programs

Require: Instruction L, initial observation $o_0 = (I_0, x_0)$, initial robot joint positions q_0, horizon H, history length h
Require: Code agent $\mathcal{G}_\phi$, world model F_θ, physical readout D_ρ, perception and policy APIs
Require: Mode $\in$ {planning, offline}
Structured Object Program: initialize physical priors -

$\mathcal{O}_0 \leftarrow \mathrm{StructuralGrounding}(I_0)$ ▷ Select τ_i; estimate Θ_i, $\mathcal{K}$, s_0
$G_0 \leftarrow \mathcal{R}(\mathcal{O}_0)$
Initialize latent histories with $z_0 = E_\psi(I_0)$ and z_0^{geom} encoded from G_0
Initialize $\mathcal{H}_0$, predictive feedback, and $t \leftarrow 0$
while task unfinished and budget remains **do**
    Load the persistent skill library and associated successful action programs and execution traces
Code Agent: generate candidate actions -

    Expose $\mathcal{O}_t$, affordances A_i, and manipulation templates F_{τ_i} to the agent
    Retrieve skills and successful examples with SOP structures similar to $\mathcal{O}_t$
    Generate or revise P_t using $\mathcal{G}_\phi$, L, available history, retrieved examples, predictive feedback, and previous execution traces
    Propose candidate sequences $\mathbf{a}_t = (a_t, \ldots, a_{t+H-1})$ through code or policy APIs ▷ PD joint delta-position commands
    for each candidate $\mathbf{a}_t$ **do**
        Copy current latent histories; $\hat{q}_t \leftarrow q_t$, $\hat{x}_t \leftarrow x_t$, $\hat{s}_t \leftarrow s_t$, $J_t(\mathbf{a}_t) \leftarrow 0$
        for $k = 0, \ldots, H-1$ **do**
World Model: predict physical effects -

            Predict $(\hat{z}_{t+k+1}, \hat{z}^{\mathrm{geom}}_{t+k+1})$ with F_θ using the latest h latent steps, $\hat{x}_{t+k}$, and a_{t+k}
            $\hat{s}_{t+k+1} \leftarrow D_\rho(\hat{z}_{t+k+1}, \hat{z}^{\mathrm{geom}}_{t+k+1}; \mathcal{O}_t)$
            Append both predicted latents to this candidate's history ▷ Autoregressive feedback
Structured Object Program: update and score -

            $\hat{\mathcal{O}}_{t+k+1} \leftarrow (c, \{P_i\}_{i=1}^{N_p}, \mathcal{K}, \hat{s}_{t+k+1})$
            $\hat{q}_{t+k+1} \leftarrow \hat{q}_{t+k} + a_{t+k}$; $\hat{x}_{t+k+1} \leftarrow \mathrm{FK}(\hat{q}_{t+k+1})$
            $\hat{r}_{t+k} \leftarrow r_L(a_{t+k}, \hat{s}_{t+k}, \hat{s}_{t+k+1})$ ▷ Robot target pose and predicted part poses
            $J_t(\mathbf{a}_t) \leftarrow J_t(\mathbf{a}_t) + \hat{r}_{t+k}$
        end for
        Cache the candidate's predicted latents, object instances, joint targets, and rewards
    end for
Code Agent: use predictive feedback -

    Inspect predicted SOP-state changes and rewards to identify ineffective operations and refine P_t or the retrieved other skills
    Use predicted instances and rewards to revise action programs or optimize invoked policies
    Re-evaluate any revised candidates before selection
    Select the evaluated sequence with the highest $J_t(\mathbf{a}_t)$ and retain its predicted trajectory
Closed-loop update: planning or offline -

    if mode is planning **then**
        Execute the selected first action a_t through the PD joint controller
        Read q_{t+1} and I_{t+1}; $x_{t+1} \leftarrow \mathrm{FK}(q_{t+1})$
        Update $\mathcal{H}_{t+1}$ with a_t and $o_{t+1} = (I_{t+1}, x_{t+1})$
        Re-estimate $\mathcal{O}_{t+1}$ from I_{t+1}; $G_{t+1} \leftarrow \mathcal{R}(\mathcal{O}_{t+1})$
        Append encoded I_{t+1} and G_{t+1} to the current latent histories
        Record executed operations, tool outputs, execution status, and observed state changes
        Compare $\mathcal{O}_{t+1}$ with the selected $\hat{\mathcal{O}}_{t+1}$; retain discrepancies and failure traces for subsequent programs repair
        if environment execution confirms a successful skill or task **then**
            Consolidate successful routines and repairs into reusable skills
            Store the skills with their SOP structures, application conditions, and successful execution traces
        end if

else
    Advance to the selected prediction: $\mathcal{O}_{t+1} \leftarrow \hat{\mathcal{O}}_{t+1}, q_{t+1} \leftarrow \hat{q}_{t+1}, x_{t+1} \leftarrow \hat{x}_{t+1}$
    Append $(\hat{z}_{t+1}, \hat{z}_{t+1}^{\text{geom}})$ to the current latent histories
    Retain static object structure and real initial observation history; store predicted object states and rewards as feedback to optimize the policies.
end if
Keep the latest h latent steps; $t \leftarrow t + 1$
end while

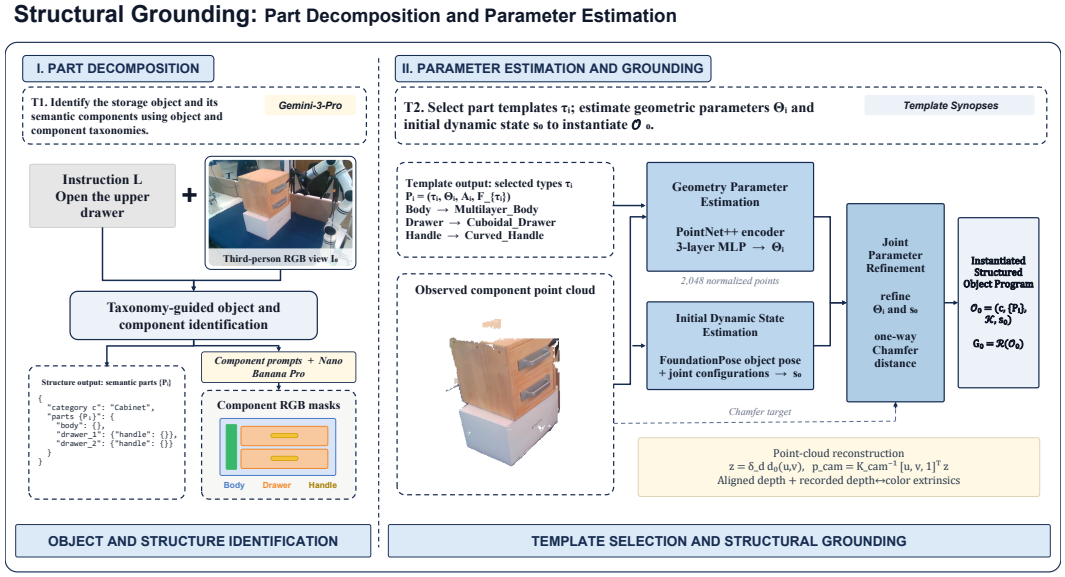


Figure 9: Overview of the structural grounding module. Given an instruction and an initial observation, the module decomposes the target object into semantic parts, selects reusable templates, estimates geometric parameters and the initial dynamic state, and instantiates the Structured Object Program representation $\mathcal{O}_0$.

A.6 Structural Grounding

As summarized in Fig. 9, structural grounding converts the initial observation into an Structured Object Program instance with explicit semantic parts, geometric parameters, kinematic relationships, and dynamic state. Given an instruction L, an initial RGB observation I_0, and calibrated depth d_0 when available, the module constructs

$$\mathcal{O}_0 = \left(c, \{P_i\}_{i=1}^{N_p}, \mathcal{K}, s_0\right), \qquad P_i = (\tau_i, \Theta_i, A_i, F_{\tau_i}), \tag{11}$$

and produces the corresponding geometric view

$$G_0 = \mathcal{R}(\mathcal{O}_0). \tag{12}$$

The grounding procedure consists of part decomposition, template selection, geometry estimation, initial-state estimation, and an optional alignment refinement.

Part decomposition. The first stage identifies the task-relevant object and decomposes it into semantic parts. A multimodal identification module uses the instruction L, the RGB observation I_0, and object and component taxonomies to predict the object category c, the number of parts N_p, and their parent–child organization. For each semantic part, a prompt-conditioned segmentation module produces a color-coded mask M_i.

For example, a storage cabinet may be decomposed into a body, drawers, and handles, with each handle associated with its corresponding drawer. This stage determines the semantic identities and observed regions of the parts, but does not yet instantiate their complete geometric and dynamic parameters.

Template selection and structural priors. Each detected part is matched to a reusable template type τ_i from the Structured Object Program library. The matching module compares the semantic description and observed shape of the part with the synopses of candidate templates.

Selecting τ_i determines the admissible parameterization of Θ_i and retrieves the template-associated affordances A_i and manipulation templates F_{τ_i}. The part hierarchy, together with the selected template priors, is also used to instantiate the kinematic structure $\mathcal{K}$. The latter contains the applicable joint topology, joint types, axes, and motion limits. The precise fields stored in $\mathcal{K}$ depend on the selected templates and the available observations.

Component point-cloud reconstruction. The main framework uses I_0 to denote the RGB observation. When aligned depth is available, we denote it separately by d_0 and use it as an auxiliary perception signal for geometric grounding.

Let δ_d denote the metric depth scale and $\mathbf{K}_{\mathrm{cam}}$ the camera intrinsic matrix. For a valid pixel (u, v), its metric depth and camera-frame position are computed as

$$\zeta_0(u,v) = \delta_d d_0(u,v), \qquad \mathbf{p}_{\mathrm{cam}}(u,v) = \zeta_0(u,v)\mathbf{K}_{\mathrm{cam}}^{-1}\begin{bmatrix} u & v & 1 \end{bmatrix}^\top . \tag{13}$$

If a different reference frame is required, the corresponding calibrated rigid transformation is applied after back-projection.

The observed point set associated with part i is

$$\mathcal{X}_i = \{\mathbf{p}_{\mathrm{cam}}(u,v) \mid M_i(u,v) = 1,\ d_0(u,v) > 0\} . \tag{14}$$

Invalid depth measurements and isolated outliers are removed before the point set is passed to the parameter-estimation modules. In the configuration illustrated in Fig. 9, each component point cloud is normalized and sampled to $2{,}048$ points. The normalization parameters are retained so that the predicted geometry can subsequently be restored to metric scale.

Template-conditioned geometry estimation. A template-conditioned point-set estimator predicts the instance-specific geometric parameters

$$\widehat{\Theta}_i = f_{\mathrm{geom}}\left(\overline{\mathcal{X}}_i, \tau_i\right), \tag{15}$$

where $\overline{\mathcal{X}}_i$ is the normalized point set. A practical realization uses a PointNet++-style point encoder followed by a lightweight multilayer perceptron. The selected template τ_i determines the output dimension and the interpretation of the predicted variables. Depending on the template, these variables may describe dimensions, radii, curvature, thickness, or relative attachment offsets.

The estimator can use either template-specific prediction heads or a shared decoder conditioned on τ_i. This implementation choice does not change the representation in Eq. (11). After estimation, the recorded normalization transform restores $\widehat{\Theta}_i$ to the scale of the observed scene.

Initial dynamic-state estimation. The initial dynamic state s_0 contains the object part pose and the part configurations associated with $\mathcal{K}$. For explanatory purposes, it can be locally decomposed as

$$\widehat{s}_0 = \left(\widehat{\boldsymbol{\xi}}_0^{\mathrm{obj}}, \widehat{\mathbf{q}}_0\right), \tag{16}$$

where $\widehat{\boldsymbol{\xi}}_0^{\mathrm{obj}}$ denotes an initial object base pose estimate and $\widehat{\mathbf{q}}_0$ denotes the initial joint configuration.

The object base pose can be initialized using an off-the-shelf pose-estimation backend conditioned on the observation and the instantiated template geometry. The joint configuration is initialized using the observed arrangement of the articulated parts and the constraints encoded by $\mathcal{K}$. We do not require a particular pose or joint-state estimator. Importantly, individual part poses are computed from $\mathcal{K}$ and s_0, rather than represented as independent variables inside P_i.

Optional grounding refinement. The initial geometry and state estimates can optionally be refined by aligning the instantiated Structured Object Program with the observed component point clouds. Let

$$\widehat{\mathcal{X}}_i\left(\tau_i, \Theta_i, \mathcal{K}, s_0\right) \tag{17}$$

denote surface points generated from the current instance of part i. We use the following one-way point-set alignment term:

$$d_{\mathrm{CD}}^{\rightarrow}\left(\mathcal{X}_i, \widehat{\mathcal{X}}_i\right) = \frac{1}{|\mathcal{X}_i|}\sum_{\mathbf{x}\in\mathcal{X}_i}\min_{\widehat{\mathbf{x}}\in\widehat{\mathcal{X}}_i}\|\mathbf{x}-\widehat{\mathbf{x}}\|_2^2 . \tag{18}$$

The asymmetric direction evaluates whether the observed surface is explained by the instantiated geometry without strongly penalizing template regions that are occluded in the input view.

A generic refinement objective is

$$\min_{\{\Theta_i\}_{i=1}^{N_p},\, s_0} \quad \sum_{i=1}^{N_p} d_{\mathrm{CD}}^{\rightarrow}\left(\mathcal{X}_i, \widehat{\mathcal{X}}_i\left(\tau_i, \Theta_i, \mathcal{K}, s_0\right)\right) + \lambda_{\mathrm{prior}}\Omega\left(\{\Theta_i\}_{i=1}^{N_p}, \mathcal{K}, s_0\right). \tag{19}$$

where Ω encourages template-valid parameter ranges and kinematically valid states. The template identities τ_i, affordances A_i, and manipulation templates F_{τ_i} remain fixed during this stage.

The exact optimizer, update schedule, and stopping criterion are implementation choices. Refinement may also be omitted when the initial estimates already provide sufficient alignment or when the available depth is unreliable.

Structured Object Program instantiation. After grounding, the final parameters are absorbed into the notation Θ_i and s_0, and the semantic parts are instantiated as

$$P_i = (\tau_i, \Theta_i, A_i, F_{\tau_i}). \tag{20}$$

The complete initial object representation is therefore

$$\mathcal{O}_0 = \left(c, \{P_i\}_{i=1}^{N_p}, \mathcal{K}, s_0\right), \tag{21}$$

and its geometric view is

$$G_0 = \mathcal{R}(\mathcal{O}_0). \tag{22}$$

The instantiated object exposes part geometry, kinematic structure, part poses, affordances, and manipulation priors to downstream components. In particular, G_0 provides the initial input to the geometric stream of the world model, while the manipulation templates F_{τ_i} can be accessed by the code agent when constructing the executable program P_t. Here, P_i always denotes a semantic object part, whereas P_t denotes the executable program generated at interaction step t.

A.7 Dual-Stream World-Model Architecture and Training

This section details the architecture and training objectives of the action-conditioned predictor F_θ and physical readout D_ρ. We retain the notation introduced in the main text: I_t denotes an RGB observation, $G_t = \mathcal{R}(\mathcal{O}_t)$ denotes the geometric view rendered from the corresponding SOP instance, a_t denotes a robot action, and x_t denotes the robot state used for conditioning. The object dynamic state remains denoted by s_t.

Shared visual encoder. RGB observations and SOP-rendered geometric views are processed by the same frozen V-JEPA 2.1 ViT-Gigantic/16 encoder:

$$z_t = E_\psi(I_t), \qquad z_t^{\text{geom}} = E_\psi(G_t). \tag{23}$$

Thus, E_{geom} in the main text denotes the geometric application of E_ψ, with shared parameters rather than a separately trained encoder. The encoder remains fixed throughout training, including when computing prediction targets; no exponential-moving-average update is applied.

Each input image has a resolution of 384×384 and is divided into 16×16 spatial patches, giving $24 \times 24 = 576$ visual tokens per stream at each time step. The two streams therefore provide 1152 visual tokens per time step. For a batch of clips, the input layout is $[B, 2, 3, T_{\text{clip}}, 384, 384]$, and the encoded features are arranged as $[B, T_{\text{clip}}, 2, 576, d_{\text{enc}}]$, where B is the batch size, T_{clip} is the clip length, and d_{enc} is the encoder feature dimension. Tokens are flattened in temporal, stream, and spatial order, with RGB tokens preceding geometric tokens within each time step.

Action-conditioned prediction. The predictor F_θ is a 24-layer Transformer with hidden dimension 1024 and 16 attention heads. Visual features are projected to the predictor width, and learnable stream embeddings distinguish RGB tokens from geometric tokens. The action a_t and robot state x_t are encoded as separate conditioning tokens. Each temporal block contains, in order, the action token, robot-state token, RGB patch tokens, and geometric patch tokens.

Attention is causal across temporal blocks but unrestricted within each block. Consequently, RGB and geometric tokens at the same time step can attend to one another, while no token can access future temporal blocks. This implements token-level fusion within a shared predictor. The visual outputs retain their stream identities and predict the RGB and geometric latents at the next time step.

Teacher-forced latent prediction. For notational convenience, let $z_t^{\text{rgb}} \equiv z_t$. During teacher forcing, the predictor receives encoded observations from the training sequence:

$$\left(\hat{z}_{t+1}^{\text{TF}}, \hat{z}_{t+1}^{\text{geom,TF}}\right) = F_\theta\left(z_{\le t}, z_{\le t}^{\text{geom}}, x_{\le t}, a_{\le t}\right). \tag{24}$$

Here, the conditioning histories are written explicitly because action and robot-state tokens are retained with their corresponding visual blocks. Each history contains at most the latest h steps.

The teacher-forced objective supervises adjacent time steps throughout the clip:

$$\mathcal{L}_{\text{TF}} = \frac{1}{2(T_{\text{clip}} - 1)} \sum_{t=0}^{T_{\text{clip}}-2} \sum_{\sigma \in \{\text{rgb,geom}\}} \left\| \hat{z}_{t+1}^{\sigma,\text{TF}} - \text{sg}(z_{t+1}^{\sigma}) \right\|_{1,\text{mean}}. \tag{25}$$

The operator sg denotes stop-gradient, and $\|\cdot\|_{1,\text{mean}}$ denotes the mean absolute error over visual tokens and feature dimensions. Losses are also averaged over the batch. This corresponds to the implementation setting `loss_exp=1`. Every teacher-forced prediction therefore uses encoded observations as its visual context.

Autoregressive latent prediction. The autoregressive branch is initialized with only the first encoded observation from each stream:

$$\tilde{z}_0 = z_0, \qquad \tilde{z}_0^{\text{geom}} = z_0^{\text{geom}}. \tag{26}$$

Subsequent predictions are recursively appended to the visual histories:

$$\left(\tilde{z}_{k+1}, \tilde{z}_{k+1}^{\text{geom}}\right) = F_\theta\left(\tilde{z}_{\le k}, \tilde{z}_{\le k}^{\text{geom}}, x_{\le k}, a_{\le k}\right). \tag{27}$$

For K supervised prediction steps, the objective is

$$\mathcal{L}_{\mathrm{AR}} = \frac{1}{2K} \sum_{k=1}^{K} \sum_{\sigma \in \{\mathrm{rgb},\mathrm{geom}\}} \|\tilde{z}_k^{\sigma} - \mathrm{sg}(z_k^{\sigma})\|_{1,\mathrm{mean}} \,. \tag{28}$$

Here, $\tilde{z}_k^{\mathrm{rgb}} \equiv \tilde{z}_k$. The training rollout length K is distinct from the evaluation horizon H.

Only the visual latents are autoregressive in this training branch. Actions and robot states remain the recorded values from the dataset at each step. The objective therefore trains recursive visual prediction under recorded control inputs, rather than a fully free-running rollout of both visual and robot states. Decoded SOP states may be used to render geometric views for inspection, but the autoregressive predictor receives its predicted latents as feedback.

Physical readout. The physical readout D_ρ uses a shared part-pose head, applied separately to the predicted RGB tokens and geometric tokens. For each output part slot, the head produces a three-dimensional position, a quaternion, a part-validity logit, and a target-part logit. The head parameters are shared between the two streams and across the teacher-forced and autoregressive branches.

Let $\boldsymbol{p}_{t,i}$ and $\boldsymbol{u}_{t,i}$ denote the position and unit quaternion of part slot i at time t. We use $\boldsymbol{u}$ for quaternions to distinguish them from the robot joint positions q_t. Let $v_{t,i}$ and $y_{t,i}$ denote the binary validity and target-part labels, respectively. For branch $m \in \{\mathrm{TF}, \mathrm{AR}\}$ and stream $\sigma \in \{\mathrm{rgb}, \mathrm{geom}\}$, the pose loss is

$$\begin{aligned} \mathcal{L}_{\mathrm{pose}}^{m,\sigma} = \mathrm{mean}_{t,i:\, v_{t,i} y_{t,i}=1} \Big[& \mathrm{SmoothL1}\big(\hat{\boldsymbol{p}}_{t,i}^{m,\sigma}, \boldsymbol{p}_{t,i}\big) \\ & + 1 - \big|\big\langle \hat{\boldsymbol{u}}_{t,i}^{m,\sigma}, \boldsymbol{u}_{t,i} \big\rangle\big| \Big]. \end{aligned} \tag{29}$$

Unit quaternions are used in the orientation term, which treats $\boldsymbol{u}$ and $-\boldsymbol{u}$ as equivalent rotations. Pose regression is supervised only for valid target parts. The validity and target-part classification losses, $\mathcal{L}_{\mathrm{valid}}^{m,\sigma}$ and $\mathcal{L}_{\mathrm{target}}^{m,\sigma}$, are supervised over all output part slots.

The physical readout loss for each stream is

$$\mathcal{L}_{\mathrm{phys}}^{m,\sigma} = \mathcal{L}_{\mathrm{pose}}^{m,\sigma} + 0.05\mathcal{L}_{\mathrm{valid}}^{m,\sigma} + 0.05\mathcal{L}_{\mathrm{target}}^{m,\sigma}. \tag{30}$$

The two streams receive equal weight:

$$\mathcal{L}_{\mathrm{phys}}^{m} = \frac{\mathcal{L}_{\mathrm{phys}}^{m,\mathrm{rgb}} + \mathcal{L}_{\mathrm{phys}}^{m,\mathrm{geom}}}{2}. \tag{31}$$

Joint training objective. We jointly train the predictor and physical readout using

$$\mathcal{L} = \mathcal{L}_{\mathrm{TF}} + \mathcal{L}_{\mathrm{AR}} + 0.1\left(\mathcal{L}_{\mathrm{phys}}^{\mathrm{TF}} + \mathcal{L}_{\mathrm{phys}}^{\mathrm{AR}}\right). \tag{32}$$

The latent losses supervise teacher-forced and autoregressive predictions, while the physical losses supervise target-part poses and part masks decoded from the corresponding predicted latents. This corresponds to $\mathcal{L}_{\mathrm{phys}} = \mathcal{L}_{\mathrm{phys}}^{\mathrm{TF}} + \mathcal{L}_{\mathrm{phys}}^{\mathrm{AR}}$, with $\lambda_{\mathrm{TF}} = \lambda_{\mathrm{AR}} = 1$ and $\lambda_{\mathrm{phys}} = 0.1$ in the main-text formulation. Training updates the predictor parameters θ and readout parameters ρ, while keeping the vision encoder parameters ψ fixed.

A.8 World-Model Use for Planning and Policy Optimization

This section details how the code agent uses the world model for action planning and visuomotor policy optimization. We retain the notation introduced in the main text: P_t denotes a control program generated by $\mathcal{G}_\phi$, $\mathcal{O}_t$ denotes the current SOP instance, and $\mathbf{a}_t$ denotes a candidate action sequence. We additionally use π_ω to denote a trainable visuo motor policy invoked through the program, with parameters ω distinct from ϕ, θ, and ρ. Program revision changes P_t, whereas policy training updates ω. The code-agent weights remain frozen in both cases.

Candidate generation and predictive evaluation. At an observed step t, the agent generates candidate programs or program variants using L, $\mathcal{H}_t$, $\mathcal{O}_t$, tool feedback, and retrieved experience. Let $b = 1, \ldots, N_{\text{cand}}$ index the evaluated candidates. Each candidate program $P_t^{(b)}$ produces an action sequence

$$\mathbf{a}_t^{(b)} = \left(a_t^{(b)}, \ldots, a_{t+H-1}^{(b)}\right), \tag{33}$$

either through geometric computations and control routines or through an invoked visuomotor policy. During proposal generation, robot commands are collected for evaluation before being dispatched to the environment. Policy outputs are converted to the action representation and temporal resolution used by the world model.

Each candidate is evaluated from an independent copy of the same initial latent histories and object state:

$$\begin{aligned}&\left(\hat{z}_{t+1:t+H}^{(b)}, \hat{z}_{t+1:t+H}^{\text{geom},(b)}\right)\\ &\qquad = F_\theta\left(z_{t-h+1:t}, z_{t-h+1:t}^{\text{geom}}, x_t, \mathbf{a}_t^{(b)}\right).\end{aligned} \tag{34}$$

The physical readout and SOP update then give

$$\begin{aligned}\hat{s}_{t+k}^{(b)} &= D_\rho\left(\hat{z}_{t+k}^{(b)}, \hat{z}_{t+k}^{\text{geom},(b)}; \mathcal{O}_t\right),\\ \hat{\mathcal{O}}_{t+k}^{(b)} &= \left(c, \{P_i\}_{i=1}^{N_p}, \mathcal{K}, \hat{s}_{t+k}^{(b)}\right), \qquad k = 1, \ldots, H.\end{aligned} \tag{35}$$

The templates, geometric parameters, and kinematic structure remain fixed within each rollout. Predicted dynamic states determine the updated part poses and rendered geometric views. The current observed instance $\mathcal{O}_t$ remains unchanged during candidate evaluation.

The agent evaluates predicted task progress using the same reward function as in the main text:

$$J_t(\mathbf{a}_t^{(b)}) = \sum_{k=1}^{H} r_L\left(a_{t+k-1}^{(b)}, \hat{s}_{t+k-1}^{(b)}, \hat{s}_{t+k}^{(b)}\right), \qquad \hat{s}_t^{(b)} = s_t. \tag{36}$$

The returned SOP instances also allow the agent to inspect which parts moved, compare their configurations with the task goal, and recompute interaction targets through F_{τ_i}. It can revise control parameters or program operations and submit the resulting candidates for another evaluation. This procedure uses the world model as a predictive component within the agent's planning loop.

Action selection and observation feedback. After evaluation and revision, the agent selects

$$b^\star = \underset{1 \le b \le N_{\text{cand}}}{\arg\max}\ J_t(\mathbf{a}_t^{(b)}), \qquad a_t = a_t^{(b^\star)}. \tag{37}$$

In planning mode, only the selected first action is executed before acquiring the next observation. The observation updates $\mathcal{H}_{t+1}$ and $\mathcal{O}_{t+1}$, and its encoded RGB and rendered geometry refresh the latent histories. Subsequent planning therefore starts from the observed outcome rather than the previously predicted state. Execution traces and differences between $\mathcal{O}_{t+1}$ and $\hat{\mathcal{O}}_{t+1}^{(b^\star)}$ provide feedback for the next program revision. This receding-horizon procedure applies to actions generated directly by code and to actions proposed by an invoked visuomotor policy.

World-model-guided VLA post-training. Given a initial observation o_t and instruction L, the behavior policy $\pi_{\omega_{\text{old}}}$ samples N_{cand} action chunks. Each chunk is independently rolled out through F_θ from the same initial histories and SOP. The physical readout D_ρ provides predicted object states, from which the SOP reward routine computes $J_t^{(b)} \equiv J_t(\mathbf{a}_t^{(b)})$. The VLA is queried only at the initial observation, so evaluating the complete chunk does not require intermediate predicted RGB images.

For GRPO, candidate returns define group-relative advantages:

$$\widehat{\text{Adv}}_t^{(b)} = \frac{J_t^{(b)} - \text{mean}_{b'} J_t^{(b')}}{\text{std}_{b'} J_t^{(b')} + \epsilon_{\text{norm}}}. \tag{38}$$

For flow-based policies, we use a stochastic denoising formulation with tractable likelihoods. Let $\ell_\omega^{(b)}$ denote the log probability of the stored denoising trajectory producing candidate b, and define $\eta_t^{(b)}(\omega) = \exp(\ell_\omega^{(b)} - \ell_{\omega_{\text{old}}}^{(b)})$. The policy objective is

$$\max_\omega \mathbb{E}\left[\frac{1}{N_{\text{cand}}} \sum_{b=1}^{N_{\text{cand}}} \min\Big(\eta_t^{(b)}(\omega)\widehat{\text{Adv}}_t^{(b)}, \text{clip}\Big(\eta_t^{(b)}(\omega), 1-\epsilon, 1+\epsilon\Big)\widehat{\text{Adv}}_t^{(b)}\Big)\right], \tag{39}$$

where the expectation is over initial observations and candidate groups sampled by the behavior policy.

Reward-weighted fine-tuning. At each post-training iteration, the policy generates fresh action candidates, which are evaluated through world-model rollouts. The resulting training records form $\mathcal{D}_{\text{off}}$, retaining the starting observations, native action chunks, predicted trajectories and returns, and sampling statistics required by the optimizer. The evaluated chunks also support fine-tuning with the policy's native action-training loss:

$$\min_\omega \mathbb{E}_{\mathcal{D}_{\text{off}}}\left[\sum_{b=1}^{N_{\text{cand}}} w_t^{(b)} \ell_{\text{act}}\Big(\omega; L, o_t, \mathbf{a}_t^{(b)}\Big)\right], \qquad w_t^{(b)} = \frac{\exp(J_t^{(b)}/\beta)}{\sum_{b'}\exp(J_t^{(b')}/\beta)}, \quad \beta > 0. \tag{40}$$

For a flow-based VLA, ℓ_{act} is its flow-matching loss. The training records retain native policy outputs when action conversion is required for world-model evaluation. Candidate actions and reward weights remain fixed during each update.

PPO within the learned simulator. A policy with a compatible latent input interface can instead generate actions recursively from predicted latents, robot states, and SOP instances. Its interactions with the world model produce multi-step trajectories and progress rewards. PPO applies the clipped objective in Eq. 39 to these transitions, using per-action probability ratios and value-based advantages estimated with V_ν, rather than group-relative chunk advantages.

After each update stage, the updated policy generates fresh model rollouts. The encoder, world model, physical readout, and code agent remain fixed throughout post-training; only the visuomotor policy and any associated value function are updated, without additional environment interactions.